

## Overview: Introductory Topics

- What is a computer?
- What is a program? What is programming?
- An Engineering Problem-Solving Methodology
- General format and components of a C program

1

## What is a Computer?

An electronic **machine** which can be programmed to carry out routine mental tasks by performing **simple operations** at very **high speed**.

### Simple operations:

add two numbers  
compare two numbers  
get the second letter of a word  
...

### High speed:

more than 10,000 million operations per second

2

**Computer = hardware + software**

### Hardware:

physical equipment used to perform computations: chips & circuits inside the machine, plus external devices (screen, keyboard, mouse, printer, network cables, ...)

### Software:

programs which control the computer and allow the user to perform useful tasks: word processor, accounting, library catalog, web browser, email, ...

**Hardware/Software distinction ⇒ computers are “reconfigurable”**

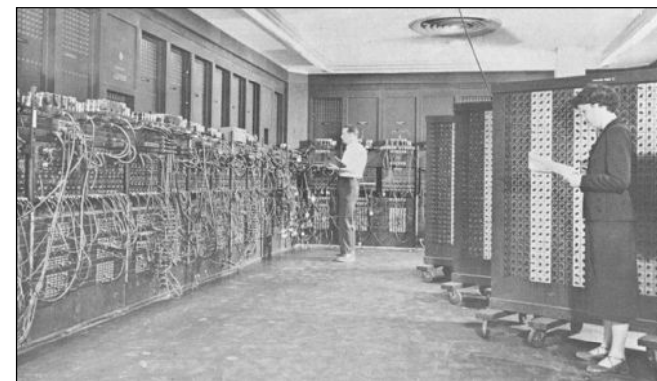
computers do whatever their programs tell them to do.

3

## Computers: history

First electronic computer was built in the 1930's by Dr. J. Atanasoff and C. Berry at Iowa State University.

1946 ENIAC: First large-scale general-purpose computer, University of Pennsylvania. 30 tons, 18000 tubes, 10×30 foot space.



4

## Computers: today / future

Computers are everywhere!

Trends: smaller, cheaper, more powerful, moveable, more useful(?)

Computers can be categorised by their size and performance:

- supercomputers
- mainframes
- workstations / personal computers
- handheld computers

Also have computer networks: Local Area Network, the Internet...

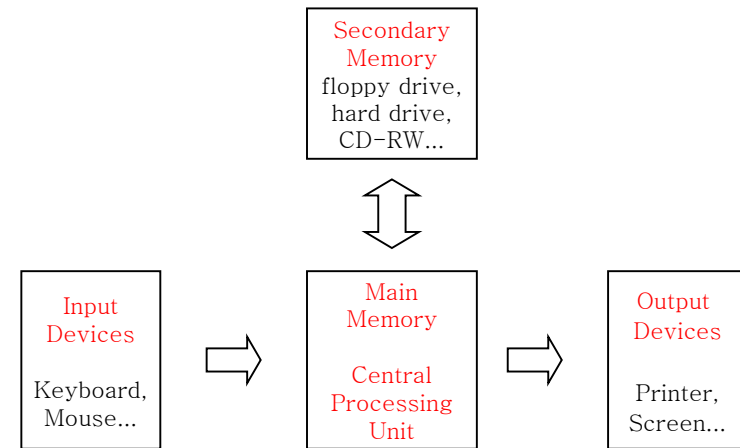
Computers will play a key role in our futures:

- human genome project
- medical diagnosis and treatment
- automated vehicles (cars, planes, ...)
- computer-based education
- and many others...

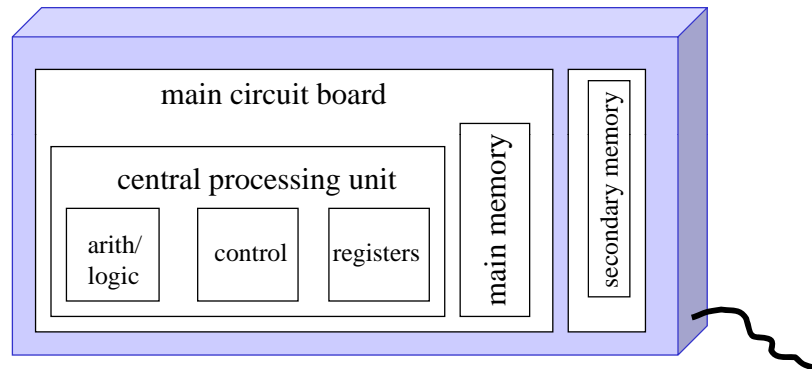
5

## Computer hardware

Most computers have the same basic architecture:



6



**Processor** (all in one “chip”)

**Central Processing Unit (CPU)**

**Arithmetic/Logic Unit (ALU)** - performs arithmetic

**Control Unit (CU)** - sequences operations

**Registers** - temporary data storage

**Speed** now measured in GHz (billions of cycles per second)

7

## Single Board Computer & Tiny Computer



Created by Harry H. Cheng, © 2009  
McGraw-Hill, Inc. All rights reserved

8

### Main Memory (separate chips on same “circuit board”)

- Temporarily holds programs (while being executed) and data.
- RAM (Random Access Memory) or ROM (Read-Only Memory)
- **Capacity** measured in MB (megabytes).
- Information stored in **binary** - sequence of **bits** (1's & 0's)  
8 bits = 1 **byte**,  $2^{10}$  bytes = 1 **kilobyte (KB)**  
 $2^{20}$  bytes = 1 **megabyte (MB)**,  $2^{30}$  bytes = 1 **gigabyte (GB)**

### Secondary Memory (hard or floppy disk, tape, CD-RW)

- Main Memory RAM loses its contents when switched off.
- Need to store programs and data on a more permanent basis  
⇒ use secondary memory.
- Size is usually measured in GB ( $2^{40}$  bytes = 1 **terabyte (TB)**).

### Peripheral Devices

Devices through which the computer communicates with the outside world e.g. keyboard, screen, mouse, printer, scanner, modem, ...

9

## Growth of Hardware

### Moore's Law (1965):

“The number of transistors that can be inexpensively placed on an integrated circuit increases exponentially, **doubling** approximately every **two** years.”

Based on Moore's law, every two years, the following approximately double:

- CPU speed at which computers execute their programs
- The amount of main memory
- The amount of secondary memory

10

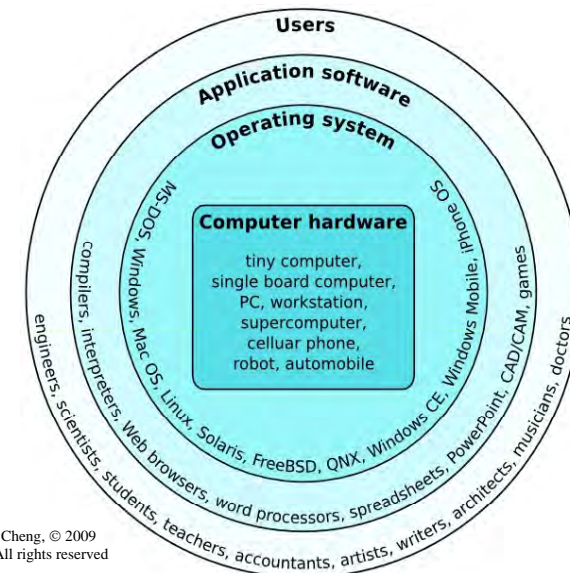
## Computer software

There are different kinds of software:

- **Operating system:** acts as interface between user programs or applications, and the hardware
  - Examples: Windows, DOS, Unix, Linux, ...
  - Example programs: copy file, save file, delete file, ...
- **Application software:** someone else wrote it for you
  - Examples: word processor, spreadsheet, web browser, ...
- **User programs:** what you write
  - e.g. write a C program to calculate the area of a circle with a given diameter
  - you'll see plenty of examples in the Practicals!

11

## Layers of Software



Created by Harry H. Cheng, © 2009  
McGraw-Hill, Inc. All rights reserved

12

## Computer software: Programming

To carry out some task, the computer must be told exactly what to do and **how to do it**.

An **algorithm** is the series of steps involved in carrying out a particular task. To carry out these steps, the algorithm must be expressed in a form that the computer can understand.

An algorithm expressed in such a form is called a **program**. The user can then tell the computer to **execute** or **run** this program.

A computer program is written in a **programming language**. “Natural” languages are for communicating with people. Programming languages are for communicating with computers.

13

## Programming Languages

• Computer only understands **machine language: strings of 1's and 0's**. Using machine language, a programmer can directly control the computer hardware.

- machine language is specific to the type of computer.
- machine language program is called **object code**.
- **extremely difficult**: requires large amounts of object code to accomplish even simple tasks; requires a detailed knowledge of the hardware.

14

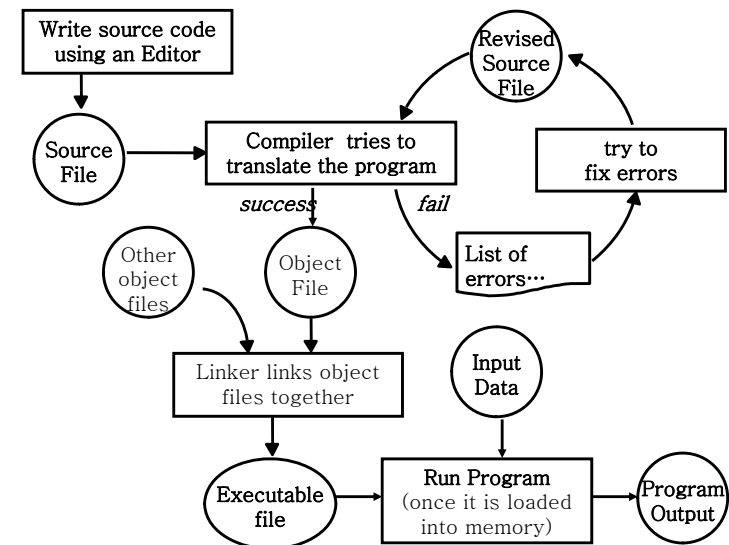
## Programming Languages (contd.)

• **High-level languages** were developed to make programming easier:

- precise enough so computer knows exactly what you mean, but don't need to worry as much about low-level details (e.g. how many bits are used to store the number “12”)
- high-level language program is called **source code** and is **machine-independent** and (ideally) **portable** from one type of computer to another
- However: computer doesn't understand high-level language programs. They must be **compiled** (translated) into machine language, by a **compiler** for the high-level language being used.
  - compiler is another example of application software

15

## Programming Flowchart



16

## Programming Languages: C

- There are hundreds of high-level programming languages: Pascal, Fortran, Basic, Ada, Lisp, Prolog, C, C++, Java, Cobol...
- All these languages are “equivalent”
  - but different languages are more/less convenient for particular tasks
- In this course, we’ll learn just one: C
  - C developed in 1972 by Dennis Ritchie at AT&T Bell Labs
  - C was designed to write the UNIX operating system
  - over the years, the power and flexibility of C have made it a very popular general-purpose programming language

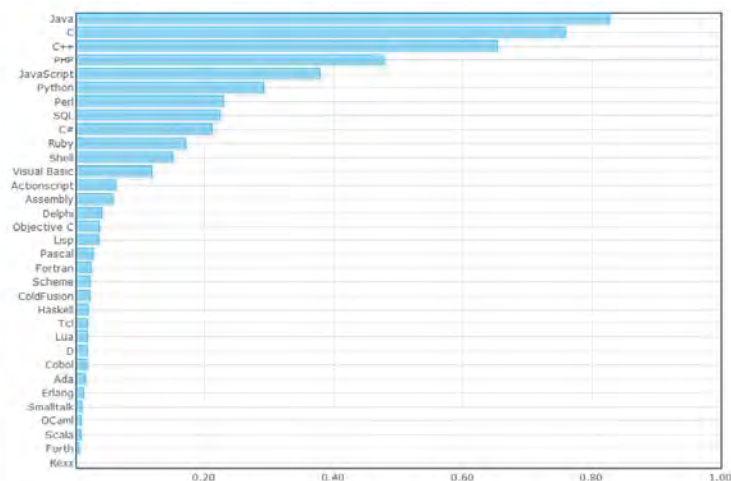
17

## Timeline for Major Programming Languages

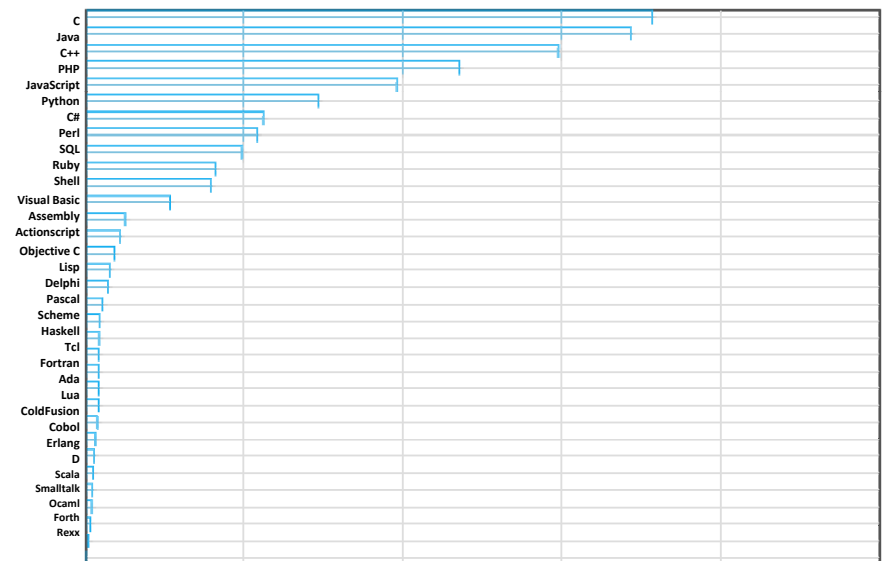
<b>FORTRAN</b>	John W. Backus, 1954
<b>BASIC</b>	George Kemeny and Tom Kurtz, 1964
<b>Pascal</b>	Nicolas Wirth, 1969
<b>C</b>	Dennis M. Ritchie, 1972
<b>C++</b>	Bjarne Stroustrup, 1979 (1983)
<b>Java</b>	Patrick Naughton, Mike Sheridan, and James Gosling of Sun, 1991
<b>C#</b>	Anders Hejlsberg, 2000

18

2011 Language Popularity <http://www.langpop.com>



2012 Language Popularity <http://www.langpop.com>



## Importance of C

C/C++ are dominant languages in industry

- Most large-scale projects are written in C/C++
- Most off-the-shelf software packages are written in C/C++
- The language of choice for real-time and embedded computing

TIOBE Programming Community Index for January 2012

<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

(Over 75% of languages in top 10 here)

1 Java	17.48%	-0.29%
<b>2 C</b>	<b>16.98%</b>	<b>1.15%</b>
<b>3 C#</b>	<b>8.78%</b>	<b>2.55%</b>
<b>4 C++</b>	<b>8.06%</b>	<b>-0.72%</b>
<b>5 Objective-C</b>	<b>6.92%</b>	<b>3.91%</b>
6 PHP	5.71%	-2.13%
7 (Visual) Basic	4.53%	-1.34%
8 Python	3.22%	-3.05%
9 Perl	2.77%	-0.08%
10 JavaScript	2.32%	0.73%

21

## What will you learn about C?

- with any programming language, there are 3 basic issues:
- **how can I write a correct program in this language?**
  - meaning: the compiler will generate an executable file from my source code, rather than a list of errors
  - you will learn how to do this for simple C programs
- **how can I solve problems using this programming language?**
  - meaning: how can I come up with a solution algorithm, and then accurately translate it into a correct program?
  - you will learn how to do this for small-scale problems
- **how can I solve problems using this programming language in a good/efficient/elegant/cheap/better-than-the-competition way?**
  - you will not learn much, if anything, about this here...

22

## An Engineering Problem-Solving Methodology

- before starting on the details of C programming, it is important to realise where programming “fits” into problem-solving in general
  - a big (and very common) mistake is to try to solve a problem by directly writing a program – only works (if at all) for experienced programmers on simple problems
- you should follow this procedure on every problem:
  1. **State the problem clearly.**
  2. **Describe the inputs and outputs.**
  3. **Work a simple example (mentally, by hand, calculator, ...)**
  4. **Develop an algorithm: a step-by-step outline of a solution.**  
**Then implement your algorithm as a C program.**
  5. **Compile & test your program (use more than 1 test case!)**

23

## Engineering Problem-Solving Methodology: Example

*Write a program to convert a price from Sterling to Euro.*

1. Problem statement:  
Write a program that will convert a number of Pounds Sterling to the equivalent number of Euro.
2. Inputs and outputs:
  - Input? `value_sterling`
  - Output? `value_euro`
  - What else? Conversion rate: 1 Euro=0.8277 Pounds Sterling
3. Simple example:
  - calculator: 10 Pounds Sterling converts to 12.08 Euro

24

## Engineering Problem-Solving Methodology: Example (contd.)

### 4. Algorithm Design:

- Get value in Pounds Sterling
- Convert value to Euro
- Display value in Euro

Refine steps?

“Convert value to Euro” becomes

- the value in Euro is equal to the value in Pounds Sterling divided by the conversion rate 0.8277

Implementation as a C program:

- Consider data requirements
- Convert each algorithm step into one or more C statements

25

## Engineering Problem-Solving Methodology: Example (contd.)

```
/* Date: 16/01/2012   Version: 1.0   */
/* Program to convert Pounds Sterling to Euro */

#include <stdio.h>                /* definition of printf and scanf */
#define CONVERSION_RATE 0.8277   /* Conversion constant */

main()
{
    float value_sterling,        /* input: value in Pounds Sterling */
          value_euro;           /* output: value in Euro */

    /* Get the value in Pounds Sterling */
    printf("Enter the value of the product in Pounds Sterling: ");
    scanf("%f",&value_sterling);

    /* Convert the amount of Pounds Sterling to Euro */
    value_euro=value_sterling/CONVERSION_RATE;

    /* Display the converted Euro value */
    printf("That equals %.2f Euro.\n",value_euro);
}
```

---

Sample Run:      Enter the value of the product in Pounds Sterling: 10.00  
                  That equals 12.08 Euro.

26

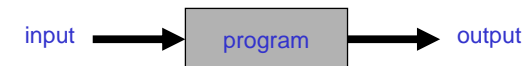
## Engineering Problem-Solving Methodology

### Program testing:

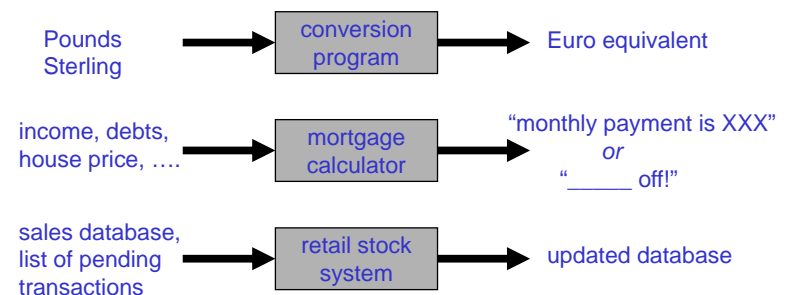
- first check you get correct answers in known, simple cases
- then check unusual (or “boundary”) cases
  - what if you enter 0 Pounds Sterling? Or a very high number?
- should also check impossible cases (if any), to make sure your program can handle them appropriately: **error message**, rather than program crashing
  - what if the above example should not accept a negative number of Pounds Sterling? (currently, it does!)
- how do you know when you’ve fully tested a program?
  - **In general: never.** But you should try!

27

Programs process some **input** and generate some **output**:



- **Input:** data the program will process -- entered by the user from the keyboard, read from a file, interpreted from mouse clicks, etc.
- **Output:** results produced by running the program. This can be displayed on the screen, written to a file, printed, etc.



28

## General format of a C program:

```
/* initial comments */
preprocessor directives
main()
{
    declarations; /* comments */
    statements; /* comments */
}
```

- C programs are *free-format* and *case-sensitive*
- comment starts with `/*` and ends with `*/` (don't forget it!)
- execution of a C program *always* starts with `main()`
- declarations *must* go before statements
- all declarations and statements *must* end with a `;`

29

## General format of a C program (contd.):

- C is free-format  $\Rightarrow$  comments, statements, etc. can begin anywhere on a line (e.g. doesn't have to be in column 1).
- however, style guidelines have been developed, e.g.
  - Use **whitespace** (spaces, blank lines, ...) to separate different components of your program.
  - Use indentation to convey information about relation of statements to each other (especially in **loops** – details later).
- C is case-sensitive: `value_sterling`, `Value_sterling`, `value_STERLING` are all different things.
  - by convention, use all-capitals for constants  
e.g. `CONVERSION_RATE`

30

## Comments in C

- Comments are ignored by the C compiler – they are purely for the benefit of the programmer, and anyone else who may read the source code.
- Comments are optional and therefore don't have to be included in a C program. However they can help to explain and document **what** the program does, and **why**.
- At a minimum, comments should explain the steps of the algorithm used to solve the problem being addressed.
- When evaluating your programs, we will take into account how appropriate (or otherwise) your commenting is...

31

## Preprocessor directives in C

- these are instructions your program gives to the C compiler
  - your program can re-use existing pieces of code which are stored in a **library**
  - the *standard C library* provides many useful services, e.g. `stdio.h` provides input and output facilities and is used in nearly every C program
  - how do you know what to **#include** ?
    - examples, textbooks, view standard C library...
- can also be used to **#define** constants – makes your program more readable and understandable
- preprocessor directives have a different **syntax** to C declarations / statements. In particular, they don't end in a `;`

32

## Declarations in C

- before using any variables in your program, you have to tell the C compiler about them so that later statements can use them
  - example: `float value_sterling` -- tells the compiler that your program will make use of a variable called `value_sterling`, and that the type of this variable is `float` (meaning: it may have a fractional part)
- a variable in C has a name (called its **identifier**), a **memory location** where it is stored, a **type**, and a **value**
  - identifier should be meaningful e.g. `x` instead of `value_sterling`
  - identifier, type, and value -- you decide
  - memory location -- the computer decides

33

## Statements in C

- statements are the detailed steps of what the program does
  - `value_euro=value_sterling/CONVERSION_RATE` tells the computer to take the current value of the variable `value_sterling` and divide it by the constant `CONVERSION_RATE`, and to store the result in the variable `value_euro` (over-writing the current value of `value_euro`)
- statements are not just for computations -- they also tell the computer when to ask the user for input, where to store the input data, what data to output and how, etc.
- good practice: one statement per line

34

## Overview: Fundamentals of C, part 1

- Some simple C programs
- C program syntax: identifiers, datatypes, punctuation
- Arithmetic operators

1

## A simple C program

What happens when you *compile* this program?

What happens when you *execute* (or *run*) this program?

```
/* Date: 18/01/2012   Version: 1.0 */
/* Program to calculate the area of a rectangle, given its length and width */

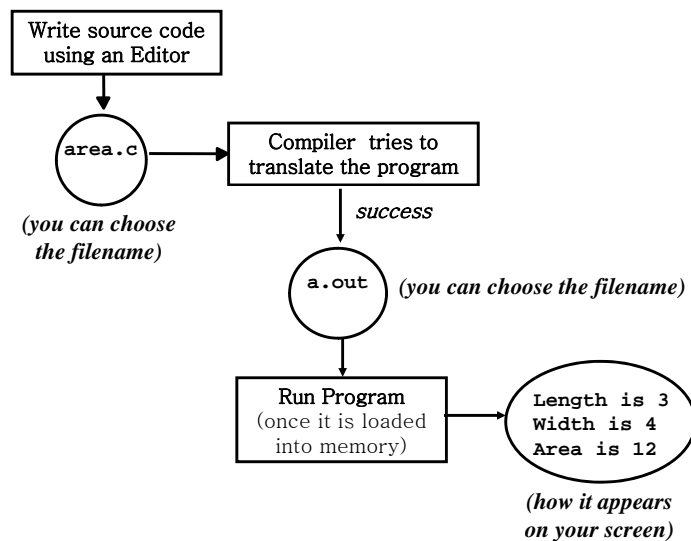
#include <stdio.h>                /* definition of printf and scanf */

main()
{
    int length=3;                 /* declare and initialise length to 3 */
    int width=4;                 /* declare and initialise width to 4 */
    int area;                   /* declare area, no initialisation */

    /* Calculate the area of the given rectangle */
    area = length * width;       /* now area has the value 12 */

    /* Output the data and results to the screen */
    printf("Length is %d\n",length); /* output: Length is 3 */
    printf("Width is %d\n",width);   /* output: Width is 4 */
    printf("Area is %d\n",area);     /* output: Area is 12 */
}
```

2



3

## A simple C program – variables

```
int length=3;
```

- This is the **declaration** of a **variable** called **length**, of type **int** (meaning: only holds integer values), and at the same time initialising **length** with the value 3
- Variables are a very important concept in C. In order to process data, we need a way of *storing* and *referring to* the data. Using variables, we can store data values in memory, and then use them in subsequent statements when the data is to be processed.
- A variable declaration reserves a memory location of the *appropriate size* in which the variable's data can be stored. In order to know how much memory to reserve, the computer needs to know what type of data (integer, floating-point, ...) will be stored.

4

## A simple C program – variables (contd.)

• recall: a variable in C has a name (called its **identifier**), a **memory location** where it is stored, a **type**, and a **value**:

- identifier should be meaningful  
e.g. `length` instead of `data1`, `x`, etc.
- identifier, type, and value -- you decide
- memory location -- the computer decides

• general syntax of a variable declaration: `type name;`  
• `type` can be `int`, `float`, `char`, ...

• optional (but a good idea): **initialise variable at declaration**

- general syntax: `type name = value;`
- `value` can be a constant, e.g. `int length = 3;`
- `value` can refer to earlier variables, e.g. `could` have had  
`int area = length * width;`

5

## A simple C program – assignment statement

```
area = length * width;
```

• the interpretation of this statement is: get the current values of the variables on the *right-hand* side, multiply these values, and store the result in the variable on the *left-hand* side.

• general syntax of an assignment statement:

```
varname = expression;
```

- `varname` is an already-declared variable
- `expression` is a piece of code that evaluates to some value

→ the computer evaluates the current value of `expression` and stores the result in `varname`, overwriting the current value of `varname`

6

## Assignment statement

**Warning:** in C, the assignment operator `=` does **NOT** mean equals (in the mathematical sense). It makes perfect sense to write

```
x = x * 2;
```

which is interpreted as: get the current value of `x`, multiply it by 2, and store the result back in `x`. *The effect is to double the value of x*

What are the final values of `x`, `y`, and `z` in the following?

```
int x = 4;
int y = 10;
int z = x;
x = y;
y = x;
```

```
int x = 4;
int y = x;
int z;
x = 2;
y = x;
z = y * 3;
```

7

## A simple C program – output to the screen

• how did the code `printf("Length is %d\n",length);`  
`printf("Width is %d\n",width);`  
`printf("Area is %d\n",area);`

produce the screen output

```
Length is 3
Width is 4
Area is 12
```

?

- usually, everything inside the double quotes " " is written as-is to the screen (the double quotes themselves are not).
- however, `%d` is a special character sequence which tells the computer to **replace** the `%d` with the value of the **corresponding integer variable** when writing to the screen.
- `\n` is an *escape sequence* which tells the computer to start the next output on a new line.
- in order to use `printf`, need: `#include <stdio.h>`

8

## A simple C program – main ( )

- recall: execution of a C program *always* starts with `main ( )`
- `main ( )` is an example of a **function** in C. In general, programs can contain a number of functions, each of which may be responsible for implementing a small part of the overall algorithm.
  - Example: write one function to get data input, one function for data output, and one or more functions to process the data. Then activate functions in the correct order to solve the problem!
  - the function `main ( )` is activated by the computer's *Operating System (OS)* when the program is run.
  - `void main (void)` means: `main ( )` receives no input data from the OS before the start of program execution, and returns no output data to the OS at program termination.
- the statements that make up a function are called the *body* of the function, and are enclosed in braces { }

9

## Another simple C program

```
/* Date: 18/01/2012   Version: 1.0 */
/* Program to calculate the area of a circle, given its radius */

#include <stdio.h>           /* definition of printf and scanf */
#define PI 3.1415927        /* definition of constant value for π */

void main(void)
{
    int diameter = 3;        /* declare and initialise diameter to 3 */
    float radius = diameter/2.0; /* declare and initialise radius to 1.5 */
    float area = PI * radius * radius; /* declare and calculate area */

    /* Output the data and results to the screen */
    printf("Diameter is %d\n", diameter); /* output: Diameter is 3 */
    printf("Radius is %.2f\n", radius); /* output: Radius is 1.50 */
    printf("Area is %.4f\n", area); /* output: Area is 7.0686 */
}
```

10

## Another simple C program – a more detailed look

- since `radius` could in general have a fractional part, it is declared as a `float`
- `area` must also be declared as a `float`, since in general it is also a floating-point number.
  - What happens if you declare them as `int` instead?  
Answer: their fractional parts are *truncated* (not rounded).
- `PI` is defined as a constant, so program can't change it. Makes the program more readable for us – computer doesn't care!
- `printf("Area is %.4f\n", area);` means that the floating-point variable `area` is to be written to the screen with 4 places of decimals (*rounding* is used if needed – as in this case).
  - `%.4f` is a *placeholder* which specifies the format in which the corresponding variable `area` will be output.

11

## Another simple C program – importance of variables

- could have written the previous program like this:

```
void main(void)
{
    float radius = 3 / 2.0; /* declare and initialise radius to 1.5 */
    float area = PI * radius * radius; /* declare and calculate area */
    (rest as before)
```

(i.e. dropped diameter and replaced it with its value 3). This results in more compact code and gives the same result as before. However, the above code is more difficult to understand.
- variables make programs easier to **read, debug, & understand**.
- a program can **change the value of a variable while the program is running**. The programs we've examined so far haven't done so, but we'll see examples of this later...

12

## Another simple C program – integer division

- `float radius=diameter/2.0;` why 2.0 and not 2?

- answer: `diameter` and 2 are both integers, and according to C, *the result of dividing one integer by another is also an integer* – even if that means truncating the result (i.e. dropping the fractional part).

- if we had `radius=diameter/2`, the value of `radius` would be 1 ( $3/2 = 1.5$ , then drop the fractional part).

- but if **any** of the quantities involved are floating-point, then the **entire calculation** is done using floating-point arithmetic. We can “force” the computer to do this by making one of the quantities of type `float`: this is why we had 2.0 instead of 2

13

## C program syntax

- when viewed as a typed document, a natural language (e.g. English) document consists of:

- words
- punctuation
- special symbols
- spaces and blank lines

- when viewed as a typed document, a C program consists of:

- keywords
- identifiers
- punctuation
- comments
- spaces and blank lines

14

## C keywords and identifiers

- C keywords: words which have special meaning to the compiler and are essentially the ‘building blocks’ of the language. These include

- `break, continue, else, enum, float, int, do, case ...`

- (this is only a partial list – see any C text for more keywords)

- when choosing an identifier (e.g. a variable name), you can only use uppercase and lowercase letters, the digits 0—9, and the underscore `_`

- further restrictions on choosing an identifier:

- **must** start with a letter or `_`
- followed by any mixture of letters, digits, and `_`
- **cannot** be a C keyword
  - also: maximum recommended identifier length = 31 characters

- examples of *invalid* identifiers: `3letter, int, Hello!, air-temp`

- examples of *valid* identifiers: `letter3, letter_3, _letter3`

- remember: C is case-sensitive – `letter` and `Letter` are different

15

## Datatypes in C

- a datatype defines a set of values and a set of operations on those values
  - operations: comparison of values, arithmetic operations, ...

- some standard C datatypes are predefined

- e.g. `float, double, int, long, short, char`

- `float` and `double` are for real numbers
- `int, long,` and `short` are for integers (i.e. whole numbers)
- `char` is used for characters

- a constant in C is simply a specific value:

- `7` (the integer value 7)
- `2345` (the integer value 2345)
- `4.2` (the floating-point value 4.2)
- `'a'` (the character 'a')
- `"abcdef"` (the character string "abcdef")

16

## Datatypes in C (contd.)

### Data Type: `int`, `long`, `short`

- not all integers can be represented, due to finite memory constraints  
e.g. `int`: 2 bytes wide, ranging from  $-2^{15}$  to  $(2^{15}-1)$   
`long`: 4 bytes wide, ranging from  $-2^{31}$  to  $(2^{31}-1)$   
`short`: 1 byte wide, ranging from  $-128$  to  $127$   
(on some computers, `int` gets 4 bytes, and/or `short` gets 2 bytes)

### Data Type: `float`, `double`

- represents real numbers
- integral part and fractional part separated by decimal point  
`float`: for single-precision floating point numbers  
`double`: for double-precision floating point numbers
- `float` and `double` are datatypes for real numbers but do not include all of them, again due to memory constraints

17

## Datatypes in C (contd.)

- floating-point can represent both integer and non-integer values  
e.g.  $-1.0$ ,  $0.05$ ,  $91.1$

**Scientific notation:**  $-1.0 \times 10^0$ ,  $5.0 \times 10^{-2}$ ,  $9.11 \times 10^1$  (*mantissa  $\times$  power of 10*)

**Exponential notation:**  $-1.0e0$ ,  $5.0e-2$ ,  $9.11e1$

- number of digits the computer allows for the decimal part of the mantissa determines the *precision* of the floating-point representation.
- number of digits the computer allows for the exponent determines the *range* of the floating-point representation.

18

## Datatypes in C (contd.)

- datatype hierarchy:**  

<code>double</code>	↑ <i>high</i>
<code>float</code>	
<code>long</code>	
<code>int</code>	
<code>short</code>	

  
*low*

- no information is lost if a value is moved to a higher-order datatype, e.g.

```
double x;  
x=10;
```

After this assignment, `x` has the value 10.0

- information may be lost if a value is moved to a lower-order datatype, e.g.

```
int a;  
a=12.8;
```

After this assignment, `a` has the value 12 (due to truncation).

19

## The `char` datatype

- represents individual character value: letter, digit, special symbol e.g. `+`, `$`, `.`, `@`
- A single character constant is enclosed in single quotes e.g. `'A'`, `'b'`, `'1'`, `' '`

Character variables are typically assigned one byte of storage. They are stored using a standard “encoding”, such as **ASCII** (American Standard Code for Information Interchange) or **EBCDIC** (Extended Binary Coded Decimal Interchange Code). Both are ways of assigning each character to a unique byte sequence of 1’s and 0’s.  
Examples: ASCII for ‘Z’ is 01011010, ASCII for ‘4’ is 00110100. In decimal: ASCII for ‘Z’ is 90, ASCII for ‘4’ is 52.

**Characters can be treated as integers!** Each `char` is stored using these binary codes just like integers, and so we can manipulate them in the same way that we can manipulate other integers:

```
char c = 'a'; /* OR: char c = 97 */  
while (c <= 'z') { /* OR: while (c <= 122) */  
    printf("%c", c);  
    c++;  
}
```

Produces the screen output: `abcdefghijklmnopqrstuvwxy`

This works because in ASCII, a-z follow each other in sequence (so do 0-9 and A-Z)

20

## Punctuation in C

- Semi-colon ; --every C statement must end with ;
- Full-stop or “dot” . --used as the **decimal point** for real numbers.
- Comma , --used to separate elements of a **list**. For example we can declare a number of variables at the same time (if they are all of the same type):  

```
int celsius, fahrenheit, count;
```
- Single quote ' --used to indicate single characters.
- Double quote " --used to indicate **strings** (sequences of characters).
- Braces { } --used to group a number of statements into a single, compound statement.
- Parentheses ( ) --used to group certain items such as arguments to functions, or to indicate the order of calculation in arithmetic expressions.

21

## C program layout

- the C compiler doesn't care about spaces, indentation, identifiers, line breaks, and other features of program layout...

```
#include <stdio.h>
#define CON1 0.88288
main() {float v1,v2;printf("Enter the value of the product in Sterling: ");scanf("%f",&v1);
v2=v1/CON1;printf("That equals %.2f Euro.\n",v2);}
```

- ... but people do!

*equivalent*

```
#include <stdio.h>
#define CONVERSION_RATE 0.88288
main()
{
float value_sterling, value_euro;

printf("Enter the value of the product in Sterling: ");
scanf("%f",&value_sterling);

value_euro=value_sterling/CONVERSION_RATE;

printf("That equals %.2f Euro.\n",value_euro);
}
```

22

## Arithmetic operators in C

- + (addition)
- (subtraction)
- \* (multiplication)
- / (division – value of  $3/2.0$  is 1.5, value of  $3/2$  is 1  
Also: value of  $2/7$  is 0)
- % (remainder – value of  $5\%2$  is 1, value of  $4\%2$  is 0,  
value of  $2\%7$  is 2)

**Warning:**  $a/b$  or  $a\%b$  will cause a **run-time error** if  $b$  is 0

- meaning: C compiler won't see an error, but during program execution, if the value of  $b$  is 0 when the above division or remainder operations are applied, the program will fail.

23

## Arithmetic operators in C (contd.)

- Examples:

```
rectarea = length * width;
circarea = radius * radius * PI;
root1 = (-b + sqrt(b*b-4*a*c))/(2.0*a);
```

- parentheses used to explicitly order arithmetic:

```
w = x - y / z;
w = (x - y) / z;
w = x - (y / z);
```

*expression in parentheses  
is evaluated first.  
nested parentheses: evaluated  
from the inside out.*

- without parentheses: \*, / before +, - and left-to-right between operators with same **precedence** (e.g. \* and /)

24

## Arithmetic operators in C – Exercise

What is the assigned (left-hand side) value in each case?

```
int s, m=3, n=5, r, t;
float x=3.0, y;

t = n/m;
r = n%m;
y = n/m;
t = x*y-m/2;
x = x*2.0;
s = (m+n)/r;
y = -n; /* unary operator */
```

(unary +, - have higher *precedence* than \*, /, %, which in turn have higher *precedence* than binary +, -)

25

## Arithmetic operators in C – operator precedence

Given the radius of a sphere R, the volume of the sphere is  $(4/3)\pi(R^3)$

Suppose we have a statement `float radius;`

Which of the following C statements calculates this volume correctly, and why?

```
float vol = 4/3 * PI * radius * radius * radius;      NO
float vol = PI * radius * radius * radius * 4/3;      YES
float vol = (4/3) * PI * radius * radius * radius;    NO
float vol = PI * radius * radius * radius * (4/3);    NO
float vol = (4.0/3) * PI * radius * radius * radius;  YES
float vol = 4.0/3 * PI * radius * radius * radius;    YES
float vol = PI * radius * radius * radius * 4/3.0;    YES
```

26

## Operators and expressions

What is the value of each of these C expressions?

```
5 * (3 / 5.0)
```

```
5 * (3 / 5)
```

```
5 * 3 / 5
```

As seen before, we can “force” the computer to use floating-point arithmetic by using `5.0` instead of `5` (if this is the desired behaviour). But what if the quantities involved in the expression are variables, rather than constants? For example, how would you ensure floating-point calculations in this case:

```
int i=3, j=5;
float k;
k = j*(i/j); /* want k to be assigned the value 3.0 */
```

We can't just write `i.0` or `j.0`...

27

## Type casting in C

To “temporarily” force the computer to regard a variable of a particular type as having a *different* type, put the desired type in parentheses before the variable name.

Examples:

```
(float)i forces the computer to temporarily regard i as a float
(int)x forces the computer to temporarily regard x as an int
```

**Note:** the type of the variable being cast is NOT changed (e.g. if `i` had been declared as an `int`, it is still an `int` – it is just treated as a `float` for the duration of the statement in which it is cast to type `float`).

Type casting is not just for variables – can also be applied to constants, e.g.

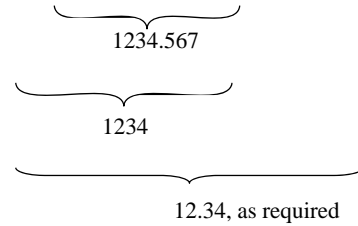
```
(int)6.5 is the integer “6”
(float)5 is the floating point number “5.0”
```

28

## Type casting in C – an example

Strip all but 2 decimal points from a float (*not rounding*): 12.34567 => 12.34

```
float trimmed = ((int)(100*number))/100.0 ;
```



(this can be generalised to stripping all but N decimal points, for any value of N – details later)

## Arithmetic operators in C – other operators

### Increment and decrement operators:

`x++ ;` is equivalent to `x = x + 1 ;`  
`y-- ;` is equivalent to `y = y - 1 ;`

### Compound (or abbreviated) assignment operators:

`x += 3 ;` is equivalent to `x = x + 3 ;`  
`sum -= y ;` is equivalent to `sum = sum - y ;`  
`product *= z ;` is equivalent to `product = product * z ;`  
`d /= 4.5 ;` is equivalent to `d = d / 4.5 ;`  
`r %= 2 ;` is equivalent to `r = r % 2 ;`

**In general:** `identifier operator= expression ;`

means `identifier=identifier operator (expression) ;`

(Note: **expression** is evaluated first in these compound assignments)

## Overview: Fundamentals of C, part 2

- Input and output with C
- Conditionals and Relational operators
- Logical operators

1

## Output with C – printf()

```
int length=3, x=1, y=2;
float area=length*2;
printf("Hello World.\n");
printf("the value of 1+1 is %d\n",2);
printf("the value of 1+1 is %d\n",1+1);
printf("the value of y/x is %d\n",y/x);
printf("Length is %d\n",length);
printf("Area is %.4f\n",area);
printf("x is %d and y is %d\n",x,y);
```

```
screen output:
Hello World.
the value of 1+1 is 2
the value of 1+1 is 2
the value of y/x is 2
Length is 3
Area is 6.0000
x is 1 and y is 2
```

- **placeholders** (e.g. %d, %.4f) are substituted with the values of their corresponding **arguments** (left-to-right) when output sent to the screen
  - argument can be constant, variable, or expression
  - if the placeholder type doesn't match the type of the corresponding argument: program compiles ok, but garbage values output for this and subsequent placeholders
  - if there are more placeholders than arguments: program compiles ok, but garbage values output for the unmatched placeholders
  - if there are fewer placeholders than arguments: no problem!

2

## Output with C (contd.)

```
int value=-145;
printf("value is %d units\n",value);
printf("value is %i units\n",value);
printf("value is %4d units\n",value);
printf("value is %3d units\n",value);
printf("value is %6d units\n",value);
printf("value is %-6d units\n",value);
```

```
screen output:
value is -145 units
value is -145 units
value is -145 units
value is -145 units
value is -145 units
value is -145 units
```

**Notes:** minimum field width will be *increased if necessary* to print the value  
%- means *left justify*  
%+ means *always print the sign (+ or -)*

```
double dbl=157.8926;
printf("dbl is %f units\n",dbl);
printf("dbl is %6.2f units\n",dbl);
printf("dbl is %+8.2f units\n",dbl);
printf("dbl is %7.5f units\n",dbl);
printf("dbl is %e units\n",dbl);
printf("dbl is %.3E units\n",dbl);
```

```
screen output:
dbl is 157.892600 units
dbl is 157.89 units
dbl is +157.89 units
dbl is 157.89260 units
dbl is 1.578926e+02 units
dbl is 1.579E+02 units
```

3

## Output with C – escape sequences

used to represent characters which would be awkward or impossible to enter directly into a source program:

\a	alert (bell)
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\?	question mark
\'	single quote
\"	double quote

4

## Input with C – scanf()

```
float value_punt;
int intvar1, intvar2;
double dblvar;

printf("enter a value for value_punt: ");
scanf("%f",&value_punt);

printf("enter a value for intvar1: ");
scanf("%d",&intvar1);

printf("enter a value for intvar2: ");
scanf("%i",&intvar2);

printf("enter a value for dblvar: ");
scanf("%lf",&dblvar);
```

**&** is the unary address-of operator

5

## Input with C (contd.)

more than one input value at a time:

```
double distance, velocity;
printf("enter values for distance and velocity: ");
scanf("%lf %lf",&distance, &velocity);
```

space between %lf's means: input values will be separated by *one or more whitespace characters* (e.g. blank space, <CR>, tab)

**scanf()** is easy to get wrong – if placeholders and arguments don't match exactly, the results are "unpredictable". Be careful!

6

## Conditionals in C

- all programming languages have some sort of **conditional** statement: do different things depending on some **condition** or **test**

- **if** statement (the most common conditional statement):

```
if (numteams > 2) {
    numteams = 2;
    printf("You had too many teams!\n");
}
```

**general syntax:**

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

the **statements** are executed if the **condition** evaluates to **true**

- if **condition** evaluates to a **non-zero** value: **true**
- if **condition** evaluates to **zero**: **false**

7

## Relational operators in C

Relational operators – for building the conditions:

==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

**Warning: do not confuse == (equality) and = (assignment)**

8

## Conditionals and relational operators in C – examples

```
int a = 20;
if (a != 10) {
    printf("a is not equal to 10\n");
}
if (a > 10) {
    printf("a is greater than 10\n");
}
if (a == 10) {
    printf("a is equal to 10\n");
}
```

produces the screen output: a is not equal to 10  
a is greater than 10

9

## Conditionals and relational operators in C – examples

```
int x = 10;
int y = 10*x;
if (y > x) {
    if (x < 100) {
        if (y > 80) {
            printf("A");
        }
        printf("B");
    }
    printf("C");
}
printf("D\n");
```

produces the screen output:

10

## Conditionals and relational operators in C

- convert mark to grade: 70 & over is A, under 40 is C, otherwise B

```
float mark;
char grade; /* character datatype */
printf("enter an exam mark: ");
scanf("%f", &mark);
if (mark >= 70) {
    grade = 'A';
}
if ((mark >= 40) && (mark < 70)) { /* logical AND */
    grade = 'B';
}
if (mark < 40) {
    grade = 'C';
}
printf("The grade is: %c\n", grade);
```

Problem? All 3 conditions are tested regardless of the value of **mark**. But if 1<sup>st</sup> condition is **true**, no point in testing 2<sup>nd</sup> and 3<sup>rd</sup>; if 2<sup>nd</sup> is **true**, no point in testing 3<sup>rd</sup>

11

## Conditionals and relational operators in C – if-else

- **else** specifies alternative action to take if condition evaluates to **false**

```
if (condition) {
    statements; /* executed if condition true */
} else {
    statements; /* executed if condition false */
}
```

so previous program could be re-written:

```
if (mark >= 70) {
    grade = 'A';
} else {
    if ((mark >= 40) && (mark < 70)) {
        grade = 'B';
    } else {
        if (mark < 40) {
            grade = 'C';
        }
    }
}
```

12



## Logical operators and expressions

What is the value of `y` in each case?

```
int i=3, j=5, x=5, y;
```

```
y = i && j;
```

```
y = i + j > 10 || x < 1;
```

```
y = !! j;
```

**Note:** when evaluating a condition, C interprets *non-zero value* as **True** and *zero value* as **False**. However, when a condition is evaluated to **True**, C assigns it the numerical value **1**; when it is **False**, C assigns it the numerical value **0**.

17

## Operator precedence in C

OPERATOR PRECEDENCE (partial list)	ASSOCIATES
!	R-to-L
*(multiplication) / %	L-to-R
+(addition) -(subtraction)	L-to-R
< > <= >=	L-to-R
== !=	L-to-R
&&	L-to-R
	L-to-R
=	R-to-L

- if you have in your program:

```
I + 1 > 5 || J - 1 < 10 && K < I + J
```

- then the C compiler interprets this expression as:

```
((I+1) > 5) || (((J-1) < 10) && (K < (I+J)))
```

- putting in explicit parentheses is always a good idea!

18

## Overview: Loops

- **while and for loops**
- **infinite and unbounded loops, nested loops**
- **do-while loops**
- **multiple-alternative decisions: switch**

1

## Why are loops needed?

- Because sometimes we want to repeat some processing step(s) many times.
  - Example: we saw earlier a C program to compute the area of a circle, given its diameter. Suppose we wanted to find the areas of 50 circles, each with a given diameter.
    - We could run our earlier program 50 times, once for each circle.
    - Or we could re-write our earlier program to sequentially find the area of each of the 50 circles (using `diameter1`, `diameter2`, ... `diameter50`, `area1`, `area2`, ... `area50`) – but this new program has 50 times as much code as our earlier program (what if we wanted the areas of 1,000 circles?! or 1,000,000, or...)
    - Better: re-write our earlier program to repeat the relevant section 50 times.
- A **loop** is the fundamental programming concept that allows this repetition (also called *iteration*).

2

## The while loop

```
/* crack a dozen eggs for a large omelette */
int n_eggs = 1;
while (n_eggs <= 12) {
    printf("cracked egg number %d\n", n_eggs);
    n_eggs = n_eggs + 1; /* or: n_eggs++; */
}
printf("Finished cracking the dozen eggs\n");
```

termination condition

loop body

Produces the screen output:

```
cracked egg number 1
cracked egg number 2
cracked egg number 3
cracked egg number 4
cracked egg number 5
cracked egg number 6
cracked egg number 7
cracked egg number 8
cracked egg number 9
cracked egg number 10
cracked egg number 11
cracked egg number 12
Finished cracking the dozen eggs
```

3

## The while loop: details

- Loops are another form of conditional statement. In the case of a loop, the computer uses the *termination condition* to determine whether the statements associated with the loop should be repeated or not.
- These statements are referred to as the *loop body*. The loop body can be a single or a compound statement.
- The general form of a **while** loop is as follows:

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

What happens: The loop body is executed only if the condition is **true**. After the loop body is executed, the condition is re-evaluated; if it is still **true** the loop body is executed again; and so on, and so on, until the condition is **false**.

*Note: If a condition evaluates to false before executing the loop body, the loop body is not executed at all.*

4

## The while loop: details (contd.)

[Aside: in certain situations the condition will never evaluate to **false** and the loop will continue to execute endlessly. Such a loop (usually, though not always, a programming error!) is called an *infinite* loop. An infinite loop may be terminated by interrupting the program or by switching off the computer, both of which terminate the program as well. To interrupt a program press Ctrl/C (and it might work!)]

while loop example: sum up the integers from 1 to 6

```
int i = 1, sum = 0;
while (i < 7) { /* or: while (i <= 6) */
    sum = sum + i;
    i = i + 1; /* or: i++; */
}
printf("Sum is %d\n", sum);
```

which produces the screen output:      **Sum is 21**

5

## The while loop: Example (contd.)

- The loop body is executed only if the condition is **true**. Since we have initialised **i** to 1, the condition (**i < 7**) evaluates to **true** and the loop body is executed. In the loop body, a running total for **sum** is calculated by adding the value of **i** to **sum**. The first time through, the variable **sum** is assigned the value 1. The variable **i** is then increased by 1. We then test the condition again.
- The variable **i** now has the value 2 so the condition is still **true**. Therefore the loop body is executed again, assigning **sum** the value 3 (i.e. 1+2) and increasing **i** to 3.
- Next time around **sum** becomes 6 and **i** becomes 4; next time **sum** becomes 10 and **i** becomes 5; next time **sum** becomes 15, **i** becomes 6; next time **sum** becomes 21, **i** becomes 7.
- Since **i** has reached 7, the condition is now **false**, and the loop exits, so the output statement is executed.

6

## The while loop: Another example

```
/* C code to sum 10 pairs of integers */
int num1, num2, result;
int count = 0;
while (count < 10) {
    printf("Enter first integer: ");
    scanf("%d", &num1);
    printf("Enter second integer: ");
    scanf("%d", &num2);
    result = num1 + num2;
    printf("Result is %d\n", result);
    count++;
}
```

If we omitted the **count++** statement, what would happen?

Answer: **infinite loop** ("logical error" – not a syntax error. The program will compile, but will execute incorrectly).

7

## The while loop: unbounded loop

- in the 2 previous examples, it is very important that the *loop control variables* **i** and **count** are initialised to appropriate values in order that the loops operate correctly.
- however, often you will not know in advance how many times to execute the loop body:
  - suppose that we want to generalise the previous program so that it continues calculating for as long as the user requires. The user may wish to sum one pair of numbers, or may wish to sum 100 pairs of numbers. We can code it so that the user indicates that they are finished by entering -1 as the first number.
  - this type of loop is known as an **unbounded loop** – it will terminate at some point, but we don't know exactly how many iterations are going to be required.

8



### 3 for loops and their while equivalents

0+1+2 +...+9	<pre>int total = 0, i; for (i = 0; i &lt; 10; i++) {     total += i; }</pre>	<pre>int total = 0, i = 0; while (i &lt; 10) {     total += i;     i++; }</pre>
ten sums	<pre>int count, num1, num2, result; for (count=0; count&lt;10; count++) {     printf("Enter 1st number: ");     scanf("%d",&amp;num1);     printf("Enter 2nd number: ");     scanf("%d",&amp;num2);     result = num1 + num2;     printf("The sum is %d\n",result); }</pre>	<pre>int count = 0, num1, num2, result; while (count &lt; 10) {     printf("Enter 1st number: ");     scanf("%d",&amp;num1);     printf("Enter 2nd number: ");     scanf("%d",&amp;num2);     result = num1 + num2;     printf("The sum is %d\n",result);     count++; }</pre>
*	<pre>int nstars, stars; for (nstars=1; nstars&lt;=10; nstars++) {     for (stars=1; stars&lt;=nstars; stars++) {         printf("****");     }     printf("\n"); }</pre>	<pre>int nstars = 1, stars; while (nstars &lt;= 10) {     stars = 1;     while (stars&lt;=nstars) {         printf("****");         stars++;     }     printf("\n");     nstars++; }</pre>

### The for loop

- The **for** loop forces the programmer to explicitly initialise the counter, to provide a condition for loop continuation, and to update the counter.

<p><b>general syntax</b></p> <pre>for (initialise; condition; update) {     statements; }</pre>
---

```
int stars;
for (stars = 0; stars < 10; stars++) {
    printf("***");
}
```

```
int stars = 0;
while (stars < 10) {
    printf("***");
    stars++;
}
```

← **Equivalent: both produce the screen output \*\*\*\*\***

- initialise** executed only once: when the loop is first encountered. Usually this sets a variable to some initial value (often called a "loop counter").
- condition** is then tested. This usually tests the value of the counter.
- If **true**, then the loop body is executed. If **false**, the **for** loop exits.
- update** called *after* each time around the loop, and we start over.

### Specifying number of iterations

Each of these loops produces the same screen output: \*\*\*\*\*

```
int i;
for (i=0; i <= 4; i++){ /* i increases by 1 each iteration */
/* condition True when i is 0,1,2,3,4 */
/* then False when i is 5 */
    printf("***");
}
```

```
int i;
for (i=0; i < 10; i=i+2){ /* i increases by 2 each iteration */
/* condition True when i is 0,2,4,6,8 */
/* then False when i is 10 */
    printf("***");
}
```

```
int i;
for (i=100; i > 95; i--){ /* i decreases by 1 each iteration */
/* condition True when i is 100,99,98,97,96 */
/* then False when i is 95 */
    printf("***");
}
```

**Exercise: re-write the above as while loops**

### for loop: syntax details

```
int j;
for (j = 0; j < 10; j++) {
    printf("****");
}
```

**produces the screen output \*\*\*\*\***

What would happen if the for statement was changed as follows:

```
for (j = 0; j < 10; ) {
```

```
for (j = 0; -1; j++) {
```

```
for (j = 0; ; j++) {
```

```
for (j; j < 10; j++) {
```

```
int j = 0; for ( ; j < 10; j++) {
```

## Nested for loop: multiplication table

We want a C program which prints out the following multiplication table:

```
    1  2  3  4  5  6  7  8  9
1  1
2  2  4
3  3  6  9
4  4  8 12 16
5  5 10 15 20 25
6  6 12 18 24 30 36
7  7 14 21 28 35 42 49
8  8 16 24 32 40 48 56 64
9  9 18 27 36 45 54 63 72 81
```

Recall the *Engineering Problem-Solving Methodology* from the Introductory Topics lecture: let's assume that the first 3 steps (clear problem statement, describe inputs & outputs, work a simple example) have been done – now we have to develop an algorithm and then implement it in C...

17

## Nested for loop: multiplication table (contd.)

- Have to compute table row-by-row.
- The first row prints out the headings for the table. This is straightforward.
- For each subsequent row, we will print out the row number and then print out the list of multiples. So an initial attempt is:
  1. Print out row of integers from 1 to 9
  2. Loop for Counter = 1 to 9
    3. Print out Counter multiples of Counter

*e.g. if Counter is 6, print 6 multiples of 6: 1\*6,2\*6,...,6\*6*
- Step 3 clearly requires further refinement:
  1. Print out row of integers from 1 to 9
  2. Loop for Counter = 1 to 9
    - 3.1 Move onto new line
    - 3.2 Print value of Counter
    - 3.2 Loop for Counter2 = 1 to Counter
      - 3.3 Print out Counter\*Counter2
- Step 1 also requires a bit more work:
  - 1.1 Loop for Counter = 1 to 9
    - 1.2 Print out Counter

18

## Nested for loop: multiplication table (contd.)

```
int counter, counter2;
for (counter = 1; counter <= 9; counter++) {
    printf("\t%d", counter); /* \t means horizontal tab */
}
for (counter = 1; counter <= 9; counter ++ ) {
    printf("\n");
    printf("%d", counter);
    for (counter2 = 1; counter2 <= counter; counter2++) {
        printf("\t%d", counter*counter2);
    }
}
```

Things for you to think about: how would you modify this code so that

- the user can choose the final value of **counter** (e.g. 7 instead of 9)
- the user can choose the operation to be performed (e.g. division or addition, rather than multiplication)

19

## The for loop: Exercise

```
int i;
for (i=0; i<=10; i++){
    i += 2;
    printf("value of i is %d\n", i);
}
```

Produces the screen output:

20

## do-while loop

• The only difference between this and a **while** loop is that the condition is checked at the *end* of the loop body, rather than at the beginning.

• General syntax of a **do-while** loop:

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

### Example:

produces the screen output:

```
int j = 0;
do {
    printf("j is %d\n", j);
    j++;
} while (j <= 5);
```

j is 0  
j is 1  
j is 2  
j is 3  
j is 4  
j is 5

The main difference between a **while** and a **do-while** is that the body of the **do-while** will **always** be executed **at least once**, whereas the body of the **while** may not be executed at all if the condition is found to be **false** first time around.

21

## Be careful with loops!

• what do these loops do?

```
for (stars = 0; stars <= 100; stars--) {
    printf("*");
}

for (j = 0; j > 0; j++) {
    printf("*");
}
```

• Note that all loops have 3 parts:

(1) initialisation, (2) condition, (3) update

(**for**: part of syntax; **while**, **do-while**: up to you)

**DO NOT START CODING A LOOP UNTIL YOU KNOW EXACTLY WHAT EACH SHOULD BE!!!**

if not careful: loop won't terminate, won't do anything, won't do what you expect...

22

## Loops are not just for numbers...

```
char ch = 'y';
while (ch == 'y') {
    /* something interesting */
    printf("Go again (y/n)?");
    scanf(" %c",&ch);
}
```

*Note blank space before %c – needed to fix a problem with scanf when making multiple reads...*

```
char ch;
for (ch = 'y'; ch == 'y'; scanf(" %c",&ch)) {
    /* something interesting */
    printf("Go again (y/n)?");
}
```

23

## Multiple-alternative decisions

Sometimes there are multiple possible paths the program could follow.

Example: mark  $\geq 70$ , First Honours;  $60 \leq$  mark  $< 70$ , Second Honours;  $40 \leq$  mark  $< 60$ , Pass; mark  $< 40$ , Fail.

You could write a C program to do this using just **if** statements, or just **if-else** statements (nested). But there is another way:

```
if (mark >= 70) {
    printf("First Honours\n");
} else if (mark >= 60) { /* know mark < 70 here */
    printf("Second Honours\n");
} else if (mark >= 40) { /* know mark < 60 here */
    printf("Pass\n");
} else { /* know mark < 40 here */
    printf("Fail\n");
}
```

24

## Multiple-alternative decisions (contd.)

As usual, there is more than one correct solution:

```
if (mark<40){
    printf("Fail\n");
} else if (mark<60){ /* know mark≥40 here */
    printf("Pass\n");
} else if (mark<70){ /* know mark≥60 here */
    printf("Second Honours\n");
} else{ /* know mark≥70 here */
    printf("First Honours\n");
}
```

25

## Multiple-alternative decisions: switch

This is another flow of control statement. It is not essential as anything that can be done with a **switch** can be done with **if-else** statements.

However it is easier to use and to read.

**switch** general syntax :

```
switch (expression) {
    case const1: statements;
    case const2: statements;
    . . .
    default : statements;
}
```

There can be as many **const**'s as you like. **expression** is evaluated and its value is compared to **const1**, **const2**... If any of them have the same value as **expression**, then the **statements** associated with this **const** (and all subsequent **statements** in the body of the **switch**) are executed. If there is a **default** present, then its **statements** are executed when no matching value is found. If there is no **default** and no matching value, then the **switch** statement does nothing.

26

## Multiple-alternative decisions: switch (contd.)

The cases are not exclusive as this example shows:

```
int i;
for (i = 0; i <= 5; i++) {
    switch(i) {
        case 1:
        case 2: printf("1 or 2\n");
        case 4: printf("4\n");
        default: printf("default\n");
    }
}
```

} default  
1 or 2  
4  
default  
1 or 2  
4  
default  
default  
4  
default  
default

The loop cycles with values of **i** from 0 to 5. 0 gives the default. 1 however triggers **case 1** and rather than this having no effect, this causes all the **printf** statements to be executed. This is because the **switch** selects one **entry point** and then executes all statements following that entry point. When **i** is 4, the entry point is **case 4** and so the last 2 messages are printed.

27

## Multiple-alternative decisions: switch (contd.)

Usually we want to express *mutually exclusive* courses of action, and this can be done by using **break** statements in the **switch**.

```
int i;
for (i = 0; i <= 5; i++) {
    switch(i) {
        case 1:
        case 2: printf("1 or 2\n");
                break;
        case 4: printf("4\n");
                break;
        default: printf("default\n");
                break; /* good practice */
    }
}
```

} default  
1 or 2  
1 or 2  
default  
4  
default

Note: **break** means "immediately halt execution and exit this **compound statement** (loop or switch)", so it can also be used with loops.

28

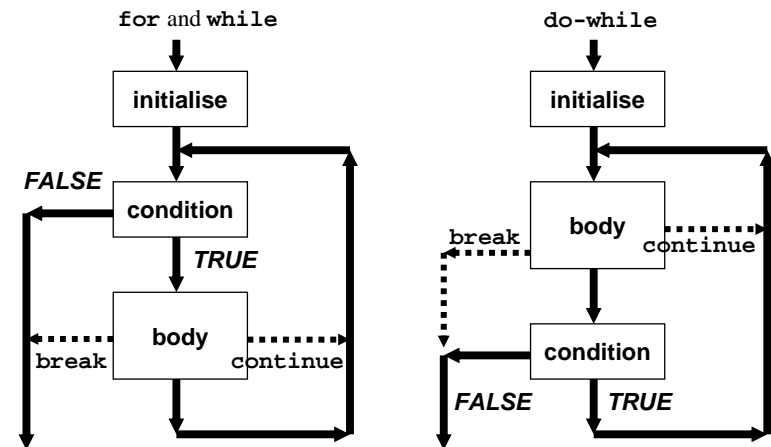
## Multiple-alternative decisions: switch (contd.)

The expression controlling the **switch** can be of any integral type.  
This includes **char**... For example:

```
char ch;
printf("enter your letter: ");
scanf("%c", &ch);
switch(ch) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    printf("A vowel\n");
    break;
  default:
    printf("A Consonant\n");
    break;
}
```

29

## break vs continue, for/while vs do-while



**break** means "immediately halt execution of this loop"

– often useful for complicated/multiple/exceptional loop exit tests

**continue** means "skip to next iteration of this loop"

– occasionally useful when loop body is very complicated

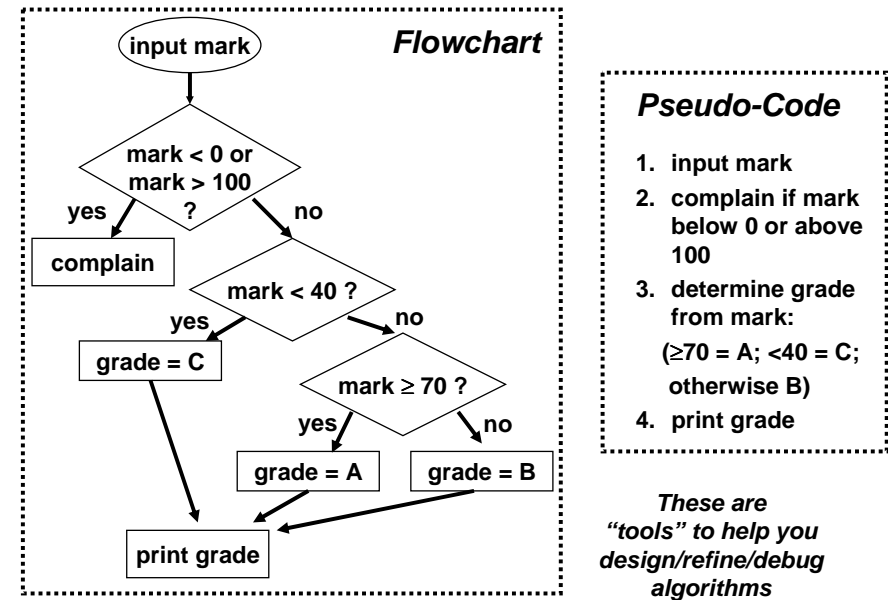
30

# Algorithm Development & Stepwise Refinement

## Algorithm Development

- In order for a computer to carry out some task, it has to be supplied with a **program**, which is an implementation of an **algorithm**. This is expressed in a computer programming language; however it is possible (and desirable) to talk and reason about algorithms in higher-level terms.
  - Developing a correct algorithm can be a significant intellectual challenge – by contrast, coding it should be straightforward (although coding it **well** may not be!)
- The most widely used notations for developing algorithms are **flowcharts** and **pseudo-code**. These are independent of the programming language to be used to implement the algorithm.
  - A flowchart is a diagram containing lines representing all the possible paths through the program.
  - Pseudo-code is a form of “stylised” (or “structured”) natural language.

1



2

## Algorithm Development (contd.)

- One of the problems encountered when writing programs is that of **preciseness**. A common fault among algorithms is that the process described is **almost** the intended one, but not quite.
  - Analogy: giving directions, following a recipe. These are rarely completely precise, but instead rely on the common sense of the person receiving the instructions. However computers are not equipped with common sense!
- Another common failing is that execution **usually** results in the intended process being carried out, but in certain circumstances (unforeseen or overlooked by the designer) it does not.
  - For example, consider the following algorithm to calculate the flight time of an aircraft using information from the timetable:  
**Look up departure time**  
**Look up arrival time**  
**Subtract departure time from arrival time**

This algorithm will usually give the correct result, but the subtraction should take into account the special case when the plane arrives on the day after departure. Also, what about:

Different time zones? Daylight savings time?

3

- Another required property of an algorithm is that each step can actually be carried out – in other words, the algorithm is **executable**. The point here is to make sure there are no “impossible” or unknown steps in your algorithm (e.g. algorithm relies on solving a sub-problem which is known to have no solution; algorithm asserts that a sub-problem will be solved without specifying how; etc).
- Most processes are supposed to terminate! There are examples of some which don't need to, but we assume that all programs we are interested in should.
- Thus the designer of an algorithm must ensure:
  - **Preciseness** of the algorithm (no ambiguities)
  - **All possible circumstances** are handled
  - The algorithm is **executable**
  - **Termination** of the algorithm
- Also have to worry about **Efficiency**
  - an algorithm may work correctly but be inefficient – by taking more time and using more resources than required to solve the problem.
  - becomes more important for larger programs.

4

## Stepwise Refinement

- Break a **complex problem** down into a number of **simpler steps**, each of which can be solved by an algorithm which is smaller and simpler than the one required to solve the overall problem.
  - Smaller and simpler, therefore easier to construct and sketch in detail
  - Sub-algorithms can themselves be broken into smaller portions
- **Refinement** of the algorithm continues in this manner until each step is sufficiently detailed.
  - Refinement means replacing existing steps/instructions with a new version that fills in more details.
- **Example: Making tea.** Suppose we have a robot which carries out household tasks. We wish to program the robot to make a cup of tea. An initial attempt at an algorithm might be:
  - 1. Put tea leaves in pot**
  - 2. Boil water**
  - 3. Add water to pot**
  - 4. Wait 5 minutes**
  - 5. Pour tea into cup**

5

- These steps are probably not detailed enough for the robot. We therefore refine each step into a sequence of smaller steps:

### 1. Put tea leaves in pot

might be refined to

#### 1.1 Open box of tea

#### 1.2 Extract one spoonful of tea leaves

#### 1.3 Tip spoonful into pot

#### 1.4 Close box of tea

Similarly:

### 2. Boil water

might be refined to

#### 2.1. Fill kettle with water

#### 2.2 Switch on kettle

#### 2.3 Wait until water is boiled

#### 2.4 Switch off kettle

### 5. Pour tea into cup

might be refined to

#### 5.1. Pour tea from pot into cup until cup is full

6

- Suppose the original 5 steps have been refined into sub-algorithms where necessary (e.g. it was not necessary to refine step 4 because it is simple enough for the robot to directly execute).
- Some of the sub-algorithms need further refinement. For example, the step
  - 2.1. Fill kettle with water** could be refined to
    - 2.1.1. Put kettle under tap**
    - 2.1.2. Turn on tap**
    - 2.1.3. Wait until kettle is full**
    - 2.1.4. Turn off tap**
- Others steps may also require further refinement. After a number of refinements the robot is able to execute every step.
- The program is then constructed by translating the final refinement of each step into C program statements.

7

### Original Algorithm

### First Refinement

### Second Refinement

#### 1. Put tea leaves in pot

#### 1.1 Open box of tea

- 1.1.1 Take tea box from shelf
- 1.1.2 Remove lid from box

#### 1.2 Extract one spoonful

#### 1.3 Tip spoonful into pot

#### 1.4 Close box of tea

- 1.4.1 Put lid on box
- 1.4.2 Replace tea box on shelf

#### 2. Boil Water

#### 2.1 Fill kettle with water

- 2.1.1 Put kettle under tap
- 2.1.2 Turn on tap
- 2.1.3 Wait until kettle is full
- 2.1.4 Turn off tap

#### 2.2 Switch on kettle

#### 2.3 Wait until water boiled

#### 2.4 Switch off kettle

#### 2.3.1 Wait until kettle whistles

#### 3. Add water to pot

#### 3.1 Pour water from kettle until pot is full

#### 4. Wait 5 Minutes

#### 5. Pour tea into cup

#### 5.1 Pour tea from pot into cup until cup is full

8

- When using stepwise refinement the designer must know **when to stop refining**. They must know when a particular step of the algorithm is sufficiently described to need no further refinement.
  - In this example the designer must know that the instruction **Switch on kettle** is directly executable by the robot, but that **Fill kettle with water** is not.
- In our case we are issuing instructions to a computer using a high-level programming language and hence **experience** will tell us when a step is directly implementable in that language or not.
- The above algorithm consists of a **sequence of steps**, each of which will be executed exactly once and in order – termination of the last step implies termination of the algorithm. However, algorithms with only sequences of steps can't do much...
  - Example: What happens if the tea-box is empty?

9

- If the tea-box is empty we wish to specify an extra step:
  - Get new box of tea from cupboard**
    - This step would not be carried out unless the tea-box is empty, and hence an algorithm incorporating this would not be entirely sequential anymore.
- We can express this by rewriting step 1.1 as
  - 1.1.1. Take tea box from shelf**
  - 1.1.2. If box is empty**
    - then get new box from cupboard**
  - 1.1.3. Remove lid from box**
    - Step 1.1.2 expresses both the step to be selected and the condition under which this selection should be made.
- More complicated conditions can use **AND, OR, NOT**

10

- Another common requirement is the need for **iteration**.
  - Example: suppose all we have access to is a timer which waits 1 second. Therefore we cannot simply say “wait 5 minutes”; this step must be refined further, e.g.

```

4.1. Set counter to 1
4.2. WHILE counter<=300 DO
    wait
    increase counter by 1
  
```

- Although the above iteration is expressed using “while”, this does not mean that the program must use a **while** loop. Any of the loop structures supported in C can be used. Similarly, the loop in the program does not have to use a loop counter which counts from 1 up to 300, e.g. you may prefer to count down to 0...

11

- In most programming languages, you can/should explicitly structure your program into “modules” that each do some specific task. Why? Makes programs easier to build, understand, debug, maintain; and encourages re-use of modules in different programs.
- In C, this can be done with a **function**: a piece of code which is given a name so that it can be invoked by other parts of the program

```

#include "stdio.h"
char convert_mark(int mark) {
    if (mark < 40) return 'C';
    else if (mark >= 70) return 'A';
    else return 'B';
}
void main(void) {
    int mark;
    char grade;
    printf("enter the mark: ");
    scanf("%d", &mark);
    grade = convert_mark(mark);
    printf("You got a %c\n", grade);
}
  
```

*If refine problem well, each function is fairly short, & its job is clear. (Soon, we'll discuss details of how to write functions; for now the point is just WHY)*

- Usually, the structure of the refinement process is reflected in the structure of the function invocations...

12

## Sources of errors

- Many errors made in analysing the problem, developing an algorithm, and/or coding the algorithm, only become apparent when you try to compile, run, and test the resulting program.
- The earlier in the development process an error is made, or the later it is discovered, the more serious the consequences.

### Sources of errors:

- **Understanding the problem to solve.** An error here may be obvious e.g. your program does nothing useful at all. Or it may be more subtle, and only become apparent when some exceptional condition occurs e.g. a leap year, incompetent user, ...
- **Algorithm design.** Mistakes here result in **logic errors**. The program will run, but will not perform the intended task e.g. a program to add numbers which returns 6 when given 3+2 has a logic error.
- **Coding of the algorithm.** Often the compiler will complain, but messages from the compiler can be cryptic. These errors are usually simple to correct e.g. spelling errors, misplaced punctuation, ...
- **Runtime.** Errors may appear at run time e.g. divide some number by zero. These errors may be coding errors or logic errors.

13

## Sources of errors (contd.)

- Programs rarely run correctly the first time (!)
- Errors are of three types:
  - syntax errors
  - run-time errors
  - logic errors
- **Syntax errors:** detected by the C compiler
  - source code does not conform to one or more of C's grammar rules
  - examples of syntax errors:
    - undeclared variable
    - missing semicolon at end of statement
    - comment not closed
- Often one mistake leads to multiple error messages – can be confusing!

14

## Sources of errors (contd.)

- **Run-time errors:**
  - detected and displayed by computer during execution
  - occur when program directs computer to perform illegal operation  
Example: `int x=y/0;`
  - will stop program execution and display message
- **Logic errors:**
  - caused by faulty algorithm
  - do not usually cause run-time errors
  - Includes errors that do not prevent execution of program  
Example: `float f; scanf("%d",&f);`
  - difficult to detect for large and complex algorithms
  - sign of error: incorrect program output
  - cure: thorough testing and comparison with expected results

15

## Sources of errors & how to reduce them

- Errors made in understanding the problem may well require you to restart from scratch.
  - You may be tempted to start coding, making up the solution and algorithm as you go along. This may work for trivial problems, but is a certain way to waste time and effort for realistic problems.
- A program can be tested by giving it inputs and comparing the observed outputs to the expected ones.
  - Testing is very important but (in general) cannot prove that a program works correctly.
  - For small programs, formal (mathematical) methods can be used to prove that a program will produce the desired outputs for all possible inputs. These methods are rarely applicable to large programs.
- A logic error is referred to as a **bug**, so finding logic errors is called **debugging**.
  - Debugging is a continuous process, leading to an **edit-compile-debug** cycle. General idea: insert extra `printf()` statements showing the values of variables affected by each major step. When you're satisfied program is correct, comment out these `printf()`'s.

16

## Debugging tips – common errors in C

- In a `for` loop, initialisation and condition end with semicolons.  
Wrong – `for(initialisation, condition; update)`
- Must use braces in `for` and `while` loops to repeat more than one statement.
- nested structure: first closing brace is associated with innermost structure.  
Example of “unexpected” behaviour:  

```
if(c=='y') {
    scanf("%d", &d);
    while(d!=0) {
        sum+=d;
        scanf("%d", &d);
    } else printf("end"); /* done even when c=='y' */
```
- inequality test on floating point numbers. Example of wrong way:  

```
while(value!=0.0) { /* could have value<1e-9 */
    ...
    value/=2.8; /* now value will be regarded as 0 */
}
```

17

## Debugging tips – common errors in C (contd.)

- Should ensure that loop repetition condition will eventually become **false**.  
Example where this doesn't happen:  

```
do {
    ...
    printf("one more time?");
    scanf("%d", &again);
} while(again=1); /* assignment, not equality test */
```
- loop count off by one, either too many or too few iterations. Or infinite loop, or loop body never executed. Can be hard to discover!

18

## Functions in C

The concept of a function is a fundamental idea in mathematics. For example, the mathematical notation

$$f(x) = x^2$$

is used to describe the simple mathematical function of squaring a value. In this example  $f$  is the name of the function, and the variable  $x$  that appears inside the parentheses is called an **argument** or **parameter** of the function.

Mathematically:  $x=2, f(x)=4$ ;  $x=9, f(x)=81$ ; and so on.

Functions take in arguments and pass back a **return value** (here: 4, 81).

*The return value of a function is determined by its definition and the values of its parameters.*

C functions are like mathematical functions in that **they can accept arguments** and **they can return values**.

Functions can be either defined by the programmer or pre-defined as part of a system library. First, we look at examples from the C system libraries. Then: how to define your own functions.

1

## Programming with Libraries

- No-one wants to reinvent the wheel. Examples: compute square root, capitalize a word, fetch a URL...
  - **How to re-use bits of software** – a major challenge for computer science and software engineering.
- These software “components” are stored in a **software library** from which they can be retrieved and incorporated into new programs as needed.
- A library has two parts:
  1. An **interface**, which specifies which items are stored in the library and how they are used. This is stored in a **header file** with filename extension “.h”

```
#include <math.h>
```
  2. An **implementation**, which contains definitions of the items in the library (i.e. the code). The library user doesn’t need to see this!
- C provides a large set of libraries for the use of the programmer: **ctype.h**, **string.h**, **stdlib.h**, **stdio.h**, **math.h**, **stdarg.h**, **setjmp.h**, **signal.h**, **assert.h**, **time.h**, **locale.h**...

2

## Examples of Library Functions

- The **math.h** library contains many functions relating to mathematical operations – if you use these, they can save you a lot of work.
- Recall the program to calculate the volume of a sphere. The formula is:

$$v = \frac{4}{3} \pi r^3$$

- The relevant line of code to carry out this calculation looked like

```
float volume = 4.0/3*PI*radius*radius*radius;
```
- Another example could be

```
float energy = mass*SPEEDOFLIGHT*SPEEDOFLIGHT;
```
- Need a general-purpose “raise to a power” function...

3

## Examples of Library Functions (contd.)

- Use the **pow** function from the **math.h** library:

$$pow(x, y) = x^y$$

- $x$  and  $y$  are its arguments and it returns  $x$  raised to the power of  $y$ .
- So we can say:

```
float volume = 4.0/3*PI*pow(radius,3);  
float energy = mass*pow(SPEEDOFLIGHT,2);
```
- In order to use the **pow** function we need (at the start of the program)

```
#include "math.h"
```
- Important: you don’t need to know/care how **pow** actually does its job -- you can just use it. (Of course, it’s just a piece of C code...)

4

## Examples of Library Functions (contd.)

- There are many other functions in `math.h`, e.g.

```
#include "math.h"
. . .
int num = 100;
float result = sqrt(num);
```

$$f(x) = \sqrt{x}$$

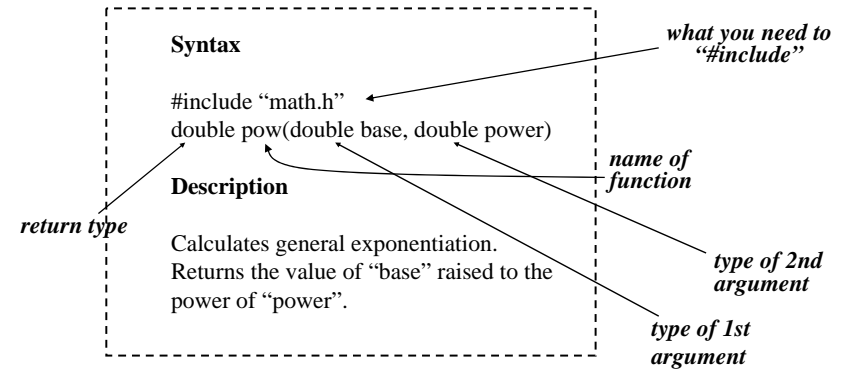
- Other examples:

<code>ceil(x)</code>	smallest integer not less than <b>x</b>
<code>floor(x)</code>	largest integer not greater than <b>x</b>
<code>fabs(x)</code>	absolute value of <b>x</b>
<code>sin(x)</code>	trig. sine of <b>x</b> , where <b>x</b> is in radians
<code>acos(x)</code>	trig. arc cosine of <b>x</b> , expressed in radians in $(0, \pi)$
<code>exp(x)</code>	exponential of <b>x</b> (i.e. <code>pow(e, x)</code> , $e=2.71828\dots$ )
<code>log(x)</code>	logarithm of <b>x</b> (base e)
<code>log10(x)</code>	logarithm of <b>x</b> (base 10)

5

## How to use library functions

- When using functions it is important to know what parameters they expect (if any) and what value they return (if any).
- Specifically, you must know the **number, order, and type of the parameters**, and the **type of the return value**.
- To find this out, use your programming environment's "help" facility:



6

## Actual and Formal Parameters

- The variables/values used in the function call are called **actual** parameters
- The variables/values used in the function itself are called **formal** parameters
- Actual variable names used are irrelevant – their role is determined solely by their **position in the list**. For example:

```
double power = 3;
double base = 10;
double res = pow(power, base); /* calculates 310, not 103 */
```

- If formal and actual parameters have different types, the values are converted:

```
float base = 3;
float power = 10;
float res = pow(base, power);
```

when executed, computer automatically assigns to formal parameters the **doubles** to which **base** and **power** correspond

then, the return value **double** is automatically converted back to a **float**

7

## Programming with Libraries (contd.)

- not just for mathematical functions! For example:
  - manipulating date and time values
  - text string processing
  - accessing files and low-level system resources
  - etc.
- But: sometimes you have to write your own functions, because (e.g.)
  - precise functionality you need isn't in standard library
  - your functions should reflect your algorithm's refinement structure
- key ideas: a function you write has to be
  - defined** (specifies what exactly the function does)
  - declared** (so that it is accessible to other functions that use it)
  - called** (when another function needs the processing provided by your function)

8

A simple example: write a function to compute integer powers: result = base<sup>exponent</sup>

```
#include <stdio.h>

int power(int base, int exponent) {
    int result = 1, i;
    for (i = 0; i < exponent; i++) {
        result = result*base;
    }
    return result;
}

void main(void) {
    int answer = power(3,4) * power(10,2);
    printf("the answer is %d\n", answer);
}
```

produces the screen output: the answer is 8100

A simple example: write a function to compute integer powers: result = base<sup>exponent</sup>

**function definition:**

```
type of value it returns  name of function
int power(int base, int exponent) {
    int result = 1, i;
    for (i = 0; i < exponent; i++) {
        result = result*base;
    }
    return result;
}
```

body of function (whatever is appropriate)

"invoke" or "call" the function

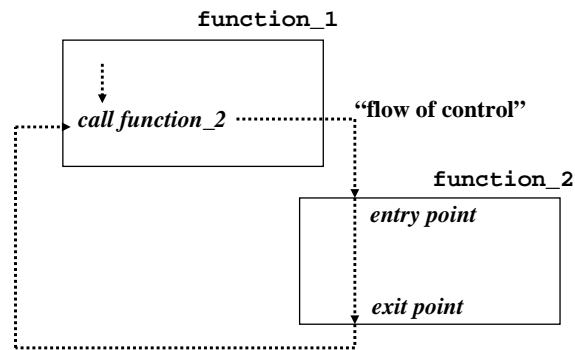
return expression; Must occur somewhere in your function if you want it to return a value. Tells the compiler what to return. (Here, we have a variable called result and expression is just the value of that variable)

```
void main(void) {
    int answer = power(3,4) * power(10,2);
    printf("the answer is %d\n", answer);
}
```

use the call as a value -- here we multiply the results of two separate calls

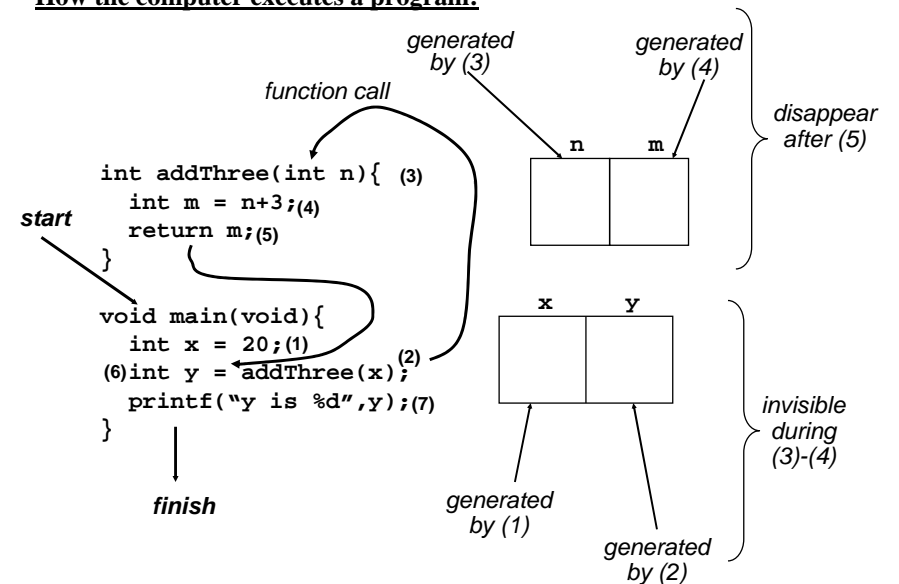
## Function call mechanism

When one function calls another function in C, the calling function is "suspended" until the called function "returns control" to it:



Note: calling function **may or may not** send data to called function; called function **may or may not** return data to calling function when it returns.

## How the computer executes a program:



## General “template” for function definition

```
return-type function-name (parameter-list) {  
    declarations & statements (as usual),  
    optionally with return somewhere in here  
}
```

If the function returns an integer, use return-type **int**

If the function returns a double (e.g. **pow**), use return-type **double**  
etc.

**Not all functions return a value** (e.g. **main**) -- use special type **void**

If no return-type specified: default is **int** (not **void**).

Like variable names, the **function-name** assigns a unique name for the function.

You should name a function according to its purpose (i.e. meaningfully).

The **parameter-list** is used to specify what arguments the function expects.

The general format is:

```
type variable, type variable, type variable, ...
```

If function doesn't take any arguments: use (**void**) or ()

13

## General “template” for function definition (contd.)

In C, function definitions CANNOT be “nested” – each function must be defined outside any other function definition.

If the calling function wants to “capture” a returned value from a called function, possible ways include **assignment** and **using it in an expression**:

```
int y = addThree(x);  
int answer = power(3,4) * power(10,2);
```

A returned value can also be **ignored** by the calling function. For example, so far we haven't been concerned with the return value from a call to **printf()**, although **printf()** does provide one – we've just discarded it every time.

In a function definition, when the closing **}** is reached, control is returned to the point in the calling function where this function was called. This is referred to as an **implicit return**. The **return** statement, on the other hand, is an **explicit return** since its effect is to **immediately** return control to the calling function. If the **return** statement has an associated **expression**, the value of that **expression** is also returned to the calling function as the “returned value”.

**Note:**                    **return expression;**  
is equivalent to       **return (expression);**

14

## Multiple return statements

Functions need not have just one return statement.

If function returns something other than **void**, then every possible “execution path” through the function body should end up at a **return** statement:

```
#include "stdio.h"  
char convert_mark(int mark) {  
    if (mark < 40) {return 'C';}  
    else if (mark >= 70) {return 'A';}  
    else {return 'B';}  
}  
  
void main(void) {  
    int mark;  
    char grade;  
    printf("enter the mark: ");  
    scanf("%d", &mark);  
    grade = convert_mark(mark);  
    printf("You got a %c\n", grade);  
}
```

[ Q: what if some “execution path” doesn't end up at a **return** statement? ]

15

## More on return statement

A function which does not return a value can still have return statement(s):

```
#include <stdio.h>  
void printer(int a, int b){  
    if (a == b) {return;}  
    else if (a > b) {printf("yes\n"); return;}  
    else {printf("no\n");} /* no explicit return */  
} /* implicit return for printer() */  
  
void main(void){  
    int x,y;  
    printf("enter value for x: ");  
    scanf("%d", &x);  
    printf("enter value for y: ");  
    scanf("%d", &y);  
    printer(x,y);  
}
```

16

## More on return statement (contd.)

The type of the expression returned must match the called function's return value, or be capable of being converted to it using C's datatype hierarchy (as seen before):

```
#include <stdio.h>
float function1 (void) {
    return 1; /* returned as 1.000000 */
}
int function2 (void) {
    return 3.1416; /* returned as 3 */
}
void main(void) {
    int x;
    float y;
    y = function1(); /* must include the () */
    printf("value of y is %.2f\n", y);
    x = function2(); /* must include the () */
    printf("value of x is %d\n", x);
}
```

Produces the screen output: value of y is 1.00  
value of x is 3

17

## Function parameters

A **function call** specifies the *actual parameters* to be used when the function is executed. These actual parameters are evaluated and used one-by-one as initial values for the function's formal parameters for this execution of the function.

An actual parameter in a function call can be:

- a constant
- a variable
- an expression
- the return value from another function

In all these cases, the actual parameter is evaluated and this value is copied into the corresponding formal parameter of the function being called.

Very very very important – the **value** of the actual parameter is *unaffected* by the execution of the called function. The value of the corresponding formal parameter is NOT copied back to the actual parameter when the called function returns (even if they have the same name!).

18

## Function parameters – example

```
#include <stdio.h>
void fn_1(int q){ /* this q is a formal param. for fn_1 */
    q = 100;
    printf("fn_1 just set parameter to %d\n", q);
}
void main(void){
    int q = 3;
    printf("value of parameter is %d\n", q);
    fn_1 (q); /* this q is the actual parameter here */
    printf("now the value of parameter is %d\n", q);
}
```

Produces the screen output:

value of parameter is 3  
fn\_1 just set parameter to 100  
now the value of parameter is 3

19

## Function parameters (contd.)

If a function has more than 1 parameter, the formal and actual parameters must match in **number, type, and order**. If not, in general you have problems...

If the types don't match, the value of the actual parameter(s) is converted according to C's datatype hierarchy (as seen before).

```
#include <stdio.h>
int max (int a, int b){
    if (a >= b) {return a;}
    else {return b;}
}
void main(void){
    int x=6, y=3, result1, result2;
    float f1=2.8, f2=4.7;
    result1 = max(x,y);
    printf("result of comparing the integers is %d\n", result1);
    result2 = max(f1,f2);
    printf("result of comparing the floats is %d\n", result2);
}
```

When max() is called with actual parameters x and y, the initial value of a is 6 and the initial value of b is 3

When max() is called with actual parameters f1 and f2, the initial value of a is 2 and the initial value of b is 4 (due to integer truncation)

Produces the screen output: result of comparing the integers is 6  
result of comparing the floats is 4

20

## Function prototypes

(another name for “function declarations”)

If you look back at all the programs we’ve examined in the previous 12 slides, you’ll notice that in every case, the functions (other than `main()`) were defined *before* the definition of `main()`. What if this were not the case (since in C, function definitions can occur in **any order** in a program file)? For example:

```
#include <stdio.h>
void main(void) {
    float y;
    y = function1(); /* first mention of function1() */
    printf("value of y is %.2f\n", y);
}
float function1 (void) {
    return 1; /* returned as 1.000000 */
}
```

Note: in C, the default return-type for a function is `int`. So when the C compiler first encounters a mention of `function1()`, it assumes that its definition is of the form `int function1(void){...}` – (this is called an “implicit declaration” of `function1()`). *So what happens when you compile/run this program?*

21

## Function prototypes (contd.)

```
#include <stdio.h>
void main(void) {
    float y;
    y = function1(); /* first mention of function1() */
    printf("value of y is %.2f\n", y);
}
float function1 (void) {
    return 1; /* returned as 1.000000 */
}
```

My C compiler issues the following warning:

```
At top level:
Line 7: warning: type mismatch with previous implicit declaration
Line 4: warning: previous implicit declaration of `function1'
Line 7: warning: `function1' was previously implicitly declared
to return `int'
```

But it does compile the program. However, the screen output is not as expected:

```
value of y is 0.00
```

Something has gone wrong! The solution – function prototype for `function1()` 22

## Function prototypes (contd.)

Each C function should be declared *before* it is called:

```
#include <stdio.h>
void main(void) {
    float y;
    float function1(void); /* prototype for function1() */
    y = function1();
    printf("value of y is %.2f\n", y);
}
float function1 (void) {
    return 1; /* returned as 1.000000 */
}
```

Now the C compiler knows what the definition for `function1()` looks like, before it has encountered it. Correct screen output produced: `value of y is 1.00`

A function prototype declares what **type of value the function returns**, and what **type(s) of parameters it accepts**. If no return value or parameters – use `void`

Optionally, a function prototype may use parameter names in the argument list, e.g.

```
void printer(int x, int y); /* function prototype */
```

As with the formal parameters in a function definition, these names are for documentation only and do not have to match the names of the actual parameters in the function call.

23

## Function prototypes: another example

```
#include <stdio.h>

void main(void) {
    int power(int a, int b);
    /* function prototype with "dummy" parameters a and b */
    int answer = power(3,4) * power(10,2);
    printf("the answer is %d\n", answer);
}

int power(int base, int exponent) {
    int result = 1, i;
    for (i = 0; i < exponent; i++) {
        result = result*base;
    }
    return result;
}
```

Produces the correct screen output: `the answer is 8100`

24

## Storage class and Scope

Key question: what variables can a particular function access?

Recall example program from slide 12: if we try EITHER of the following changes –

```
#include <stdio.h>
int addThree(int n){
    int m = x+3; /* ILLEGAL! Won't compile */
    return m;
}

void main(void){
    int x = 20;
    int y = addThree(x);
    printf("y is %d",m); /* ILLEGAL! Won't compile */
}
```

The general rule: A function has access only to variables declared in its function body (*local variables*) and variables in its parameter list.

25

## Storage class and Scope (contd.)

A variable's *scope* refers to the region of a program in which the variable is accessible. Variables defined within a function (including its formal parameters) are **local** to that function, and their scope is that function's body.

One consequence of this is that different functions can have local variables with the same name – such variables are unrelated, and any operation performed on one such variable has *no effect* on other variables of the same name in other functions. See slide 19 for an example of this!

It is possible to define a variable outside the body of any function – such a variable is called a **global variable**, and its scope is from the point of declaration of the variable to the end of the program file.

Since C functions are defined outside of any other function, they are treated as “global identifiers” and are accessible to other functions that come after them in the program file. To make them accessible to all functions in the program – use a function prototype.

26

## Storage class and Scope (contd.)

```
#include <stdio.h>
int count=0; /* global variable */
void main(void) {
    int x, y; /* x and y are local to main() */
    void function1(int, int); /* function prototype */
    x = count + 2;
    printf("value of x in main is %d\n", x);
    count++; /* increase global variable "count" by 1 */
    y = count;
    printf("value of y in main is %d\n", y);
    function1(x,y);
    y = count;
    printf("now value of y in main is %d\n", y);
}
void function1(int a, int b) { /* a and b are local to function1() */
    int x=4; /* this x is a local variable in function1() */
    printf("value of x in function1 is %d\n", x);
    printf("value of a in function1 is %d\n", a);
    printf("value of b in function1 is %d\n", b);
    count--; /* decrease global variable "count" by 1 */
}
```

Produces the screen output:

```
value of x in main is 2
value of y in main is 1
value of x in function1 is 4
value of a in function1 is 2
value of b in function1 is 1
now value of y in main is 0
```

27

## Storage class and Scope (contd.)

*Storage class* of a variable refers to how long their value is held by the computer.

- Local variables have **automatic** storage class – their value is lost when the function in which they are defined exits.
- Global variables have **external** storage class – their value is held until the program terminates.
- Since C functions are defined outside any other function, they have **external** storage class.

Sometimes you may want a local variable to hold its value until the program terminates. This will be the case if the variable's declaration is preceded by the keyword **static**. Local **static** variables hold their value across multiple function calls; any initialisation is done only the first time the function is executed.

Local **static** variables are often used to count how many times the function in which they are defined has been executed during the lifetime of the program...

28

## Storage class and Scope (contd.)

```
#include <stdio.h>
void main(void) {
    int i;
    void fn(void); /* function prototype */
    for (i=0; i<5; i++) {
        fn(); /* call to function fn() */
    }
}
void fn(void) {
    static int var_s = 0; /* done once */
    int var_a = 0; /* done every time fn() is entered, */
                    /* since var_a has automatic storage */
    printf("var_a = %d, var_s = %d\n", var_a, var_s);
    var_a++; /* increase var_a by 1 */
    var_s++; /* increase var_s by 1 */
}
```

Produces the screen output:

```
var_a = 0, var_s = 0
var_a = 1, var_s = 1
var_a = 2, var_s = 2
var_a = 3, var_s = 3
var_a = 4, var_s = 4
```

29

## A problem on functions

What will be output when the following code fragment is executed ?

```
#include "stdio.h"
int f1(int, int); /* function prototype for f1() */
int f1(int i, int j){
    return (i+j);
}
int f2(int i){
    int j = f1(i, i-1);
    return j;
}
void main(void){
    int f2(int); /* function prototype for f2() */
    printf("result of function call is %d\n", f1(4,f2(4)) );
}
```

30

## Introduction to Arrays

An array is a data structure used to store a collection of data items all of the same type.

The name of the array is associated with the collection of data. To access an individual data item, you need to indicate to the computer which array element you want. This is indicated using an array **index** (or **subscript**).

Why are arrays useful? Suppose you want to write a program which accepts 5 integers input by the user, and prints them out in reverse order. You could do it like this:

```
int first, second, third, fourth, fifth;
printf("enter 5 integers, separated by spaces: ");
scanf("%d %d %d %d %d", &first, &second, &third, &fourth, &fifth);
printf("in reverse order: %d, %d, %d, ", fifth, fourth, third);
printf("%d, %d\n", second, first); /* output is all on 1 line */
```

This works as required. But – what if you had 50 inputs? Or 500?! Or... Using integer variables would become very cumbersome...

1

## Introduction to Arrays (contd.)


A **variable** holds a **single** value, e.g.

```
int mark;          mark 
```

An **array** holds **several** values, all of the same type, e.g.

```
int marks[5];

marks[0]----->
marks[1]----->
marks[2]----->
marks[3]----->
marks[4]----->
```



General array declaration:

**type arrayname[size]**  
where **size** is an **integer-valued**

- **constant**, or
- **variable**, or
- **expression**

- The computer provides a **contiguous** block of memory for storing the array.
- All array elements have the same type (**int**, **float**, **char**, ...)
- Many algorithms require data to be collected and stored, and later processed in some way(s). If data is **all of the same type**, an array is the natural way to do this.
- Can access values using **marks[3]**, **marks[i]**, **marks[j\*k]**, etc. (index must be integer-valued). An array element can then be treated just like a variable.
- **First** element has index **0** (*not 1*). **Last** element has index **size-1** (*not size*).

2

## Introduction to Arrays (contd.)

To solve the reversed-inputs problem using an array:

```
int i; /* loop counter */
int table[5];
/* declares an array "table" with storage for 5 integers */
for (i=0; i<=4; i++) {
    printf("enter integer %d: ", i+1);
    /* "i+1" so user sees count starting at 1 */
    scanf("%d", &table[i]);
}
printf("in reverse order: ");
for (i=4; i>=0; i--) {
    printf("%d ", table[i]); /* last element is table[4], etc */
}

```

Produces the screen output (*user inputs in italics*):

```
enter integer 1: 3
enter integer 2: 4
enter integer 3: 5
enter integer 4: 6
enter integer 5: 7
in reverse order: 7 6 5 4 3
```

3

## More details on arrays

- “Scalar” variables hold a single value: **float temperature**;
- “Array” variables hold a collection of values: **float temperatures[NCITIES]**;
- When an integer variable is declared a sufficient amount of memory (e.g. 2 bytes) is reserved for it.
- Similarly when an array is declared a sufficient amount of memory is reserved. If each **float** value requires 8 bytes, an array of 100 **floats** takes up 800 bytes.
- The compiler takes care of these details for you -- but it’s important you know them.

```
float t[10];

1 float 1 float 1 float 1 float 1 float 1 float 1 float 1 float
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|       |       |       |       |       |       |       |       |       |       |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
t[0]  t[1]  t[2]  t[3]  t[4]  t[5]  t[6]  t[7]  t[8]  t[9]
```

**example memory locations**

```
1258-1265 1266-73 1274-81 1282-1289 1290-1297 1298-1305 1306-1313 1314-1321 1322-1329 1330-1337
```

- **C does not check or enforce array bounds** – it’s up to you not to access memory locations that are not part of the array.
  - Get junk/error if you try to access **t[10]**, **t[-1]**, etc.

4

## Array initialization: examples

```
#include "stdio.h"
void main(void){
    int i;
    int somearray[5] = {5,6,7,8,9}; /* initialisation statement */
    for (i = 0; i <= 4; i++){
        printf("element number %d is %d\n", i, somearray[i]);
    }
}
```

Produces the screen output:

```
element number 0 is 5
element number 1 is 6
element number 2 is 7
element number 3 is 8
element number 4 is 9
```

*What if initialisation statement is changed to:*

```
int j=5, k=9; int somearray[5] = {j,j+1,j+2,k-1,k};

int somearray[5]={5,6,7};

int somearray[] = {5,6,7,8,9};

int somearray[5];
```

5

## A simple use of arrays

```
#include <stdio.h>
#define NSTUDENTS 5
#define BONUS 10
void main(void) {
    int i, marks[NSTUDENTS], total=0;
    float average;
    for (i = 0; i < NSTUDENTS; i++) {
        printf("Enter mark for student %d: ", i);
        scanf("%d", &marks[i]); /* read in all the marks */
    }
    for (i = 0; i < NSTUDENTS; i++) {
        marks[i] = marks[i] + BONUS; /* everyone gets a bonus */
        total += marks[i]; /* get sum of all marks */
    }
    average = total / (float) NSTUDENTS; /* cast to float */
    for (i = 0; i < NSTUDENTS; i++) {
        printf("Student %d's mark deviates from average", i);
        printf(" by %.2f\n", marks[i]-average);
    }
}
```

- note **for** loop – very useful for dealing with arrays
  - how would you write this with **while** loops instead of **for**?
- note that **arrayname[index]** can be treated just like a regular variable
  - in this case, **marks[i]** treated just like a variable of type **int**

6

## A simple use of arrays – running the program

```
Enter mark for student 0: 40
Enter mark for student 1: 35
Enter mark for student 2: 61
Enter mark for student 3: 77
Enter mark for student 4: 28
Student 0's mark deviates from average by -8.20
Student 1's mark deviates from average by -13.20
Student 2's mark deviates from average by 12.80
Student 3's mark deviates from average by 28.80
Student 4's mark deviates from average by -20.20
```

(with the above marks, the adjusted marks – when the bonus 10 marks are given to all students – are 50, 45, 71, 87, and 38. The average of these adjusted marks is 58.2, so the students' deviations from this average are -8.2, -13.2, 12.8, 28.8, and -20.2, which agrees with the above program output)

7

## Yet Another Example

A program to read in an array of 5 exam marks and calculate the highest mark:

```
#include <stdio.h>
void main(void) {
    int index, marks[5];
    int high_index = 0;
    for (index = 0; index < 5; index++) {
        printf("enter mark no. %d: ", index);
        scanf("%d", &marks[index]);
    }
    for (index = 1; index < 5; index++) { /* NOTE 1 */
        if (marks[index] > marks[high_index]) {
            high_index = index;
        }
    }
    printf("The highest mark was %d, ", marks[high_index]);
    printf("obtained by student %d\n", high_index);
}
```

Things for you to think about:

NOTE 1: why **index = 1** and not **index = 0** ?

Also: what happens here if 2 or more students get the same "highest" mark?

8

## Yet Another Example – running the program

```
enter mark no. 0: 70
enter mark no. 1: 55
enter mark no. 2: 71
enter mark no. 3: 29
enter mark no. 4: 44
The highest mark was 71, obtained by student 2
```

```
enter mark no. 0: 60
enter mark no. 1: 60
enter mark no. 2: 60
enter mark no. 3: 60
enter mark no. 4: 60
The highest mark was 60, obtained by student 0
```

9

## Linear Search

**A common task:** you have an array, and you need to find a specific element.

**Technique:** “search” -- systematically examine the array to find the element

**Simplest kind of search:** linear search

**Example:** two arrays: `patientIDs[]` and `ages[]`

```
int patientIDs[5000];
int ages[5000];
/* fill up these arrays somehow (read from file, keyboard,...) */
```

**Question:** Is there a 100-year-old patient?

```
int i;
for (i = 0; i < 5000; i++) {
    if (ages[i] == 100) {
        printf("yes\n");
    }
}
```

10

## Linear Search (contd.)

**Problems with previous program:** never print “no”; don’t know which patient is 100, if there is one; check every element even after we know the answer!

**Solution:** use a `while` loop, stop loop early if find what you’re looking for; then test value of counter `i` after program execution

```
int i = 0;
while ((i < 5000) && (ages[i] != 100)) {
    i++; /* keep going while unsuccessful */
}
if (i == 5000) { /* did we search the whole array? */
    printf("no hundred-year-old patient\n");
} else { /* stopped early, therefore success */
    printf("patient number %d", i);
    printf(" (ID %d) is 100\n", patientIDs[i]);
}
```

11

## Linear Search: another example

### Find the age of the patient whose ID is 347822

```
int targetPatientID, targetAge, i;
printf("What patient ID are you looking for: ");
scanf("%d", &targetPatientID);
for (i = 0; i < 5000; i++) {
    if (patientIDs[i] == targetPatientID) {
        targetAge = ages[i]; /* found target patient */
        i = 5000; /* force loop to exit early */
    }
} /* if target not found, loop runs to completion */
printf("Patient %d's age is %d\n", targetPatientID, targetAge);
```

*MORAL of the story: the loop control variables can be used to do something besides merely control the number of iterations, e.g. to control the number of iterations in a “special” way.*

*Problem with the above program: if target patient not found, print out result (with rubbish value for `targetAge`) anyway. How would you fix this?*

12

## Passing 1D arrays to functions

```

/* a general-purpose averaging function */
float average(float array[], int size) {
    int i;
    float sum = 0;
    for (i = 0; i < size; i++) {
        sum += array[i];
    }
    return sum/size;
}

void main(void) {
    float temperatures[20];
    /* somehow fill in temperatures array */
    float marks[50];
    /* somehow fill in marks array */
    printf("avg. temp.=%.2f", average(temperatures, 20));
    printf("avg. mark=%.2f", average(marks, 50));
}

```

*leave size  
unspecified  
in formal  
parameter; just  
give array  
name as actual  
parameter...*

*... but **can** also  
pass in size  
as separate  
argument, to  
make sure the  
function only  
accesses valid  
indices*

**Q: under what assumption could you omit the size of the array as a parameter of the called function?**

13

## More Examples -- Passing 1D arrays to functions

```

/* returns sum of elements A[0] ... A[size-1] */
int sum(int A[], int size) {
    int i, tot = 0;
    for (i=0; i<size; i++) {tot += A[i];}
    return tot;
}

/* returns the maximum value in A */
int max(int A[], int size) {
    int m = A[0], i;
    for (i=1; i<size; i++) {if (A[i]>m) {m=A[i];}}
    return m;
}

/* count the number of times a given "target" occurs in A */
int count(int A[], int size, int target) {
    int n = 0, i;
    for (i=0; i<size; i++) {if (A[i]==target) {n++;}}
    return n;
}

/* return index of the first occurrence in A, or -1 if not found */
int find(int A[], int size, int target) {
    int i;
    for (i=0; i<size; i++) {if (A[i]==target) {return i;}}
    return -1; /* target not found, since for loop completed */
}

```

14

## Arrays & function calls

By default, "simple" data types (int, char, double, ...) are passed by value.

In contrast, arrays are passed by address. For example:

```

#include <stdio.h>
void zap(int array1[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        array1[i] = 0; /* sets all elements to 0 */
    }
}
void main(void) {
    int i, charges[5]={7,8,7,6,7};
    printf("elements of charges[] are ");
    for (i=0;i<5;i++){printf("%d ", charges[i]);}
    zap(charges, 5); /* function call will affect charges */
    printf("\nnow elements of charges[] are ");
    for (i=0;i<5;i++){printf("%d ", charges[i]);}
}

```

*array1[] in zap()  
is another name for  
charges[] in main()*

Produces the screen output: elements of charges[] are 7 8 7 6 7  
now elements of charges[] are 0 0 0 0 0

15

## Introduction to Pointers

- **Pointer** - *a whole new kind of variable*
  - **scalar** variable: **single** value
  - **array** variable: **several** values (of same type)
  - **pointer** variable: **memory location** of a value
- pointers are incredibly *flexible/useful*
- unfortunately, pointers are also a bit *complicated* – takes a while to see their uses and benefits

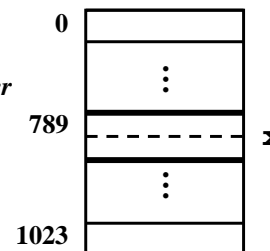
1

## Names and Addresses

- every variable has a location in memory. This memory location is uniquely determined by a **memory address**.
- use the **&** operator to find out the address of a variable – as already used in **scanf** statements
  - e.g. `scanf("%d", &x);`

*x* is the name the computer uses to refer to the value stored in the memory location (here, the integer entered by the user);

**&x** is 789 here, and refers to the address of the memory location



Suppose:

Memory size = 1024 bytes (obviously unrealistic!); Integer stored in 2 bytes.

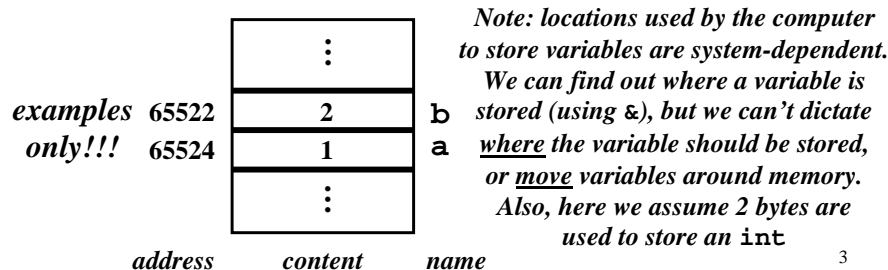
Note: address of memory location is the address of 1<sup>st</sup> byte of the location.

2

## Names and Addresses: example

```
#include "stdio.h"
void main(void){
    int a=1, b=2;
    printf("a = %d; address of a: %u\n", a, &a);
    printf("b = %d; address of b: %u\n", b, &b);
}
```

Produces the screen output: a = 1; address of a: 65524  
b = 2; address of b: 65522

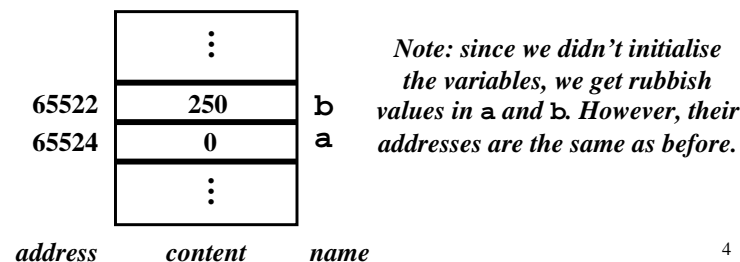


3

## Names and Addresses: another example

```
#include "stdio.h"
void main(void){
    int a, b; /* a, b not initialised here */
    printf("a = %d; address of a: %u\n", a, &a);
    printf("b = %d; address of b: %u\n", b, &b);
}
```

Produces the screen output: a = 0; address of a: 65524  
b = 250; address of b: 65522



4

## Pointers

- **A pointer is a datatype which stores addresses.**
  - Compare to: “an int is a datatype which stores integers”
- A particular pointer – say, **ptr** – is a variable which stores an address. We say that **ptr** “points to” the object whose address is stored in **ptr**
  - Every pointer has an associated datatype and is only allowed to store addresses of objects of that type.
    - therefore we have a *pointer-to-int*, a *pointer-to-float*, a *pointer-to-char*, etc.
  - Because a pointer is a variable:
    1. the address stored in the pointer can be changed;
    2. the pointer itself must be stored in some memory location, which must have an address. Some other pointer could therefore “point to” **ptr**

5

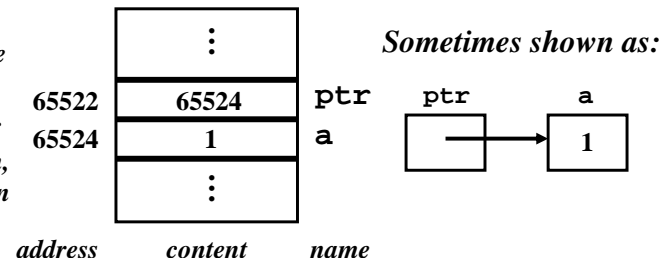
## Pointers (contd.)

Declaring and initialising a pointer variable:

```
int a = 1;
int *ptr = &a; /* declare and initialise pointer */
```

*Note: it is good practice to initialise a pointer variable when it is declared.*

*Also, by convention, pointer names begin with p, as in ptr*



Equivalently:

```
int a = 1;
int *ptr; /* declare pointer */
ptr = &a; /* assign address of a to pointer */
```

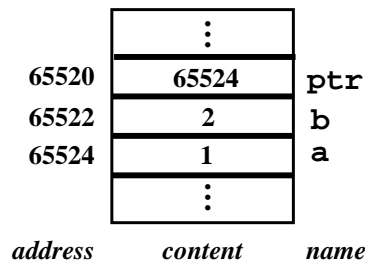
6

## Pointers (contd.)

```
#include "stdio.h"
void main(void){
    int a=1, b=2, *ptr=&a;
    printf("a = %d; address of a: %u\n", a, &a);
    printf("b = %d; address of b: %u\n", b, &b);
    printf("ptr = %u; address of ptr: %u\n", ptr, &ptr);
    printf("ptr points to the value %d\n", *ptr);
}
```

Produces the screen output:

```
a = 1; address of a: 65524
b = 2; address of b: 65522
ptr = 65524; address of ptr: 65520
ptr points to the value 1
```



*Note: except in a declaration, \*ptr means “the value pointed to by ptr”. This is called dereferencing the pointer ptr*

7

## Pointers (contd.)

```
int a=5, b=9, *ptr=&a;
/* assign the value pointed to by ptr to b */
b = *ptr; /* *ptr is another name for a here */
```

*Before b=\*ptr;*

*After b=\*ptr;*



```
int a=5, b=9, *ptr=&a;
/* assign value of b to whatever ptr points to */
*ptr = b; /* *ptr is another name for a here */
```

*Before \*ptr=b;*

*After \*ptr=b;*



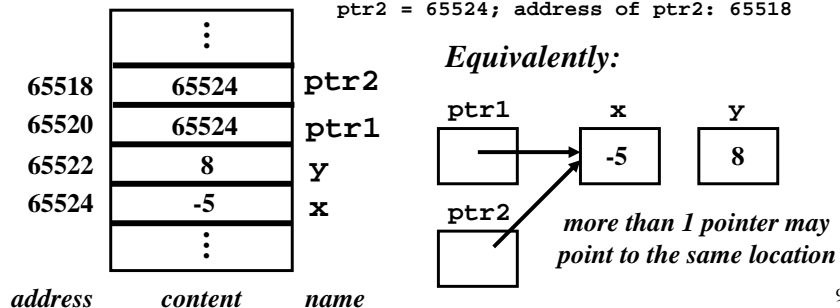
8

## Pointers (contd.)

```
int x=-5, y=8, *ptr1, *ptr2;
ptr1=&x; /* ptr1 now points to x */
ptr2=ptr1; /* ptr2 now also points to x */
printf("x = %d; address of x: %u\n", x, &x);
printf("y = %d; address of y: %u\n", y, &y);
printf("ptr1 = %u; address of ptr1: %u\n", ptr1, &ptr1);
printf("ptr2 = %u; address of ptr2: %u\n", ptr2, &ptr2);
```

Produces the screen output:

```
x = -5; address of x: 65524
y = 8; address of y: 65522
ptr1 = 65524; address of ptr1: 65520
ptr2 = 65524; address of ptr2: 65518
```



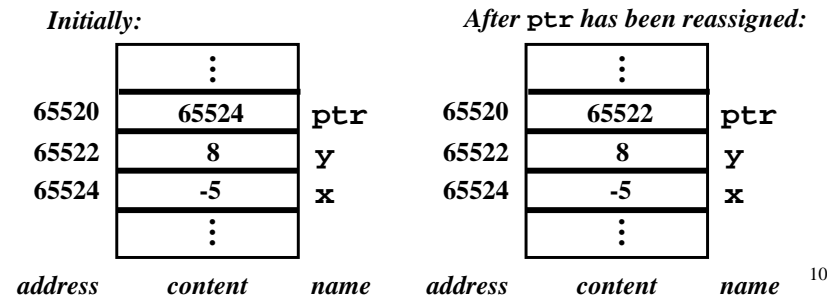
9

## Pointers (contd.)

```
int x=-5, y=8, *ptr;
ptr=&x; /* ptr now points to x */
printf("x = %d; address of x: %u\n", x, &x);
printf("y = %d; address of y: %u\n", y, &y);
printf("ptr = %u; ptr points to the value %d\n", ptr, *ptr);
ptr=&y; /* ptr now points to y */
printf("now ptr = %u; ptr points to the value %d\n", ptr, *ptr);
```

Produces the screen output:

```
x = -5; address of x: 65524
y = 8; address of y: 65522
ptr = 65524; ptr points to the value -5
now ptr = 65522; ptr points to the value 8
```



10

## Pointers: example

```
int intvar=3, *ptr1=&intvar;
float fltvar=1.5, *ptr2=&fltvar;
```

Expression	holds	evaluates to	address	content	name
intvar	int	3			
&intvar	address	65524	65516	65518	ptr2
ptr1	address	65524	65518		
&ptr1	address	65522		1.5	fltvar
*ptr1	int	3			
fltvar	float	1.5	65522	65524	ptr1
&fltvar	address	65518	65524		
ptr2	address	65518		3	intvar
&ptr2	address	65516			
*ptr2	float	1.5			

address      content      name

Note: here, pointer variables are stored in 2 bytes, regardless of the type of object they point to. Variables of type int are also stored in 2 bytes here; variables of type float are stored in 4 bytes here.

11

## Pointers and arrays

- In C there is a close relationship between pointers and arrays: the **name** of an array is a **pointer to the first element of the array**

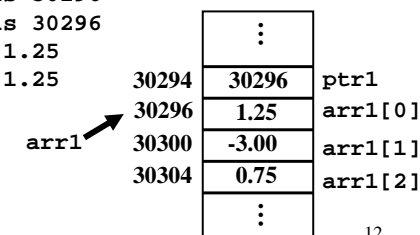
```
float arr1[3] = {1.25, -3, 0.75};
float *ptr1 = &(arr1[0]);
printf("address of arr1's 1st element is %u\n", ptr1);
printf("address of arr1's 1st element is %u\n", arr1);
printf("value of arr1's 1st element is %.2f\n", *ptr1);
printf("value of arr1's 1st element is %.2f\n", *arr1);
```

Produces the screen output:

```
address of arr1's 1st element is 30296
address of arr1's 1st element is 30296
value of arr1's 1st element is 1.25
value of arr1's 1st element is 1.25
```

An array name (e.g. arr1) is treated by the C compiler as a pointer to the first element of the array, arr1[0].

Therefore arr1 == &(arr1[0])  
and \*arr1 == arr1[0]

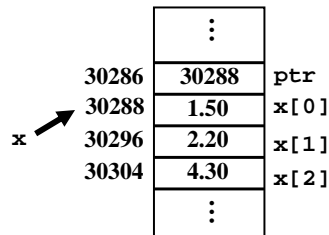


12

## Pointers and arrays: example

```
double x[3] = {1.5, 2.2, 4.3};
double *ptr = &x[0];
```

Expression	holds	evaluates to
x[0]	double	1.50
x[1]	double	2.20
x[2]	double	4.30
x	address	30288
&x[0]	address	30288
&x[1]	address	30296
&x[2]	address	30304
ptr	address	30288
&ptr	address	30286
*ptr	double	1.50
ptr+1	address	30296
*(ptr+1)	double	2.20
ptr+2	address	30304
*(ptr+2)	double	4.30



*Note: since each double is stored in 8 bytes here, and since ptr is a pointer to a double, the C compiler interprets ptr+1 as “the address of the next double after the address in ptr”, and ptr+2 as “the address of the double 2 locations away from the address in ptr”, etc.*

13

## Pointer arithmetic

- From the preceding example, you can see that the C compiler interprets expressions of the form **pointer+n** in a different way to the usual arithmetic addition
  - First the C compiler determines, based on what datatype **pointer** is pointing to, what “address unit” to use in evaluating the expression **pointer+n**
    - If **pointer** is a *pointer-to-double* and doubles are stored in 8 bytes, the address unit is 8
    - If **pointer** is a *pointer-to-int* and ints are stored in 2 bytes, the address unit is 2
  - Then the address unit is multiplied by **n** to figure out what the **offset** is to the address currently stored in **pointer**
  - This offset is added to the address in **pointer** to yield the address referred to by **pointer+n**
- Main use of all this: another way to access array elements...

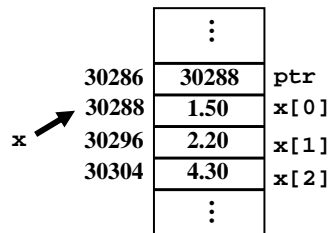
14

## Pointers and arrays (contd.)

- Pointer notation to access array elements is often preferred

```
double x[3] = {1.5, 2.2, 4.3};
double *ptr = &x[0];
```

Expression	equivalent to	evaluates to
*ptr	x[0]	1.50
*(ptr+1)	x[1]	2.20
*(ptr+2)	x[2]	4.30
x	&x[0]	30288
x+1	&x[1]	30296
x+2	&x[2]	30304
*x	x[0]	1.50
*(x+1)	x[1]	2.20
*(x+2)	x[2]	4.30



*Key difference: a pointer is a variable and so can be reassigned; an array name “pointer” is fixed at compile time and can’t be reassigned.*

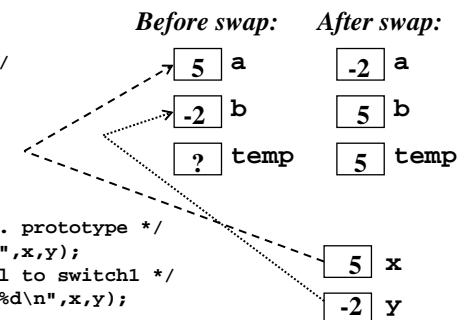
15

## Pointers and functions

- Why are pointers useful when used with functions?
  - A: because they provide a way for a called function to access the calling function’s actual parameters. For example, what if we wanted the called function to “return” more than 1 value?

Incorrect swap program: `switch1()` cannot change values of `x` and `y` in `main()`

```
#include <stdio.h>
void switch1(int a, int b){
    int temp; /* local variable */
    temp = a;
    a = b;
    b = temp;
} /* implicit return */
void main(void){
    int x=5, y=-2;
    void switch1(int, int); /* fn. prototype */
    printf("x is %d and y is %d\n",x,y);
    switch1(x,y); /* function call to switch1 */
    printf("now x is %d and y is %d\n",x,y);
}
```



Produces the screen output: `x is 5 and y is -2`  
`now x is 5 and y is -2`

16

## Pointers and functions (contd.)

- Provide called function with **addresses** of actual parameter variables in the calling function, if you want called function to modify actual parameters:

Correct swap program:

```
#include <stdio.h>
void switch2(int *a, int *b){
    int temp;
    temp = *a; /* *a is x from main() */
    *a = *b; /* *b is y from main() */
    *b = temp;
}
void main(void){
    int x=5, y=-2;
    void switch2(int *, int *);
    printf("x is %d and y is %d\n",x,y);
    switch2(&x,&y);
    printf("now x is %d and y is %d\n",x,y);
}
```

<i>Before swap:</i>	<i>After swap:</i>
<span style="border: 1px solid black; padding: 2px;">&amp;x</span> a	<span style="border: 1px solid black; padding: 2px;">&amp;x</span> a
<span style="border: 1px solid black; padding: 2px;">&amp;y</span> b	<span style="border: 1px solid black; padding: 2px;">&amp;y</span> b
<span style="border: 1px solid black; padding: 2px;">?</span> temp	<span style="border: 1px solid black; padding: 2px;">5</span> temp

<i>Before swap:</i>	<i>After swap:</i>
<span style="border: 1px solid black; padding: 2px;">5</span> x	<span style="border: 1px solid black; padding: 2px;">-2</span> x
<span style="border: 1px solid black; padding: 2px;">-2</span> y	<span style="border: 1px solid black; padding: 2px;">5</span> y

Produces the screen output: x is 5 and y is -2  
now x is -2 and y is 5

17

## Pointers and functions (contd.)

- Can also get the same effect by using pointers in the function call:

Another correct swap program:

```
#include <stdio.h>
void switch2(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
void main(void){
    int x=5, y=-2, *ptr1=&x, *ptr2=&y;
    void switch2(int *, int *);
    printf("x is %d and y is %d\n",x,y);
    switch2(ptr1,ptr2); /* no need for & here */
    printf("now x is %d and y is %d\n",x,y);
}
```

*Note: here, pointers ptr1 in main() and a in switch2() both point to x in main(); pointers ptr2 in main() and b in switch2() both point to y in main()*

Produces the screen output: x is 5 and y is -2  
now x is -2 and y is 5

18

## Pointers, functions and arrays

- Use `switch2()` to swap first and last elements in array in `main()`

```
#include <stdio.h>
void switch2(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
void main(void){
    int intarray[] = {5,7,9};
    int *ptr = &intarray[0];
    void switch2(int *, int *);
    printf("1st element is %d, ", intarray[0]);
    printf("last element is %d\n", intarray[2]);
    switch2(&intarray[0],&intarray[2]);
    printf("now 1st element is %d, ", intarray[0]);
    printf("last element is %d\n", intarray[2]);
}
```

*Note: instead of switch2(&intarray[0],&intarray[2]); could use switch2(intarray,intarray+2); or switch2(ptr,ptr+2);*

Produces the screen output: 1st element is 5, last element is 9  
now 1st element is 9, last element is 5

19

## Pointers, functions and arrays (contd.)

- Recall: arrays are passed by address. For example:

```
#include <stdio.h>
void zap(int array1[], int size) { /* knows address of charges[] */
    int i;
    for (i = 0; i < size; i++) {
        array1[i] = 0; /* sets all elements to 0 */
    }
}
void main(void) {
    int i, charges[5]={7,8,7,6,7};
    printf("elements of charges[] are ");
    zap(charges, 5); /* pass address of 1st element of charges[] */
    printf("\nnow elements of charges[] are ");
    for (i=0;i<5;i++){printf("%d ", charges[i]);}
}
```

array1[] in zap()  
is another name for  
charges[] in main()

Produces the screen output: elements of charges[] are 7 8 7 6 7  
now elements of charges[] are 0 0 0 0 0

20

## Pointers: example

What is the screen output of this program?

```
#include <stdio.h>
void main(void){
double a = 0.5;
double *ptr = &a;
a = a + 1.0;
*ptr = a + 3.0;
*ptr = (*ptr)*10.0;
printf("final value of a is %.2f\n", a);
}
```

21

## Pointers: another example

What is the screen output of this program?

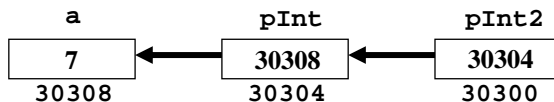
```
#include <stdio.h>
int func(int x, int *y){
    x = 6;
    *y = *y + 2;
    return x;
}
void main(void){
    int a=5, b=7, c;
    int func(int, int *);
    c = func(a, &b);
    printf("a is %d, b is %d, c is %d\n",a,b,c);
}
```

22

## A pointer to a pointer

```
#include <stdio.h>
void main(void){
int a=7;
int *pInt = &a;
int **pInt2 = &pInt;
/* pInt2 is a pointer to a pointer-to-int */
printf("a is %d, address of a is %u\n", a,&a);
printf("pInt is %u, address of pInt is %u\n", pInt,&pInt);
printf("pInt points to %d\n", *pInt);
printf("pInt2 is %u, address of pInt2 is %u\n", pInt2,&pInt2);
printf("pInt2 points to %u\n", *pInt2);
printf("pInt2 doubly points to %d\n", **pInt2);
}
```

Produces the screen output: a is 7, address of a is 30308  
pInt is 30308, address of pInt is 30304  
pInt points to 7  
pInt2 is 30304, address of pInt2 is 30300  
pInt2 points to 30308  
pInt2 doubly points to 7



23

## Introduction to Strings

Programming is about more than just numbers & arithmetic! Words, sentences, paragraphs: in computer-speak, represented with “strings”. You’ve already seen the use of **char**: a single character.

### A string is an array of characters ending with the NULL character

- the NULL character is written `'\0'` and has ASCII code 0
- note difference between NULL and the number **0**, which has ASCII code 48

A **string constant** is enclosed in double quotes, e.g. `"hello"`, `"I like C"`

- so far, all we’ve done is print them out, e.g. `printf("hello");`

To declare and initialise a string:

```
char a[6] = {'h','e','l','l','o','\0'};
alternatively, char a[6] = "hello"; /* more natural way */
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
h	e	l	l	o	\0

1

## Introduction to Strings (contd.)

Don’t have to specify size of array, if string is initialised when it’s declared:

```
char a[] = "hello"; /* same effect as before */
```

What if array size is larger than current string value?

```
char a[10] = "hello";
printf("string is: %s\n", a); /* use %s for string */
printf("values in char array are: ");
for (i=0;i<10;i++) {printf("%c, ASCII value %d\n",a[i],(int)a[i]);}
```

Produces the screen output:

```
string is: hello
values in char array are: h, ASCII value 104
e, ASCII value 101
l, ASCII value 108
l, ASCII value 108
o, ASCII value 111
, ASCII value 0
, ASCII value 0
, ASCII value 0
, ASCII value 0
, ASCII value 0
, ASCII value 0
```

String-terminating NULL

Here, all remaining array elements have also been set to NULL. In general, such values are system-dependent.

2

## Strings and pointers

Since a string is stored as an array of characters, the name of the string is treated by the C compiler as a **fixed** pointer-to-**char** whose value is the address of the first array element.

We can also declare a variable of type pointer-to-**char** and assign it the address of the first element of a character array (i.e. a string). The difference between this pointer and the “name of string” pointer is that the latter cannot be reassigned a different value.

```
char a[] = "hello";           used %s, so print out string contents
char *ptr1 = &a[0];
char *ptr2 = a; /* a == &a[0] */
printf("string is: %s\n",a);   used %u, so print out
                              location of string (i.e.
                              address of first element)
printf("string is stored at location %u\n",a);
printf("string is also: %s\n",ptr1);
printf("string is also also: %s\n",ptr2);
ptr1 = "hi there"; /* pointer a can't be reassigned like this */
printf("string is still: %s\n",a);
printf("but now ptr1 points to the string: %s\n",ptr1);
```

Produces the screen output:

```
string is: hello
string is stored at location 30296
string is also: hello
string is also also: hello
string is still: hello
but now ptr1 points to the string: hi there
```

3

## String input

We can use `%s` to read in a string from the keyboard:

```
char message1[80], message2[80];
printf("enter first string: ");
scanf("%s", message1); /* don't need & */
printf("first string is: %s\n", message1);
printf("enter second string: ");
scanf("%s", message2);
printf("second string is: %s\n", message2);
```

Produces the screen output:

```
enter first string: hello
first string is: hello
enter second string: hi there
second string is: hi
```

Problem: `scanf()` stops reading input when it reaches a whitespace character, such as the blank space between “hi” and “there”. In this case, `scanf()` only read in the string “hi”; a second `scanf()` statement would read in the string “there”.

This is a problem if we want to read in a complete line of text, since in general we don’t know in advance how many individual strings are contained in the line...

4

## String input (contd.)

Solution for reading in a complete line of text: read input character-by-character, stopping only when the newline character '\n' is encountered.

```
int i=0;
char message[80];
printf("enter string: ");
scanf("%c",&message[i]); /* read in first input character */
while (message[i]!='\n'){
    i++;
    scanf("%c",&message[i]); /* read in next input character */
}
message[i]='\0'; /* terminate string with NULL */
printf("string is: %s\n", message);
```

Produces the screen output:     enter string: hi there  
                                  string is: hi there

This solution is a bit complicated (!). There is an easier way – *C has special functions for inputting and outputting a single character:*

```
int getchar(void); /* returns ASCII value of next input character */
int putchar(int c); /* outputs character whose ASCII value is c */
```

Example: input a single character and immediately echo it to the screen

```
char ch = getchar(); /* read in a character from the keyboard */
putchar(ch); /* then output this character to the screen */     5
```

## String input (contd.)

So another solution for reading in a complete line of text is:

```
int i=0;
char message[80];
printf("enter string: ");
while ((message[i]=getchar())!='\n'){
    i++;
}
message[i]='\0'; /* terminate string with NULL */
printf("string is: %s\n", message);
```

Produces the screen output:     enter string: hi there  
                                  string is: hi there

Q: what happens if you forget to NULL-terminate the string?

6

## Strings and functions

Since the name of the string is treated as a pointer to the first element of the string, function calls involving strings use **call-by-address** (just like arrays):

```
#include "stdio.h"
void main(void){
    char message1[80]="hello";
    char *pmsg2="I like C";
    int strlen1(char str[]); /* function prototype */
    printf("\n%s\n" has length %d\n",message1,strlen1(message1));
    printf("\n%s\n" has length %d\n",pmsg2,strlen1(pmsg2));
}
int strlen1(char s[]){ /* no need to pass in size of array */
    int count=0;
    while (s[count]!='\0'){
        count++; /* increase by 1 as long as NULL not reached */
    }
    return count;
}
```

*s is the local name in strlen1()  
for the string in main() given as the  
actual parameter in the function call*

Produces the screen output: "hello" has length 5  
                                  "I like C" has length 8

7

## Strings and functions (contd.)

Instead of the called function using a character array, it could use a pointer variable:

```
#include "stdio.h"
void main(void){
    char message1[80]="hello";
    char *pmsg2="I like C";
    int strlen2(char *str); /* function prototype */
    printf("\n%s\n" has length %d\n",message1,strlen2(message1));
    printf("\n%s\n" has length %d\n",pmsg2,strlen2(pmsg2));
}
int strlen2(char *ps){ /* no need to pass in size of array */
    int count=0;
    while (*ps!='\0'){
        count++; /* increase by 1 as long as NULL not reached */
        ps++; /* move pointer on to next element */
    }
    return count;
}
```

*ps is a local pointer in strlen2()  
pointing to the string in main() given as  
the actual parameter in the function call*

Produces the screen output: "hello" has length 5  
                                  "I like C" has length 8

8

## Strings and functions (contd.)

In either case, since the location of the string is passed by the calling function, the called function can change the contents of the string which is given as the actual parameter:

```
#include "stdio.h"
void main(void){
    char message[80]="hello";
    void bye(char *str); /* function prototype */
    printf("string is \"%s\"\n",message);
    bye(message); /* function call */
    printf("now string is \"%s\"\n",message);
}
void bye(char *p){
    *p='g';*(p+1)='o';*(p+2)='o';*(p+3)='d';
    *(p+4)='b';*(p+5)='y';*(p+6)='e';*(p+7)='\0';
}
```

Produces the screen output: string is "hello"  
now string is "goodbye"

9

## Library string functions

The standard C library contains several useful functions for manipulating and examining strings. Must `#include <string.h>` to get them to work.

Suppose `ps` is a pointer to the string `s`, and `pt` is a pointer to the string `t`

```
strlen(ps)
/* returns integer == length of s (not counting NULL) */

strcmp(ps,pt)
/* returns negative integer if s<t (alphabetically), returns */
/* 0 if s==t, and returns positive integer if s>t */
```

[Alphabetically: a<b, an<at, at == at, z>x, fire>field, etc.]

```
strcpy(ps,pt)
/* copies contents of t into s, overwriting previous contents */
/* of s, and returns pointer to new 1st character of s */
```

Use these string library functions whenever possible! There are other string library functions you may find useful; see any C textbook (or the C standard library) for details...

10

## Library string functions: Example

```
#include <stdio.h>
#include <string.h>
void main(void){
    int i, j;
    char *p_result;
    char s[]="C is easy to learn", t[]="C is hard to learn";
    char *ps = s, *pt = t;
    printf("length of s is %d\n",strlen(ps));
    i=strlen(pt);
    printf("length of t is %d\n",i);
    j=strcmp(ps,pt);
    if (j<0) {printf("string s comes before string t\n");}
    else if (j==0) {printf("strings equal\n");}
    else {printf("string s comes after string t\n");}
    printf("s is \"%s\"\n",s);
    p_result=strcpy(ps,pt);
    printf("now s is \"%s\"\n",s);
    printf("also: s is \"%s\"\n",p_result);
}
```

Produces the screen output: length of s is 18  
length of t is 18  
string s comes before string t  
s is "C is easy to learn"  
now s is "C is hard to learn"  
also: s is "C is hard to learn"

11

## String application: "uppercaseify"

Problem: write a C program which changes every lowercase letter in a string into uppercase.

```
#include <stdio.h>
void main(void){
    int i;
    char s[]="This is just a Test";
    printf("string is \"%s\"\n",s);
    for (i=0;s[i]!='\0';i++){ /* keep going until NULL reached */
        if ((s[i]>='a') && (s[i]<='z')){ /* lowercase letter only */
            s[i] = s[i] + 'A' - 'a'; /* uses ASCII values... */
        }
    }
    printf("now string is \"%s\"\n",s);
}
```

Produces the screen output: string is "This is just a Test"  
now string is "THIS IS JUST A TEST"

Reminder: this works because in ASCII, a-z and A-Z follow each other in sequence (so do 0-9).

If `s[i]` is 'a': `s[i]+'A'-'a' == 'a'+'A'-'a' == 'A'`.

If `s[i]` is 'b': `s[i]+'A'-'a' == 'b'+'A'-'a' == 'a'+1+'A'-'a' == 'A'+1 == 'B'`.

If `s[i]` is 'c': `s[i]+'A'-'a' == 'c'+'A'-'a' == 'a'+2+'A'-'a' == 'A'+2 == 'C'`, etc.

12

## Introduction to File Input/Output

### Introduction

- All Input/Output up to now has been done from/to the keyboard/screen.
- There is a serious limitation with this: *when the program terminates, the data is lost*. So we can't write a database program, which stores information and retrieves this information next time the program is started. Or a word processor. Or an email system. Or a spreadsheet. Or ...
- We can however use external storage devices like the computer's hard disk for storage. Instead of writing output to the screen, we can write it to a file. *This data won't be lost when the program terminates*.

### Binary and Text Files

C can handle two types of files: *binary* files and *text* (or ASCII) files.

- The difference between the two file types is in the way they store numeric data. If numeric data is stored in a binary file, it is stored as binary numbers. If numeric data is stored in a text file, each digit of the numeric data is converted to its ASCII format and stored like that.
- So the number 123 requires three bytes of storage in a text file because it has three digits: '1', '2', and '3'.

1

## Introduction to File Input/Output (contd.)

- 123 only requires 2 bytes of storage in a binary file because the binary representations of integer values take up two bytes. 123 is stored as

00000000	01111011
----------	----------

- So binary files are more compact for storing data which is primarily numeric. Also, numeric data must be converted to its ASCII form before writing to a text file and converted from its ASCII form when reading from a text file. With binary files, no such conversions are necessary.
- However, text files are easily displayed on the screen (using e.g. the Windows notepad editor), so their contents can be easily examined or checked. They can also be read easily by other programs. Neither of these is true for binary files.

### Important concepts:

- **File** -- a named set of data stored together on the disk. C views any file as a stream of bytes – it's up to the programmer to give it a structure.
- to use: (1) **open** the file; (2) read/write from/to the file; (3) **close** file. 2

## Input from File – Example

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
/* fp is a variable of type "FILE *" and is */
/* called a file pointer. The datatype */
/* "FILE" is defined in stdio.h, and is a */
/* structure in which C keeps information */
/* about a file */
fp=fopen("datafile.txt", "r"); /* open read-only */
fscanf(fp, "%d %d", &var1, &var2);
fclose(fp);
printf("first value read in was %d\n", var1);
printf("second value read in was %d\n", var2);
}
```

Contents of file datafile.txt:        3  
   -1

Produces the screen output:    **first value read in was 3**  
   **second value read in was -1**

3

## Input from File – Example (contd.)

```
fp=fopen("datafile.txt", "r");
```

This statement "associates" the file **datafile.txt** with the file pointer **fp**, and tells the C compiler that the program is opening **datafile.txt** for reading only.

2<sup>nd</sup> argument to **fopen()** gives the file's "open mode" – possibilities include:

- r**    open an existing file for *reading only*
- w**    open a new file (or overwrite an existing file) for *writing only*
- a**    open a file for *appending* (writing at the end of the file) – creates a new file if it doesn't currently exist
- r+**   open an existing file for *reading and writing*, starting at the beginning of the file
- w+**   open a new file (or overwrite an existing file) for *reading and writing*
- a+**   open a file for *reading and appending* – creates a new file if it doesn't currently exist

4

## Input from File – Example (contd.)

```
fscanf(fp, "%d %d", &var1, &var2);
```

Enables formatted input from the file pointed to by **fp**, otherwise has the same functionality as **scanf()**

```
fclose(fp);
```

When program is finished with an open file, it should “break” the association that was formed when the file was opened. This **fclose()** statement closes the file pointed to by **fp**.

Any files which have not been closed by **fclose()** are automatically closed (by the Operating System) when program execution is finished.

5

## Output to File – Example

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
FILE *fptr;
fp=fopen("datafile.txt", "r");
fscanf(fp, "%d %d", &var1, &var2);
fclose(fp);
fptr=fopen("datafile.txt", "w"); /* open write-only */
fprintf(fptr, "%d\n%d", var2, var1); /* swap values */
fclose(fptr);
}
```

Contents of file datafile.txt before program execution: 3  
-1

Contents of file datafile.txt after program execution: -1  
3

(so in this case, the contents of **datafile.txt** have been overwritten)

6

## Output to File – Example (contd.)

```
fptr=fopen("datafile.txt", "w");
```

Now we open **datafile.txt** again, this time for writing. We use a different file pointer, **fptr**, to point to the opened file.

```
fprintf(fptr, "%d\n%d", var2, var1);
```

Enables formatted output to the file pointed to by **fp**, otherwise has the same functionality as **printf()**

```
fclose(fptr);
```

Again, it is important to close the file when the program is finished with it.

7

## Output to File – Example (alternative solution)

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
fp=fopen("datafile.txt", "r");
fscanf(fp, "%d %d", &var1, &var2);
fp=fopen("datafile.txt", "w"); /* change mode */
fprintf(fp, "%d\n%d", var2, var1);
fclose(fp);
}
```

Contents of file datafile.txt before program execution: 3  
-1

Contents of file datafile.txt after program execution: -1  
3

8

## Output to File – Example of appending

```
#include <stdio.h>
void main(void){
int var1, var2; /* not initialised */
FILE *fp;
FILE *fptr;
fp=fopen("datafile.txt", "r");
fscanf(fp, "%d %d", &var1, &var2);
fclose(fp);
fptr=fopen("datafile.txt", "a");
fprintf(fptr, "%d\n%d", var2, var1);
fclose(fptr);
}
```

Contents of file datafile.txt before program execution: 3  
-1

Contents of file datafile.txt after program execution: 3  
-1  
-1  
3

9

## Reading data from a file

First, you have to know the name of the file and where it is stored.

You also need to know how the data is laid out in the file. In particular:

- Order and datatype(s) of the data values.
- How much data is in the file. 3 common methods for this:
  - First line tells the number of data “records” that follow.
  - Sentinel signal: a value outside the range of actual data values which indicates that the end of the data has been reached.
  - Test to see if the special *end-of-file indicator* (inserted into every file by the Operating System) has been reached. This is usually the best choice.

Of course, if you write data to a file, you may have to decide which way of storing the data is most suitable for your application.

10

## Reading data from a file: Examples

Contents of datafile1.txt: 3  
0.2 80  
0.7 60  
0.1 50

(here, the first value tells us the number of lines of data – to read them, you should use a *counter-controlled loop*...)

Contents of datafile2.txt: 0.2 80  
0.7 60  
0.1 50  
-99 -99

(here, the sentinel values tell us that the end of the data has been reached – you should use a *loop and test for the sentinel values each time around*...)

Contents of datafile3.txt: 0.2 80  
0.7 60  
0.1 50

(here, the file is just data – you should check whether each read got the end-of-file indicator. An example program to do this is given next)

11

## Reading data from a file: Example program

```
#include <stdio.h>
void main(void){
float prob, avg=0.0;
int quantity, num_values=0;
FILE *fptr;
fptr=fopen("datafile3.txt", "r");
while(fscanf(fptr, "%f %d", &prob, &quantity)==2){
/* if return value from this fscanf() is not 2, */
/* the end-of-file indicator has been reached */
avg = avg + (prob*quantity);
}
fclose(fptr);
printf("average value is %.2f\n", avg);
}
```

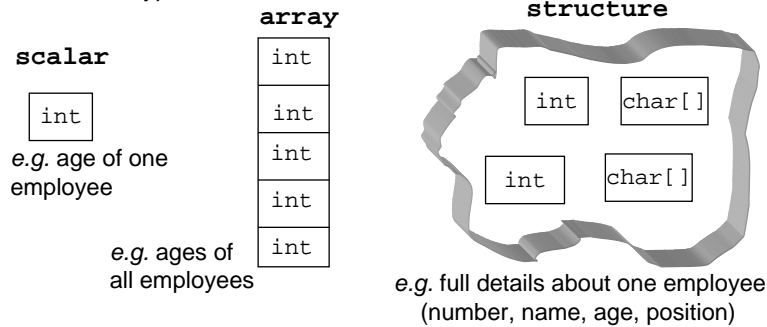
Produces the correct screen output: average value is 63.00

Note: return value from `fscanf()` – and `scanf()` – is the number of successful assignments. So in this case, this return value should be 2 every time a valid data “record” has been read. Otherwise – error or end-of-file.

12

## Structures

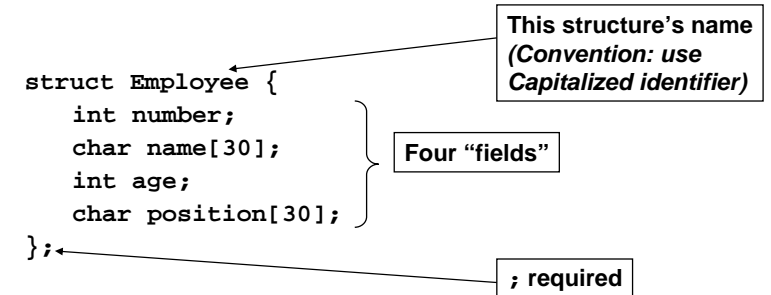
- A **scalar** holds a **single value** of a **single type**
- An **array** holds **several values** of a **single type**
- It is sometimes convenient to be able to group together items of information which are of **different types**
  - Example: an employee's **number**, **name**, **age** and **position** are logically related, but these pieces of data are of different types and so are not suitable for storage in an array
- A **structure** allows the programmer to group together data items of different types



1

## Defining Structures

- To define a structure we must create a **structure template**:



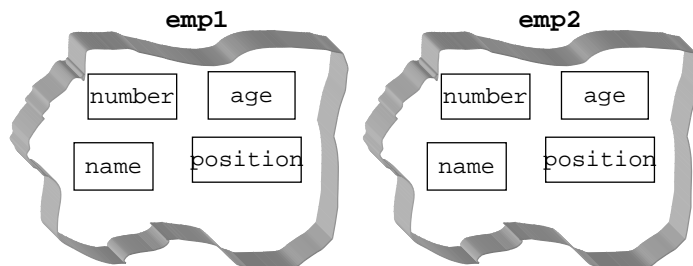
- **Does not allocate any memory** or define a new variable
- Just **defines a new datatype**, called **Employee**

2

- Now that we've defined the new datatype, we can define variables to be of type **Employee** (just as we can define variables to be of type **int**, **char**, **float**, ...):

```
struct Employee emp1, emp2;
```

- This declares two variables, **emp1** and **emp2**, to be of type **Employee**

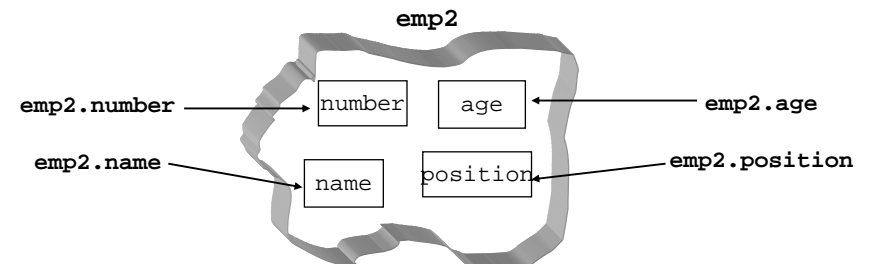


- Each of these structure variables has four fields: **number**, **name**, **age** and **position**

3

- In order to access structure fields, use the **field selection operator** .
 

```
emp1.number = 4321;
emp1.age = 34;
strcpy(emp1.name, "John Smith");
printf("what is %d's position:", emp1.number);
scanf("%s", emp1.position);
```
- Each member can be treated just like any other value of that datatype: can use string functions on **name** and **position**, can use **age** and **number** for arithmetic, etc.
  - Similar to arrays: if **a[]** is an array of **floats**, then **a[i]** can be treated just like a "normal" **float** for any valid subscript **i**



4

- If we wanted to print out the contents of this structure we could do it as follows:

```
printf("employee number: %d\n", emp2.number);
printf("\tname: %s\n", emp2.name); /* \t means tab */
printf("\tage: %d\n", emp2.age);
printf("\tposition: %s\n", emp2.position);
```

- Often a good idea to define a function to print out structures of this type (saves you rewriting this over and over; makes your program easier to understand, improve, debug, ...):

```
void printEmployee (struct Employee e){
    printf("employee number: %d\n", e.number);
    printf("\tname: %s\n", e.name);
    printf("\tage: %d\n", e.age);
    printf("\tposition: %s\n", e.position);
}
```

- Note: `printEmployee()` has to know what an `Employee` structure is...

```
#include <stdio.h>
struct Employee {
    int number;
    char name[30];
    int age;
    char position[30];
};
void printEmployee (struct Employee e){
    printf("employee number: %d\n", e.number);
    printf("\tname: %s\n", e.name);
    printf("\tage: %d\n", e.age);
    printf("\tposition: %s\n", e.position);
}
main(){
    struct Employee emp;
    emp.number = 4321;
    emp.age = 34;
    strcpy(emp.name, "John Smith");
    printf("what is %d's position:", emp.number);
    scanf("%s", emp.position);
    printEmployee(emp);
```

**Produces the screen output:**  
 what is 4321's position:lecturer  
 employee number: 4321  
                   name: John Smith  
                   age: 34  
                   position: lecturer

Note: this is an example of a *call-by-value* function call

```
#include <stdio.h>
struct Employee {
    int number;
    char name[30];
    int age;
    char position[30];
};
void zapEmployee (struct Employee *pe){
    (*pe).number=-1;
    strcpy((*pe).name, "invalid");
    (*pe).age=-1;
    strcpy((*pe).position, "invalid");
}
void printEmployee (struct Employee e){
    printf("employee number: %d\n", e.number);
    printf("\tname: %s\n", e.name);
    printf("\tage: %d\n", e.age);
    printf("\tposition: %s\n", e.position);
}
main(){
    struct Employee emp;
    emp.number = 4321;
    emp.age = 34;
    strcpy(emp.name, "John Smith");
    printf("what is %d's position:", emp.number);
    scanf("%s", emp.position);
    printEmployee(emp);
    zapEmployee(&emp);
    printEmployee(emp);
}
```

**Produces the screen output:**  
 what is 4321's position:lecturer  
 employee number: 4321  
                   name: John Smith  
                   age: 34  
                   position: lecturer  
 employee number: -1  
                   name: invalid  
                   age: -1  
                   position: invalid

`pe` is a *pointer* to a structure of type `Employee`

Note: this is an example of a *call-by-address* function call

## Pointers to Structures: A shorthand

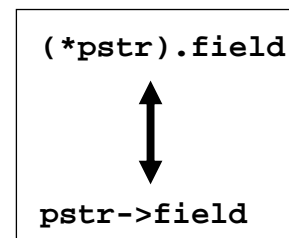
Expressions like this are used a lot:

`(*pe).number`

C has a shorthand notation for this:

`pe->number`

Or more generally, if `pstr` is a pointer to a structure, and `field` is one of that structure's fields, then these two mean the same thing:



**EXAMPLE:**

```
void zapEmployee (struct Employee *pe){
    pe->number = -1;
    strcpy(pe->name, "invalid");
    pe->age = -1;
    strcpy(pe->position, "invalid");
}
```

## Nested Structures

- Quite complex structures can be put together by **nesting structures**: using a structure for a field of another structure. For example, suppose we defined a structure template as follows:

```
struct Address {
    int number;
    char street[20];
    char city[20];
};
```

- We could then incorporate this into our **Employee** structure template as follows:

```
struct Employee {
    int number;
    char name[30];
    int age;
    char position[30];
    struct Address addr;
};
```

9

## Nested Structures (contd.)

- We can then declare a variable of this type:  
`struct Employee emp;`
- We could then assign values such as:  
`emp.number = 22; /* sets the Employee field */`  
`emp.addr.number = 56; /* sets the Address field */`  
`strcpy(emp.addr.street, "Elm Street");`

an **Address**  
structure  
that structure's  
**street** field

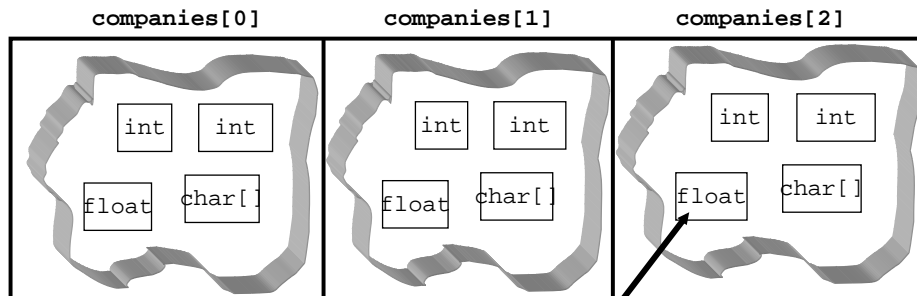
- Note that both **Employee** and **Address** have a field called **number**. These are different -- the structure will have two fields, one for each. This is OK -- there is no ambiguity, because you must include the appropriate "." to tell the compiler which field you intend.

10

## Arrays of structures

```
struct Company {
    int sales;
    int profit;
    char name[30];
    float PERatio;
};
struct Company companies[3];
```

This declares a 3-element array called **companies**, each of whose elements is a **Company** structure.



11