A Review of C

1. Introduction

High-Level Languages (Examples)

There are a large number of high-level languages. Common ones include:

- FORTRAN (Formula Translation) 1954
- Pascal 1971
- COBOL (business data processing) 1960
- C 1972, C++ 1985
- Java 1995
- C# (C Sharp) 2000
- Perl 1987
- Python 1991

Language Classifications

Programming Languages can also be classified in other ways. For example

- General Purpose (C, C++), Special Purpose
- Procedural (C, Pascal), Object-Oriented (C++, Java, C#)
- Procedural, Declarative (Prolog)

About C

- C was created by Dennis Ritchie at the Bell Labs, U.S.A. in the early 1970s.
- It was originally meant for systems programming, i.e. for writing operating systems (UNIX) and compilers.
- It is now a very widely used, general purpose programming language.
- C++ extends C to include *object-oriented extensions*.

Language Standard

- Each high-level language has a language standard that describes its syntax (grammar). This is usually contained in a published document.
- The syntax rules are very strict. Deviations will result in syntax errors.

We study ANSI/ISO C (**C89**) in this module. Some features of **C99** will also be discussed. Latest standard is **C11**, released in Dec 2011.

The Programming Process

1. Problem Solving:
   (a) Problem Specification & Analysis
   (b) Algorithmic Design

2. Implementation
   (a) Coding
   (b) Testing

Algorithm

- An algorithm is a set of precisely stated, finite sequence of executable steps for solving a problem.
- Finding a suitable algorithm is frequently the most difficult part of the problem solving process.
- An algorithm is usually written using informal English-like statements known as pseudocode or represented using flowcharts.

Pseudocode Example 1

To calculate the volume of a cylinder, given its radius and height.

```
INPUT radius, height
CylinderVol=3.14159*radius*radius*height
OUTPUT CylinderVol
```

Pseudocode Example 2a

To find the larger of two user-input numbers.

```
INPUT number1, number2
IF number1 > number2 THEN
   SET max = number1
ELSE
   SET max = number2
OUTPUT max
```

Pseudocode Example 2b

To find the larger of two user-input numbers.

```
INPUT number1, number2
SET max = number1
IF max < number2 THEN
   SET max = number2
OUTPUT max
```

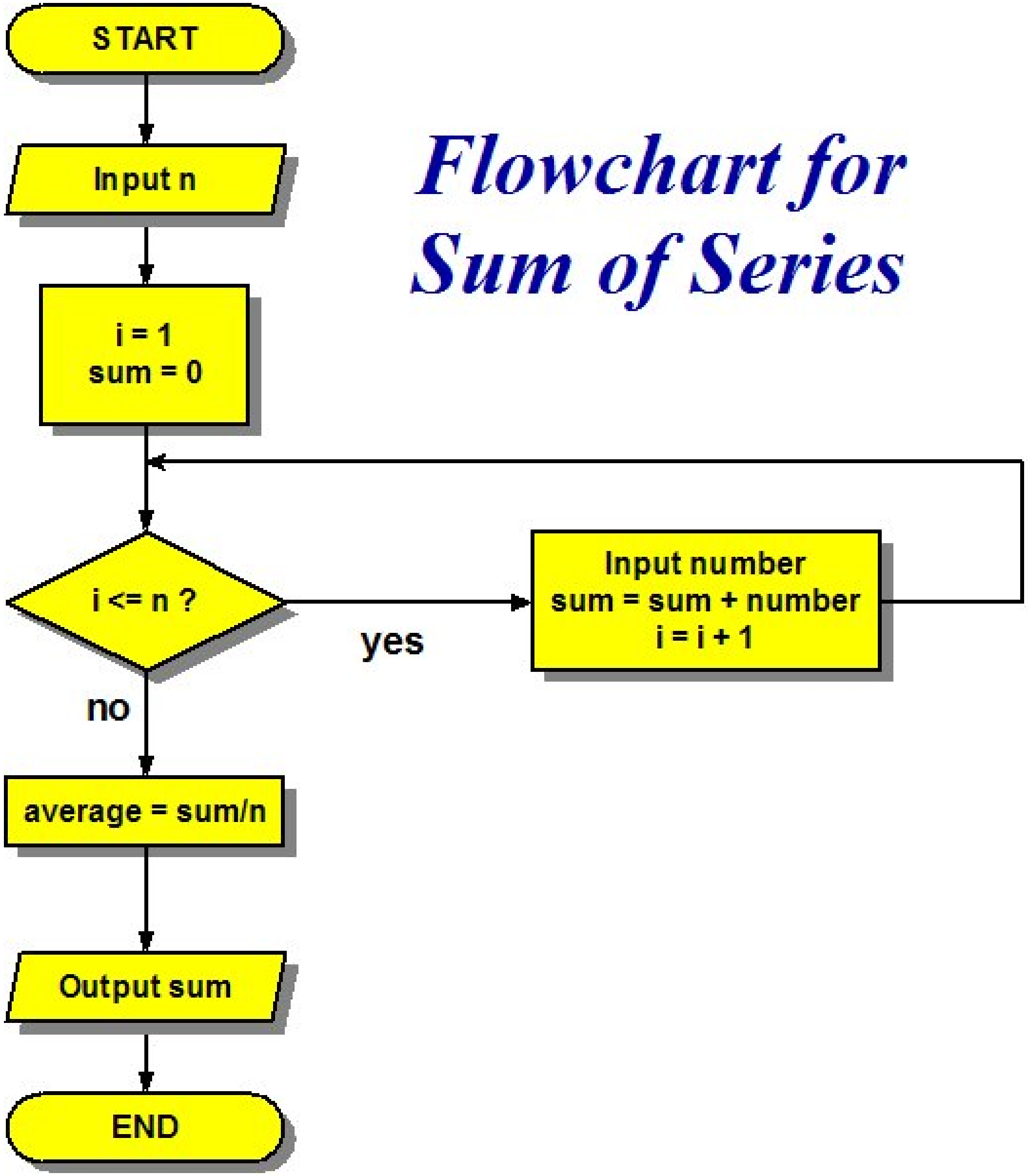
Note: This is better than Example 2a. Why? Try with 3 or 4 numbers.

Pseudocode Example 3

Accept n numbers and calculate the average.

```
Input n
Set sum = 0
Set counter = 1
WHILE counter <= n
        Enter number
        Add number to sum
        Add 1 to counter
END WHILE
Divide sum by n to get average
Output the average
```

Flowchart for Sum of Series



START

Input n

i = 1
sum = 0

i <= n ?

yes

Input number
sum = sum + number
i = i + 1

no

average = sum/n

Output sum

END

Coding

- Translate algorithm into a program using a programming language -- **coding**.
- **Source code** – All the statements in a program.
- How to type in the program statements?

  Use a text editor.
- We normally use an **IDE** (Integrated Development Environment) which contains an editor and other tools
- Example

  CodeBlocks, Orwell Dev-C++, Visual C++

Edit, Compile & Link Cycle

1. Create the source program (**simple.c**) by typing program statements into an editor.
2. Use a **compiler** to translate the source program into machine language resulting in an object file (**simple.obj, simple.o**) if there are no errors. Then go to Step 4.
3. If there are errors, **edit** to remove the errors. Then go back to Step 2.
4. Use a **linker** to combine this object file with other components to get a stand-alone executable file (**simple.exe**).

Errors in Programs

Errors in programs are called **bugs**. The process of detecting and correcting such errors is known as **debugging**.

3 types of errors:

- Syntax errors –- Errors in grammar
- Logic errors –- Errors caused by mistakes in formula, reasoning, etc.
- Runtime errors –- Executing syntactically correct instructions but with erroneous data. Example: division by zero.

A Review of C

2. Basic Concepts

Objectives

- Introduce fundamental concepts in programming through examples.

- Concepts:
  - statements and comments
  - preprocessor directives
  - functions in C
  - identifiers and declarations
  - data types, operators

Program 2.1

```
/* A very simple C program */
#include <stdio.h>
int main()
{


   printf("My first C program.\n");
   printf("Programming is fun!");




   return 0;
}
```

- This is a comment.
- A preprocessor directive.
- The main() function: starting point of program execution.
- { } enclose the function body.
- A semicolon ends a C statement.
- printf() is a function for displaying information on the screen.
- \n is the "newline" character
- Returns a value to the OS; required by ANSI C.

Program Comments

- Comments are included for human readers to understand the purpose of the program or statement—an important part of program **documentation**. They are ignored by the compiler.

```
/* . . . */
```

- A comment may occupy several lines, e.g.

```
/* This is a
     two-line comment  */
```

- We may also use single-line comments (allowed by the newer C99 standard):

```
// This is a single-line comment.
```

Preprocessor Directives

- A **preprocessor directive** which tells the compiler how to process the pieces of a program and where to find essential definitions. Example,

  ```
  #include <stdio.h>
  ```

- In the above, **stdio.h** stands for

  standar**d** **i**nput **o**utput **h**eader file.

- This is a file found in C systems, giving information on input and output functions such as printf().

Functions in C

- A function is a block of program statements that performs a specific task.
- Each function has a unique name followed by a pair of brackets. The statements in a function are enclosed between curly braces { … }.
- Functions are building blocks for C programs.
- Functions may be pre-written as part of a ***standard library*** or may be created by you.

Functions in C

- main() is a function. It is the starting point of execution for all C programs.
- There must be only **ONE** main() function in any C program.
- The **int** before main() means that main() returns an integer value to the OS through the statement: **`return 0;`**
- This returned value will not be used in this course. It basically means that the program ends normally.

Program 2.2

```
#include <stdio.h>
int main()
{   int num1, num2, sum;


   num1 = 1;
   num2 = 10;

   sum = num1 + num2;
   printf("The sum is %d.\n", sum);
   return 0;
}
```

- Declaration of **identifier**s num1, num2, sum. Must be done before their use and before any executable statement.
- Assignment statements. Left side must be the name of a variable.
- Arithmetic operation.
- %d is for outputting an integer value.

Variables and Identifiers

```
int num1, num2, sum;
```

- num1, num2 and sum are **variables**.
- A variable is a name associated with a *memory location*. Example (Values are computer-dependent):

      Address of num1 is 2686748
      Address of num2 is 2686744
      Address of sum   is 2686740

- The name of a variable is also known as an **identifier**.
- The above statement is known as a **declaration** statement.
- It declares that 3 variables num1, num2 and sum are to be used, and the variables are of integer **data type**.

Variables and Identifiers

- It is essential to state the data type in the declaration statement because the computer must allocate the correct amount of memory space based on the data type (4 bytes each for the integer variables here).
- All variables must be declared before they can be used.
- A variable can also be ***initialized*** with a value when it is declared.
- Example:

```
   int num1 = 1, num2 = 10, sum;
```

- The variables **num1** and **num2** are given values 1 and 10 when declared instead of using two separate statements to give them values as in Program 2.2.

Keywords

- Keywords are reserved words that have standard, predefined meanings. There are 32 keywords in Standard C (C89).
- Some keywords:

```
•char     int      float    double
•if       else     case     switch
•break    default  continue for
•while    do       return   void
```

Rules for Naming Identifiers

- It can contain letters, digits, and the underscore character (_).
- The first character must be either alphabetic or the underscore character.
- It is case sensitive.
- C keywords cannot be used.
- *Valid* names: sum, average, number_1, _id
- *Invalid* names: 1name, stud#, exam mark, int
- Always choose meaningful names.

Data Types

C has the following basic data types:

Data Type	Description	Memory Requirement	Range of values
char	Single character	1 byte	Also an int type -128 to 127
int	Integer	4 bytes	-2,147,483,648 to 2,147,483,647
float	Real numbers Single-precision	4 bytes	1.2E-38 to 3.4E38 6-7 digits of precision
double	Real numbers Double -precision	8 bytes	2.2E-308 to 1.8E308 15-16 digits of precision

short, long, signed, unsigned

The int data types can be further subdivided.

Data Type	Description
unsigned char	0 to 255
signed char	Same as char: -128 to 127
short int (signed / unsigned)	Can be shortened to ***short***, 2 bytes -32768 to 32767 (signed) 0 to 65535 (unsigned)
long int (signed / unsigned)	Same as int (4 bytes) -2,147,483,648 to 2,147,483,647 (signed) 0 to 4,294,967,295 (unsigned)

Program 2.3: sizeof.c

```
/* sizeof.c -- to find out the size of data types */

#include <stdio.h>
int main()
{
     printf( "A char      is %d byte\n", sizeof( char ));
     printf( "An int      is %d bytes\n", sizeof( int ));
     printf( "A short     is %d bytes\n", sizeof( short ));
     printf( "A long      is %d bytes\n", sizeof( long ));
     printf( "An unsigned char  is %d byte\n",
              sizeof( unsigned char ));
     printf( "A float     is %d bytes\n", sizeof( float ));
     printf( "A double    is %d bytes\n", sizeof( double ));

     return 0;
 }
```

Constants

- Integer constants

  1, 20, 5000

- Floating-point constants

  3.14159265, 1.5E-2, 1.5e-2

- Character constants

```
char grade;
grade = 'A';
grade = 65;
```

- String constants

  "This is a string constant."

The ASCII Character Set

- Each character constant has an integer value determined by a character set.
- The most common is the ASCII character set.
- **ASCII** – American Standard Code for Information Interchange
- In Standard C, each character is represented by a 7-bit number.

The ASCII Table

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	“	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Escape Sequences

- **\n** escape sequence, \ escape character

Sequence	Meaning
\n	new line
\t	horizontal tab
\b	backspace
\0	null char

Sequence	Meaning
\'	single quote
\"	double quote
\\\\	backslash

Example

- To output a double quote "

```
printf("This is a double quote ".\n");
```

This is wrong!

- It should be

```
printf("This is a double quote \".\n");
```

Program 2.4

```
#include <stdio.h>
int main()
{  float radius, area;



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

   area = 3.14159265*radius*radius;
   printf("The area is %f.\n", area);

   return 0;
}
```

- Declaration of floating-point variables radius, area.
- Prompting statement
- scanf() captures keyboard input from user.

Program 2.4

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

- %f is known as a **conversion specifier**.

  It tells the computer that a floating-point number is to be input.

- Note the ampersand sign (&) before the variable name.

- Use *%d* for int, *%c* for char, *%lf* for double.

Program 2.4

- In printf, %f will print the value of **area** with 6 decimal places. The output format can be controlled by using %f in the modified form:

  %m.nf

  where m is known as the **field width** and n is the number of decimal places.

- Example:

  %10.3f will print **area** (assuming that it is 12.566370614) as:

  □ □ □ □ **12.566**

  where □ denotes a blank space.

Program 2.4

- We could have used the double data type:

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

- Note that **`%lf` MUST** be used in scanf in this case.
- But we may use either %lf or %f in printf for output.

Program 2.4b

```
#include <stdio.h>
int main()
{  double radius, area;

   printf("Enter radius: ");
   scanf("%lf", &radius);
   area = 3.14159265*radius*radius;
   printf("The area is %f.\n", area);
   return 0;
}
```

Field Type Conversion Specifiers

Character	Argument	Output
%d	integer	signed integer
%f	float	floating point number
%e	float	float using e notation
%E	float	float using E notation
%c	character	a single character

Conversion specifiers: %c, %d

```
#include <stdio.h>
int main()
{ char ch = 'A';   int number = 25;

  printf("%c\n", ch);    //A
  printf("%5c\n", ch);   //□□□□A
  printf("%-5c\n", ch);  //A□□□□

  printf("%d\n", number);    //25
  printf("%5d\n", number);   //□□□25
  printf("%-5d\n", number);  //25□□□

  return 0;
}
```

Float type conversion specifiers

- %f, %g, %e, %E
- Example: if number = 12.34

  %f gives 12.340000

  %g gives 12.34

  %e gives 1.234000e+001

  %E gives 1.234000E+001

- Example: if number = 0.0001234

  %f gives 0.000123

  %e gives 1.234000e-004

Program 2.5: Symbolic Constant

```
#include <stdio.h>
#define PI 3.14159265    /* PI is defined as
                         a symbolic constant
                         */

int main()
{  double radius, area;

   printf("Enter radius: ");
   scanf("%lf", &radius);
   area = PI*radius*radius;
   printf("The area is %f.\n", area);
   return 0;
}
```

Macros

- We have used the preprocessor directive:

```
#define PI 3.14159265
```

  This is also called a **macro**.

- ***General Form***: ***#define*** macro_name replacement
- **Example:**

```
#define CIRCLE_AREA(r)  (3.14159*(r)*(r))
```

- **How it is used:** Suppose we write

```
volume = CIRCLE_AREA(1.5)*height;
```

  The preprocessor will automatically replace it by

```
volume = (3.14159*(1.5)*(1.5))*height;
```

Program 2.6: Macro

```
#include <stdio.h>
#define PI 3.14159265
#define CIRCLE_AREA(r)  (PI*(r)*(r))

int main()
{  double radius, height=1, volume1, volume2;

   printf("Enter radius: ");
   scanf("%lf", &radius);
   printf("The circle area is %8.3lf\n", CIRCLE_AREA(radius));

   volume1 = CIRCLE_AREA(1.5)*height;
   volume2 = CIRCLE_AREA(radius)*height;

   printf("Radius=1.5 Volume 1=%8.3lf\n", volume1);
   printf("Radius=%g, Volume 2=%8.3lf\n", radius, volume2);

   return 0;
}
```

Macros

- Note that the macro **PI** defined above is **not** a variable. It does not use memory space.
- Macros are frequently used to define physical/chemical and other constants:

```
#define LIGHT_SPEED 299792458.0
#define GRAVITY_CONST 6.67384E-11
#define GAS_CONST 8.3144621
#define GST 0.07
```

- Note that using names instead of numbers makes programs easier to understand.

The const keyword

- We can also use the `const` keyword to define these values:

```
const double LIGHT_SPEED = 299792458.0;
const double GRAVITY_CONST = 6.67384E-11;
const double GAS_CONST = 8.3144621;
const double GST = 0.07;
const double PI = 3.14159265;
```

- In these cases, we are defining **variables** whose values *cannot* be changed later in our program.

#define, const, variable

(1) Program 2.4:

```
area = 3.14159265*radius*radius;
```

(2) Program 2.5:

```
#define PI 3.14159265
area = PI*radius*radius;
```

(3) We can also write

```
const double PI = 3.14159265;
```

(4) Or even

```
double PI = 3.14159265;
area = PI*radius*radius;
```

What are the differences?

#define, const, variable

(1)

```
area = 3.14159265*radius*radius;
```

Direct use of the value of PI. A reader needs to be able to recognize the value as pi. If the same value occurs many times and we wish to change it to, say 3.1416, multiple changes must be made.

(2)

```
#define PI 3.14159265
area = PI*radius*radius;
```

PI is ***not*** a variable so it does not occupy any memory space. Use of a name (PI) makes the meaning clearer. If we wish to change the value and it occurs many times, only one change needs to be made:

```
#define PI 3.14156
```

#define, const, variable

(3)
```
const double PI = 3.14159265;
area = PI*radius*radius;
```

This defines a variable PI. It occupies 8 bytes of memory. The use of the keyword **const** means that this value of PI cannot be changed in statements later in the program.

(4)
```
double PI = 3.14159265;
area = PI*radius*radius;
```

In this case, PI is a normal variable, similar to Item (3) above except that its value can be modified later in the program.

A Review of C

3. Operators

Contents

- Operators in C
- Arithmetic Operators
- Assignment Operator
- Arithmetic Assignment Operators
- Increment and Decrement Operators
- Precedence and Associativity
- Type Promotion and Cast

Arithmetic Operators

- An ***operator*** is a symbol that instructs C to perform some action on one or more ***operands***.

- Arithmetic Operators: + - * / %

  2 + 3 = 5 **operands: 2, 3**

  + is a binary operator because it acts on 2 operands.

- **Unary minus operator (-): -a**

  "Unary" because there is only one operand.

The Modulus Operators

- The % operator is known as the **modulus** or **remainder** operator.

- If a and b are ***integers***, a%b gives the remainder when a is divided by b. For example,

  5%3 = 2

- **Example**:

  To check whether an integer n is even or odd, we can examine the value of n%2.

The Assignment Operator (=)

- **Syntax:** identifier = expression;
- An ***expression*** is a combination of variables, constants and operators. Every expression has a value.
- **Examples:** (The right hand sides are expressions)

```
answer = 'y';
radius = 5;
area = 3.14159*radius*radius;
value = cos(2.5);
sum = sum + number;
a = b = 10;
```

Additional operators

- Arithmetic assignment operators
- Increment and decrement operators
- Cast operator
- Relational operators
- Logical operators

Arithmetic Assignment Operators

- The statement

  ```
  a = a + b;
  ```

  can be written

  ```
  a += b;
  ```

- += is the addition assignment operator.
- We may also combine = with -, *, /, %

Arithmetic Assignment Operators

```
a += b;          a = a + b;
a -= b;          a = a - b;
a *= b;          a = a * b;
a /= b;          a = a / b;
a %= b;          a = a % b;
```

Increment and decrement operators (++, --)

- Adding or subtracting 1 to/from a variable is a very common operation in programming.

- All of the following increment **a** by 1:

```
    a = a+1;    a += 1;    a++;
```

- Similarly,

```
    a = a-1;    a -= 1;    a--;
```

x++, x--, ++x, --x

- x++ Post-increment
- x-- Post-decrement
- ++x Pre-increment
- --x Pre-decrement

Original Value of x	Expression	New Value of x	Value of Expression
1	x++	2	1
1	++x	2	2
1	x--	0	1
1	--x	0	0

Program

```
int a = b = 5;

printf("%d   %d\n", a--, --b);
printf("%d   %d\n", a--, --b);
printf("%d   %d\n", a--, --b);
printf("%d   %d\n", a--, --b);
printf("%d   %d\n", a--, --b);
```

OUTPUT

```
5    4
4    3
3    2
2    1
1    0
```

Precedence and Associativity of Operators

- **Precedence** is the order in which operations are performed.

a + b / c	Division before addition
(a + b) / c	Addition before division

- **Associativity** refers to left-to-right or right-to-left evaluations of expressions when two or more operators at the ***same*** precedence level are involved.

Precedence and Associativity Table
(Selected Operators Only)

Operator (Precedence: High to Low)	Associativity
()	from left
++ -- ! &(address of), *(dereference), cast, unary -	from right
* (multiplication) / (division) % (modulus)	from left
+ (addition), - (subtraction)	from left
< <= > >=	from left
== !=	from left
&&	from left
||	from left
?:	from right
Assignment: = += -= *= /= %=	from right

Example (left-associative)

Consider **15 % 6 * 2**

- **%** and ***** are at the **same precedence level**.
- Evaluation goes from left to right (left-associative).

- This is done first: **15 % 6 = 3,** then **3 * 2 = 6**

- It is clearer to write it as
- **(15 % 6) * 2**

Example (right-associative)

Consider `x = y += z -= 4`

- `=`, `+=` and `-=` are at the same precedence level.
- Evaluation goes from right to left.

```
    x = y += z -= 4
    x = y += (z -= 4)
    x = (y += (z -= 4))
    (x = (y += (z -= 4)))
```

Type Promotion

- Suppose we have two variables:

  `int num;     double radius;`

- and the ***mixed mode*** expression: **`num/radius`** is to be calculated.
- A ***copy*** of `num` will be ‘promoted’ to type double first before the division is carried out. `num` itself remains as an integer.

- The **Promotion order** is from left to right when mixed mode expressions are evaluated:
- **`char, int, long, float, double`**

Conversion by assignment

```
int num;    double radius;
```

- Suppose **`num = 5`** and we have **`radius = num;`**

  Result: `radius = 5.0`

- Suppose **`radius = 5.8`** and **`num = radius`**;

  Result: `num = 5`

- The decimal part has been truncated. Note that **`radius`** still remains as 5.8

Typecast

- A **typecast** uses the cast operator to explicitly control type conversions.

  (double)num ‘casts’ **num** to type double
  (float)num ‘casts’ **num** to type float.

  (int)radius ‘casts’ **radius** to type int.

- **Example**:

  Suppose **int x = 7, y = 5;**

  The integer division x/y results in 1 whereas

  (double)x/y results in 1.4

A Review of C

4. Standard Library Functions

Contents

- Functions in C (brief introduction)
- The C Standard Library
- Commonly Used Header Files
  - stdio.h
  - math.h
  - ctype.h
- Examples of Standard Library Functions

Functions in C

- **Functions** are the *building blocks* of C programs, e.g. **`printf(), scanf(), cos().`**
- Functions maybe pre-written (e.g. printf()) or user written (See Chapter 7).
- A function has a name, e.g. cos, printf.
- The name is followed by a pair of brackets, e.g. cos().
- A function is **called** or **invoked** by typing its name, e.g. to output something to the screen, we type

```
    printf(" . . . ");
```

Functions in C

- Most functions require some given values in order to work. For example, to calculate cos(2), we must provide the value 2. Such a value that is passed (or given) to a function is known as an **argument (parameter)**. So cos(x) needs one argument or parameter. Some functions need more than one parameter, e.g. the power function pow(x,y) requires two parameters.

- Many functions **return** values to the calling function, e.g. cos(x) returns a value to its **caller** after calculation.

The C Standard Library

- The C language provides a rich collection of powerful functions known as the **C Standard Library**, ready to be used.

- The library has a set of associated header files such as **stdio.h**. These contain
  - ***prototypes*** (to be studied later) of functions in the library
  - ***macro*** definitions
  - other programming elements

- To use a function (e.g. **printf( )**), we need to include the corresponding header file.

Selected Header Files

Header File	Description/Examples
stdio.h	Input/Output: printf, scanf
math.h	sqrt, pow, sin, cos, tan, exp, log, log10, asin
ctype.h	Character handling: isalpha, isdigit, toupper
stdlib.h	abs, rand
string.h	String handling: strlen, strcat

stdio.h

- **`printf`**
- **`putchar`** Displays one char on screen
- **`scanf`**
- **`getchar`** Gets one char from input stream
- **`fflush`** Flushes I/O buffer
- **. . .**

Input Stream & Input Buffer

- The characters you type at the keyboard form an **input stream**.

- In many cases, this stream of characters first goes into an area of memory known as the **input buffer** before being used in a running program.

getchar()

- **getchar()** reads the next character from the **input stream** and returns its value as an integer.
- It does not need a parameter/argument and this is indicated by the keyword **void**.
  - syntax: **int getchar(void);**
  - usage: **ch = getchar();**
  - **or** **getchar();**

- The computer will pause and wait for user input.
- getchar() will only start to read a character after the ENTER key is pressed.

math.h

- Trigonometric and hyperbolic functions, such as cos(), acos(), and cosh().
- Exponential and logarithmic functions, such as exp(), pow(), and log().
- Miscellaneous math functions, such as sqrt(), fabs(), fmod().

- The mathematical functions all return data as type **double**.

math.h -- sqrt()

- Syntax: double sqrt(double x);
- It takes a *double precision* argument and returns a value of type *double*.

- How it is used:
  - ✓ y = sqrt(2);
  - ✓ y = sqrt(2.0);

- Note that we may type the integer 2 instead of 2.0 as argument.

Program 4.1

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

int main()
{
   float f = 3.14159265;
   double d = 3.14159265;

   printf("sqrt of %.8f is %.15f using float\n", f, sqrt(f));
   printf("sqrt of %.8f is %.15f using double\n", d, sqrt(d));

   return 0;
}
```

```
✓ sqrt of 3.14159274 is 1.772453875567027 using float
✓ sqrt of 3.14159265 is 1.772453849892854 using double
```

math.h –- pow()

- Syntax: double pow(double x, double y);

- Calculates x raised to power y.

```
double x = 3.14159265, y = 2.5;
z = pow(x,y);
```

- **Result:**

```
float:      z = 17.493419647216797
double:     z = 17.493418277652001
```

Trigonometric Functions

- **`sin(x), cos(x), tan(x)`**
- Note that x must be in radians.
- Other functions:
  - **`acos(x), asin(x), atan(x)`**
  - **`cosh(x), sinh(x), tanh(x)`**

Logarithms and Exponential Function

- exp(x) Exponential function
- log(x) Natural logarithm
- log10(x) Logarithm to the base 10

ctype.h

- Some **character** handling functions:

• **isalnum**	Alphanumeric character?
• **isalpha**	Alphabetic character?
• **isdigit**	Decimal digit?
• **islower**	Lower case character?
• **isupper**	Upper case character?
• **tolower**	Upper case --> lower case
• **toupper**	Lower case --> upper case

isdigit

```
#include <ctype.h>
char ch='2';
if (isdigit(ch) == 0)
   printf("%c is not a digit.", ch);
else
   printf("%c is a digit.", ch);
```

- **Note:**
- **isdigit** returns TRUE (not 0) or FALSE (0).

toupper()

```
#include <ctype.h>
char ch1='a', ch2 = 'B';

ch1 = toupper(ch1);
  /* Converts ch1 to 'A' */

ch2 = toupper(ch2);
  /* No effect on ch2  */

ch2 = tolower(ch2);
  /* Converts ch2 to 'b' */
```

stdlib.h

- We shall only consider the 4 functions:
  abs(), system(), rand() and **srand()**

- **abs(x)** returns the absolute value of an integer **x**.

- **system()** enables us to execute operating system commands.

- Example:

  ```
  system("Pause");
  ```

  causes program execution to pause until we press any key.

A Review of C

5. Decision Making

Contents

- The 3 basic Programming Structures
- Relational Operators
- Logical Operators
- if-else, Nested if
- switch and break

Three Basic Program Structures

1. **Sequence**

   Statements are executed line by line in sequential order.

2. **Selection**

   Flow of program execution may be changed based on some conditions.

3. **Repetition**

   A group of statements is repeated a number of times.

Selection (Decision Making)

- Statements normally execute from top to bottom, in the same order as they appear in the source code.
- Selection and Repetition modify the order of statement execution. They are known as **program control statements**.
- An **if** statement is a type of selection, and it has the form:

```
if (condition)
{
        . . .
}
```

Relational Operators in C

- To form the *condition*, we need relational operators:

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

True and False

- An expression containing a relational operator has a value 1 (TRUE) or 0 (FALSE).

- Do not confuse == with =

  a==b Checks whether values of a and b are equal.

  a=b Value of b is assigned to a.

- Suppose a = 1, b = 5
- `printf("The value of a==b is %d", a==b);`
- `printf("The value of a=b is %d", a=b);`

Non-Zero Value for True

- In C, any non-zero numeric value can also represent TRUE.
- **Example**:

```
    a = 5;
    if (a) printf("%d", a);
```

- As a is not zero (meaning TRUE), printf() will be executed.
- If a is 0 (meaning FALSE), printf() will not be executed.

Non-Zero Value for True

- Suppose we write the statement as

```
if (a = 5) printf("%d", a);
```

- **Interpretation**:

  Value of ***expression*** **a = 5** is 5.

  Condition is TRUE.

Comparing Characters

- Characters can be compared using relational operators. Basis for comparison?
- **ASCII values**.
- '9' >= '0' 1 (TRUE)
- (57) (48)
- 'B' <= 'A' 0 (FALSE)
- 'A' <= 'a' 1 (TRUE)
- (65) (97)
- 'a' <= '$' 0 (FALSE)

Logical Operators

- Logical operator **AND, OR, NOT**
- Symbols used in C: &&, ||, !

- These are needed to form complex conditions. They increase the decision-making capabilities of our C programs.

- && and || are ***binary*** operators because they act on 2 expressions. ! is ***unary***.

- Check the precedence/associativity of these operators.

Truth Values and Truth Table

The truth value (True or False) of a relational expression (**P** and **Q** below) is determined by a Truth Table.

P	Q	P || Q	P && Q	!P
F	F	F	F	T
F	T	T	F	T
T	F	T	F	F
T	T	T	T	F

Logical Assignment

- We may even assign logical expressions to variables which will then evaluate to 0 or 1.

- Suppose the variables are all integers:

```
pass_status = (mark >= 50);
in_range = (n > -5 && n < 5);
even = (n%2 == 0);
```

if-else

```
if (expression){
   statement_1;
          . . . . . . .
}
else {


          . . . . . . .
}
```

NOTE: The else is optional. Braces are not needed if there is only one statement following the **`if`** or the **`else`**.

Program 5.1 (if without else)

```
printf("Input an integer value for x: ");
scanf("%d", &x);
printf("Input an integer value for y: ");
scanf("%d", &y);

/* Test values and print result */
if (x == y)
   printf("x is equal to y");
if (x != y)
   printf("x is not equal to y");
```

NOTE: We usually do not write *if* this way.

Program 5.2 (if else)

```
printf("Input an integer value for x: ");
scanf("%d", &x);
printf("Input an integer value for y: ");
scanf("%d", &y);

/* Test values and print result */
if (x == y)
    printf("x is equal to y");
else
    printf("x is not equal to y");
```

Program 5.3 (3 cases)

```
printf("Input an integer value for x: ");
scanf("%d", &x);
printf("Input an integer value for y: ");
scanf("%d", &y);

/* Test values and print result */
if (x == y)
   printf("x is equal to y");

if (x > y)
   printf("x is greater than y");

if (x < y)
   printf("x is smaller than y");
```

NOTE: We usually do not write *if* this way.

Program 5.4 (3 cases with else)

```
printf("Input an integer value for x: ");
scanf("%d", &x);
printf("Input an integer value for y: ");
scanf("%d", &y);

/* Test values and print result */

if (x == y)
    printf("x is equal to y");
else
   if (x > y)
       printf("x is greater than y");
   else
       printf("x is smaller than y");
```

Nested if statements

- Program 5.4 contains the following structure:

```
if(condition 1)
     statement1;
else if(condition 2)
     statement2;
else if(condition 3)
     statement3;
else
     statement4;
```

- This is known as a **nested if** structure: an if statement containing other if statements.

Nested if statements

- We may have nested-if structures such as the following:

```
if(condition 1){
   statement1;
   if(condition 2)
        statement2;
   else
        statement3;
}
else
        statement4;
```

- Note: An `else` always matches the nearest `if` before it.

A Review of C

6. Loops

Contents

- The 3 loop/iteration structures:
  - while
  - do while
  - for
- Nested Loops
- break
- continue

What is a loop?

- Other names: **Repetition**, **Iteration**.
- A group of statements is repeatedly performed a certain number of times, or until some condition becomes false.
- Computers are very good in doing repetitive work, fast and accurately.
- 3 iteration structures
  - **while**
  - **do while**
  - **for**

while

```
while (loop repetition condition)
{
        statement;
        . . . . .
        statement;
}
```

- The statement block will be executed as long as the loop repetition condition is ***true***. The braces are not necessary if there is only one statement in the loop.

Infinite loop

- It is important to make sure that the loop repetition condition can become ***false*** inside the while structure.
- If the condition is always true, we’ll get an **infinite loop** (or **endless loop**).

```
while (1)
{
   printf("How do I stop this endless
                 loop?\n");
}
```

Program 6.1

```
// Displays the digits 0 to 9
// This loop is a counter-controlled loop
int digit = 0;
while (digit <= 9)
{
   printf("%d ", digit);
   digit++;
}
```

Display:

```
0 1 2 3 4 5 6 7 8 9
```

Program 6.2

```
/* Pseudocode example in Chapter 1 */
int n, counter = 1;
double number, sum=0, average;
printf("How many numbers are there?");
scanf("%d", &n);
while (counter <= n)
{
   scanf("%lf", &number);
   sum += number;
   counter++;
}
average = sum/n;
```

Program 6.3

```
int quantity;
double price;

// Sentinel value is 0
printf("Enter 0 to quit.\n");
printf("Enter price & quantity: ");
scanf("%lf%d", &price, &quantity);

while (price != 0)
{  printf("Total = $%6.2f.\n", price*quantity);
   printf("Enter price & quantity: ");
   scanf("%lf%d", &price, &quantity);
}
```

do - while

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

- Checking of the value of the expression is done at the end.
- Body of loop is executed at least once. This is also known as a **posttest loop** whereas the **while** loop is called a **pretest loop**.

Program 6.4

Program 6.1 can be rewritten as follows:

```
/* Displays the digits 0 to 9 */

do
{
   printf("%d ", digit++);
} while (digit <= 9);
```

Program 6.5 (Input Validation)

```
// To make sure that user-input is < 5
#include <stdio.h>
int main()
{  int input;     /* User-input number */
   do
   {  printf("Input an integer less than 5: ");
      scanf("%d", &input);

      if (input >= 5)
        printf("Value is too large. Try again.\n");
    } while (input >= 5);

    printf("Your input is correct. Thank you!\n");
    return 0;
}
```

The for loop

```
for (initialization; test; counter)
{
    statements;
}
```

- The braces are not necessary if there is only one statement in the loop body.
- The **for** loop is often used as a counter-controlled loop though it is more general than that.

Program 6.6

```c
int digit;

for (digit = 0; digit <= 9; digit++)
   printf("%d ", digit);
```

```c
// Program 6.1
int digit = 0;
while (digit <= 9)
{
   printf("%d ", digit);
   digit++;
}
```

Nested Loops

- A loop within another loop.
- No overlap allowed: one loop must be completely embedded inside the other.
- Each loop must be controlled by a different control variable or counter.

Program 6.7

```
int outer, inner;
for (outer=1; outer <= 3; outer++)
{  printf("outer = %d  ", outer);
   for (inner=1; inner <= 4; inner++)
      printf("inner = %d  ", inner);
      printf("\n");
}
```

The break statement

- The **break** statement causes an exit from the innermost loop or from a switch statement.

```
while (1)    // Not recommended
{
  printf("Enter number, negative to quit. ");
  scanf("%f", &x);
  if (x < 0.0) break;  // exit loop if x < 0
  printf("%f\n", sqrt(x));
}
/* break causes exit from loop and program execution
jumps here. */
```

The continue statement

- Causes the current iteration of a loop (for, while, do) to stop and the next iteration to begin.

Program 6.8

```
/* To exclude invalid marks (<0 or >100) */
int count = 1, stud_num = 25, mark, sum = 0;
while (count <= stud_num)
{  printf("Enter a mark => ");
   scanf("%d", &mark);
   if (mark <0 || mark > 100){
     printf("Invalid mark. Type again.\n");
     continue; //Discards invalid mark
   }
   sum += mark;
   count++;
}
```

A Review of C

7. Functions

Contents

- Programmer-defined functions
  - function header and function prototype
  - return value
  - parameter passing

- Storage Classes
  - scope and lifetime
  - local and global variables

Modular Programming

- Large programs should be subdivided into small modules, each performing a specific, small task.
- This makes debugging, maintenance and addition of new modules easier.
- C is a modular language as C programs are made up of functions (modules).

Functions

- A function in C is a named block of statements that performs a specific task.
- A typical C program contains many functions.
- The **`main()`**function is the first function to be executed.
- The **`main()`** function generally contains many calls to other functions.
- Functions are called by using its name, e.g.

```
y = cos(x);
```

Functions

- Some functions may also call other functions.
- These functions may come from the standard library or are written by the programmer (known as **programmer-defined functions** or **user-defined functions**).
- A function may need additional data from the calling function for it to work (i.e. passing parameters).

  e.g. `y = pow(2.5, 5.6);`
- A function may return a value to its caller.
- A function has to be declared before it is used (**function prototype**).

Structure of a Function

```
Return_type name (parameter list)
{
        declarations
        statements
}
```

- Function header

- Function body is within {   }

Header and Body

- The **function header** contains the following:
  - return data type
    - o **char, int, float, double**

      e.g. **double sqrt(double x)**
    - o **void** if no data is returned.

      e.g. **void print_heading(. . .)**
  - function name
  - parameter list with data type

- The **function body** is a ***block*** of statements enclosed in {}.

Parameter List

- If a function does not receive any data from the calling function, the parameter list contains only the keyword **`void`**.

- When data is to be received, the parameter list contains the variable names (with data types) that pass the data to the function,
- e.g. **`double pow(double x, double y)`**

Passing Parameters to a Function

- The parameters specified in a function header are also referred to as **formal parameters**.

- The arguments specified within a ***function call*** are also known as **actual parameters**.

- Some authors use **parameters** (for formal parameters) and **arguments** (for actual parameters).

- When a function call is made, the values of the actual parameters are copied into the formal parameters—this is known as **passing parameters by value**.

Passing Parameters to a Function

- It is important to note that the function only receives a **copy** of the actual parameter.
- Any changes to this copy inside the called function will not affect the original value of the parameter in the calling function.
- Another way of passing data is passing **parameters by reference (a reference/pointer to a variable is its address)**.

Program 7.1 (void)

```
/* A void return data type with a void
parameter list */

void print_heading(void)
{
    printf("Computing Course Report\n");
    printf("=======================\n");
}
```

Program 7.2a

```
/* A void return data type with one
parameter */

void print_line(int numOfLines)
{
  int i;
  for (i=1; i<=numOfLines; i++)
    printf("Line number %d\n", i);
}
```

Program 7.2b

```
/* main() calling print_line() */
#include <stdio.h>
void print_line(int);      /* Function protoype -- will be
                            explained later */
int main()
{  int i = 5;
   print_line(i);          // calls print_line
   print_line(5);          // same effect as previous line
   return 0;
}

void print_line(int numOfLines)  /* num_lines is a local
                                 variable – to be explained */
{
   int i;
   for (i=1; i<=numOfLines; i++)
      printf("Line %d\n", i);
}
```

Function Prototypes

- ANSI C requires that each function used must be declared using a function prototype, usually placed at the beginning of the program before main().

- A function prototype is very similar to a function header:

  return_type name (parameter list);

- With 2 differences:
  - It ends in a semicolon.
  - The parameter list must list the data types of the variables but the names are optional.

- Note that the order and data types of the variables must agree with those in the function header.

Returning a Value from a Function

- A function may return a value to its caller.
- In the next example, the function **`cube()`** returns the value of the cube of an integer to main().
- In such a case, there must be at least one **`return`** statement in the function definition.
- After such a **`return`** statement is executed, the function terminates.

Program 7.3a (int return type)

```
/* Passing parameter by value */
#include <stdio.h>
int cube(int x);  /* name x is optional */

int main()
{  int x = 2, y;

   y = cube(x);  /* Calls function cube()*/
   printf("y = %d.\n", y);
   printf("Calls cube without assignment. cube(%d) =
            %d\n", x, cube(x));

   return 0;
}
```

Program 7.3a

```
/* Return type is int with one parameter */
int cube(int x)
{  int result;
   result = x*x*x;
   return result;
}
```

```
/* A shorter way of defining cube() */
int cube(int x)
{
   return x*x*x;
}
```

Program 7.3b

```
#include <stdio.h>
int cube(int);
int main()
{  int x = 2, y;
   y = cube(x);
   printf("y = %d in main.\n", y);
   printf("x = %d in main.\n", x);
   return 0;
}
```

```
/* The memory location of x in cube is different from that of
x in main(). */
int cube(int x)
{  x = x*x*x;
   printf("Value of x returned by cube is %d. \n", x);
   return x;
}
```

Program 7.3c

```c
#include <stdio.h>
int cube(int);
int main()
{  int x = 2, y;
   y = cube(x);
   printf("y = %d in main.\n", y);
   printf("x = %d in main.\n", x);
   return 0;
}
```

```c
/* The y defined here is local to cube and has nothing to do
with the y in main(). We could have used other identifiers.
*/
int cube(int y)
{
   y = y*y*y;
   return y;
}
```

Program 7.4 (many parameters)

```
#include <stdio.h>
#include <math.h>
double length(double, double, double, double);

int main()
{  double Ax, Ay, Bx, By;

   printf("Input the coordinates of point A: ");
   scanf("%lf%lf", &Ax, &Ay);
   printf("Input the coordinates of point B: ");
   scanf("%lf%lf", &Bx, &By);
   printf("\nThe distance between A and B is %.2f\n",
            length(Ax, Ay, Bx, By) );
   return 0;
}
```

Program 7.4 (many parameters)

```
double length(double x1,double y1,double x2,double y2)
{
   return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
```

Storage Class of a Variable

- The **Storage Class** of a variable defines its scope and lifetime.
- The **scope** of a variable is the extent within a program to which the variable is available/visible.
- A variable that is defined and available only within a ***function*** is referred to as a **local variable**.
- A variable that is available to a function but defined ***outside*** the function is known as a **global variable**.
- In C, we can control the scope of a variable. It may be restricted to a block, one function, a group of functions, or the entire program.

Local Variables

- Local variables are variables defined inside functions.

- Two or more functions can have local variables with the ***same*** names. Such identically named variables are totally independent of each other. (See Program 7.3b).

- A local variable has a ***limited lifespan***. It is created when the function in which it is declared is called. After the function terminates, the variable no longer exists.

Local Variables

```
/* Program 7.3 */
int cube(int x)  /* x is a local variable */
{
   x = x*x*x;
   printf("Value of x returned by cube() is %d. \n", x);
   return x;
}
```

Global Variables

- Any variable defined ***outside*** a function is known as a **global variable**.
- Such a variable is available to any function that ***follows*** its definition within the same source file if the function does not declare a local variable with the same name.
- Such variables are destroyed only when the *program* terminates.
- Global variables are initialized to 0 by default by the compiler (danger!).

Program 7.5a

```c
#include <stdio.h>
void funct(void);
int a = 1, b = 2, c = 3;  // Global variables

int main()
{
   printf("Before: a = %d, b = %d, c = %d\n", a, b, c);
   funct();
   printf("After: a = %d, b = %d, c = %d\n", a, b, c);
   return 0;
}

void funct(void)
{
   printf("Inside: a = %d, b = %d, c = %d\n", a, b, c);
}
```

Output

```
•Before: a = 1, b = 2, c = 3
•Inside: a = 1, b = 2, c = 3
•After: a = 1, b = 2, c = 3
```

Program 7.5b

```
void funct(void)
{
   int a = 100;
   printf("Inside: a = %d, b = %d, c = %d\n", a,
b, c);
}
```

Output:

```
Before: a = 1, b = 2, c = 3
Inside: a = 100, b = 2, c = 3
After: a = 1, b = 2, c = 3
```

Program 7.5c

```
void funct(void)
{
   a = 100;
   printf("Inside: a = %d, b = %d, c = %d\n", a, b, c);
}
```

Output:

```
Before:     a = 1, b = 2, c = 3
Inside:     a = 100, b = 2, c = 3
After:      a = 100, b = 2, c = 3
```

Program 7.6a

```
int a = 1, b = 2, c = 3;  /* Global variables */
int main()
{  funct1();
   funct2();
   return 0;
}

void funct1(void)
{  a = 100;   /* This changes global variable a */
   . . .
}

void funct2(void)
{  int result;
   result = a*b*c; /*Suppose funct2 assumes that a=1.
                    Value for result will be wrong. */
}
```

Program 7.6b

```
int b = 2, c = 3;  // Global variables
int main()
{  funct1();
   funct2();
   return 0;
}

void funct1(void)
{  int a = 100; . . .}

int a = 1;  // This a is global to funct2 only

void funct2(void)
{  int result;
   result = a*b*c;
}
```

Local & Global Variables

- **Advantage of local variables:**

  They cannot be changed by other functions. This is very important, especially in larger programs.

- **Advantage of global variables:**

  Easy to communicate data between functions. If two or more functions must share or communicate a large number of variables or data values, then this eliminates the need for long function argument lists.

- **Disadvantage of global variables:**

  Values can be changed by any function. This may lead to hard-to-find bugs. **Global variables should not be used whenever possible**.

Returning More Than One Data Item

- A C function can only return one data item to its caller.

- If we need to return more than one item, a different mechanism is needed.

- This is done through the concept of a **pointer** or the related concept of an **array** (see next chapter).

What is a Pointer?

- A pointer is a variable that represents the location or address (rather than the value) of a data item, such as a variable.

- Pointers are used frequently in C, as they have a number of useful applications, including returning more than one value to the calling function.

A Review of C

8. Arrays

Contents

- Concept of an Array
- Declaration and Initialization
- Input and Output
- Passing Arrays to Functions

Arrays

- Many applications require the processing of multiple data items that have common characteristics.
- Example: a large set of numerical data such as exam marks for 1000 students.
- It is often convenient to place the data items into an **array**, where they will all share the ***same name*** (e.g., exam_mark).
- The individual data items can be characters, integers, floating-point numbers, etc.
- However, they must all be of the *same type* and have the *same storage class*.

Arrays

- Each array element is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets,

  e.g. **`exam_mark[i], i=0,1,2,…`**

- Each subscript must be a non-negative integer. In an n-element array **`x`**, the array elements are

  `x[0], x[1], x[2],…, x[n–1].`

  Note that the subscript starts from 0.

- The value of each subscript can be an integer constant, an integer variable, or a more complex integer expression.

Arrays

- The number of subscripts determines the **dimensionality** of the array.
- For example,
  - **x[i]** refers to an element in the one-dimensional array **x**.
  - Similarly, **y[i][j]** refers to an element in the two-dimensional array **y**. (We can think of a two-dimensional array as a table.)

- Higher-dimensional arrays can also be formed, e.g., **density[100][50][75]**.

Declaring an Array

- General format:

```
    data-type array_name[integer];
```

- **Examples:**

```
     double exam_mark[1000];
     char text[80];
     float n[12];
```

- A subscripted variable can be used just like other variables,
- Example

```
      for (i=0; i<1000; i++)
         exam_mark[i] = exam_mark[i] + 5;
```

Program 8.1

```
/* We frequently use a for loop to deal with values of
array elements */

double exam_mark[1000];

for (i=0; i < 1000; i++)
{
   printf("Enter mark[%d]: ", i+1);
   scanf("%lf", &exam_mark[i]);
}
/* Outputting this array is similar: use a for loop */
```

Program 8.2

```
/* To calculate the sum and average of the array
elements */

double exam_mark[1000], sum = 0, average;
for (i=0; i < 1000; i++)
{  printf("Enter mark[%d]: ", i+1);
   scanf("%lf", &exam_mark[i]);
}
for (i=0; i < 1000; i++)
{  sum += exam_mark[i];
}
average = sum/1000;
```

Program 8.3

```
/* Converts a line of text with 10 characters to
uppercase */
#include <stdio.h>
#include <ctype.h>
#define SIZE 10
int main()
{  char letter[SIZE];
   int count;
   /* read in the line */
   for (count = 0; count < SIZE; count++)
      letter[count] = getchar();

   /* display the line in upper case */
   for (count = 0; count < SIZE; count++)
      putchar(toupper(letter[count]));

   return 0;
}
```

Array Initialization

- Array elements can be initialized:

```
int number[10] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
char colour[3] = {'R', 'G', 'B'};
```

- Suppose we write

```
int number[10] ={1, 2, 3};
```

- This means

```
number[0] = 1,  number[1] = 2, number[2] = 3
```

- *All other elements are 0.*

Array Initialization

- The array size need not be specified explicitly when initial values are included as a part of an array definition. For a numerical array:

```
int number[] = {1, 2, 3, 4, 5, 6};
```

- This is the same as

```
int number[6] = {1, 2, 3, 4, 5, 6};
```

Processing Arrays

- If a[] and b[] are similar arrays (i.e., same data type, same dimensionality and same size), assignment operations, comparison operations, etc. must be carried out on an ***element-by-element*** basis.

- This is usually accomplished within a loop, where each pass through the loop is used to process one array element.

- Single operations such as a=b which involve entire arrays are **not** permitted in C.

Passing Arrays To Functions

- An entire array can be passed to a function as an argument.
- However, the way it is done is different from that for an ordinary variable.
- To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an argument within the function call.
- The corresponding formal parameter is written in the same manner, though it must be declared as an array within the formal parameter declarations.

Passing Arrays To Functions

- When declaring a 1-D array as a formal parameter, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal parameter declaration.

Function Prototypes with Arrays

- In function prototypes that include array parameters, an empty pair of square brackets must follow the name of each array parameter.
- If parameter names are not included in a function declaration, then an empty pair of square brackets must follow the array parameter data type.

Example 8.4

- This example uses a function **fn_average()** to calculate the average of a list of numbers which are input as an array **num_array[1000]** in main().

- The array is passed to **fn_average()** where the actual calculation of the average is done. The main aim of this example is to show how an array is passed to a function.

- The result is then passed back as a double to main() for further processing.

Example 8.4

```
double fn_average(int, double []);  // prototype

int main()
{  int num=100; //How many numbers to average?
   double average, num_array[1000];
   . . .        //Get numbers from user
   average = fn_average(num, num_array);

   . . .
}

double fn_average(int size, double x[])
{ . . .
}
```

Pass by Reference

- When an array is passed to a function, the values of the array elements are **not** passed to the function. Rather, the array name is interpreted as the address of the first array element (i.e., the address of the memory location containing the first array element).

- This address is assigned to the corresponding formal parameter when the function is called. Arguments that are passed this way are said to be **passed by reference** rather than **by value**.

- If an array element is altered *within* the function, the original array in the calling function is directly altered.

Program 8.5

```
// Effect of changing array elements in function
#include <stdio.h>
void modify(int a[]);
int main()
{
   int count, a[3]={101, 102, 103};

   modify(a); /* Array elements are changed to -9 in function
                    modify(). This affects array a[] here. */

   printf("From main, after calling the function:\n");
   for (count = 0; count <=2; ++count)
      printf("a[%d] = %d\n", count, a[count]);
}
```

Program 8.5

```
void modify(int a[])
{  int count;
   printf("From function, after modifying the values:\n");
   for (count = 0; count <= 2; ++count) {
       a[count] = -9;
       printf("a[%d] = %d\n", count, a[count]);
}
```

A Review of C

THE END