Lecture 2

Variables & Introduction to Problem Solving

Dr. Mohammad Ahmad

Variables

- **A *variable* is a name for a location in memory**
- **A variable must be *declared* by specifying the variable's name and the type of information that it will hold**

data type

variable name

```
int total;

int count, temp, result;
```

Multiple variables can be created in one declaration

Rules for valid variable names

- **The name can be made up of letters, digits, the underscore character (_), and the dollar sign**
- **Variable names cannot begin with a digit**
- **C is *case sensitive* - `Total, total, and TOTAL` are different identifiers**
- **By convention, programmers use different case styles for different types of names/identifiers, such as**
    - ***title case* for variable names - `Lincoln`**
    - ***upper case* for constants - `MAXIMUM`**

Variable Initialization

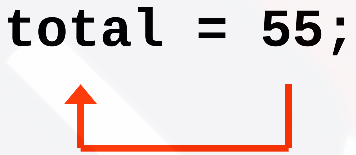
- A variable can be given an initial value in the declaration

```
int sum = 0;
int base = 32, max = 149;
```

- When a variable is referenced in a program, its current value is used

Assignment

- An *assignment statement* changes the value of a variable
- The assignment operator is the = sign



```
total = 55;
```

- The expression on the right is evaluated and the result is stored in the variable on the left
- The value that was in `total` is overwritten
- You can only assign a value to a variable that is consistent with the variable's declared type

Assignment Through scanf()

int variable;

scanf(“%d”, &variable);

- <keyboardinput> 30

- There is not assignment operator in this case

Constants

- **A constant is an identifier that is similar to a variable except that it holds the same value during its entire existence**

- **As the name implies, it is constant, not variable**

- **The compiler will issue an error if you try to change the value of a constant**

- **In C, we use the `const` modifier to declare a constant**

```
const int MIN_HEIGHT = 69;
```

Constants

- Constants are useful for three important reasons
- First, they give meaning to otherwise unclear literal values
  - For example, MAX_LOAD means more than the literal 250
- Second, they facilitate program maintenance
  - If a constant is used in multiple places, its value need only be updated in one place
- Third, they formally establish that a value should not change, avoiding inadvertent errors by other programmers

#define primitive

- **Constants can also be defined using the primitives of the C preprocessor**

- **#define KMS_PER_MILE 1.609**

Some Primitive Data Types

- **int**
- **float**
- **double**

float and double analogy

float and double analogy

Numeric Primitive Data

- **The difference between the various numeric primitive types is their size, and therefore the values they can store:**

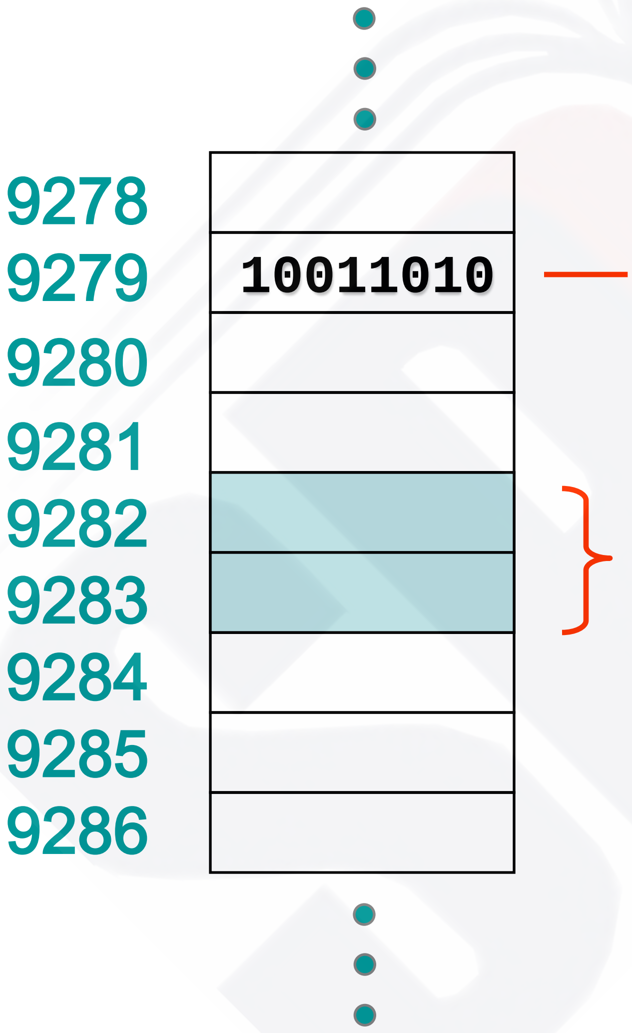
Type	Storage	Min Value	Max Value
`char`	8 bits	-128	127
`short`	16 bits	-32,768	32,767
`int`	32 bits	-2,147,483,648	2,147,483,647
`long`	64 bits	$< -9 \times 10^{18}$	$> 9 \times 10^{18}$
`float`	32 bits	$+/- 3.4 \times 10^{38}$ with 7 significant digits	
`double`	64 bits	$+/- 1.7 \times 10^{308}$ with 15 significant digits	

Computer Memory

Main memory is divided into many memory locations (or *cells*)

Each memory cell has a numeric *address*, which uniquely identifies it

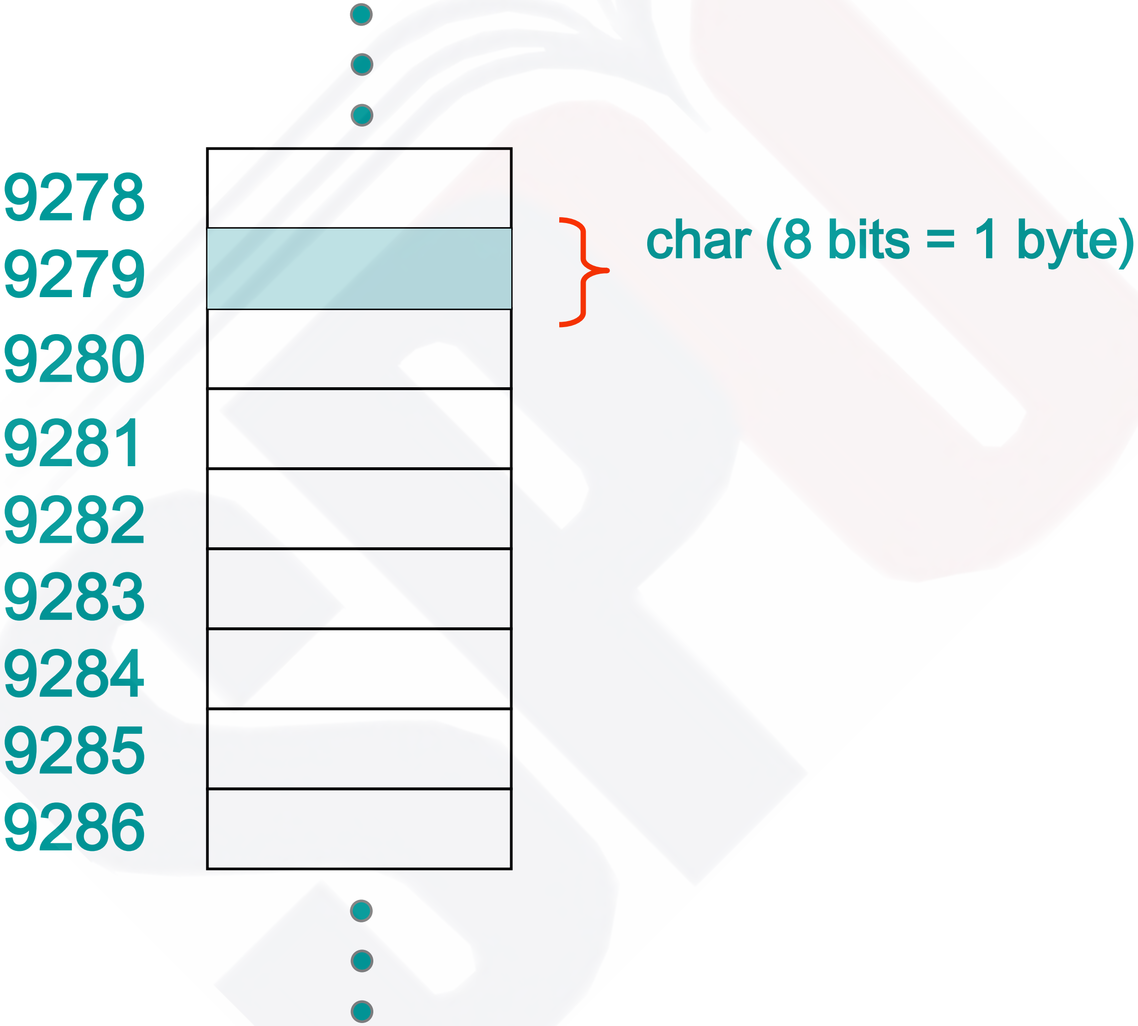
Storing Information



Each memory cell stores a set number of bits (usually 8 bits, or one *byte*)

Large values are stored in consecutive memory locations

Storing a char



9278

9279

9280

9281

9282

9283

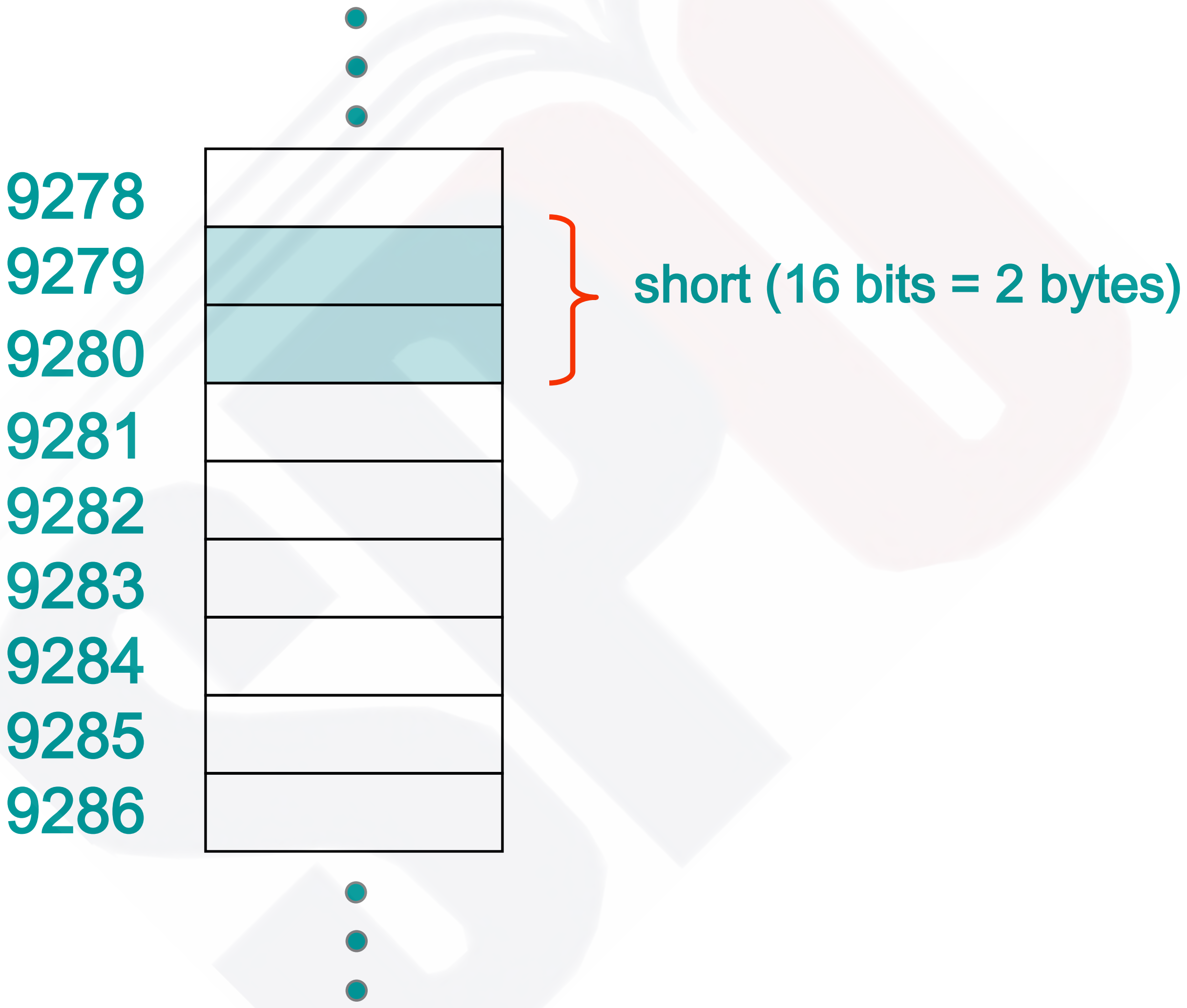
9284

9285

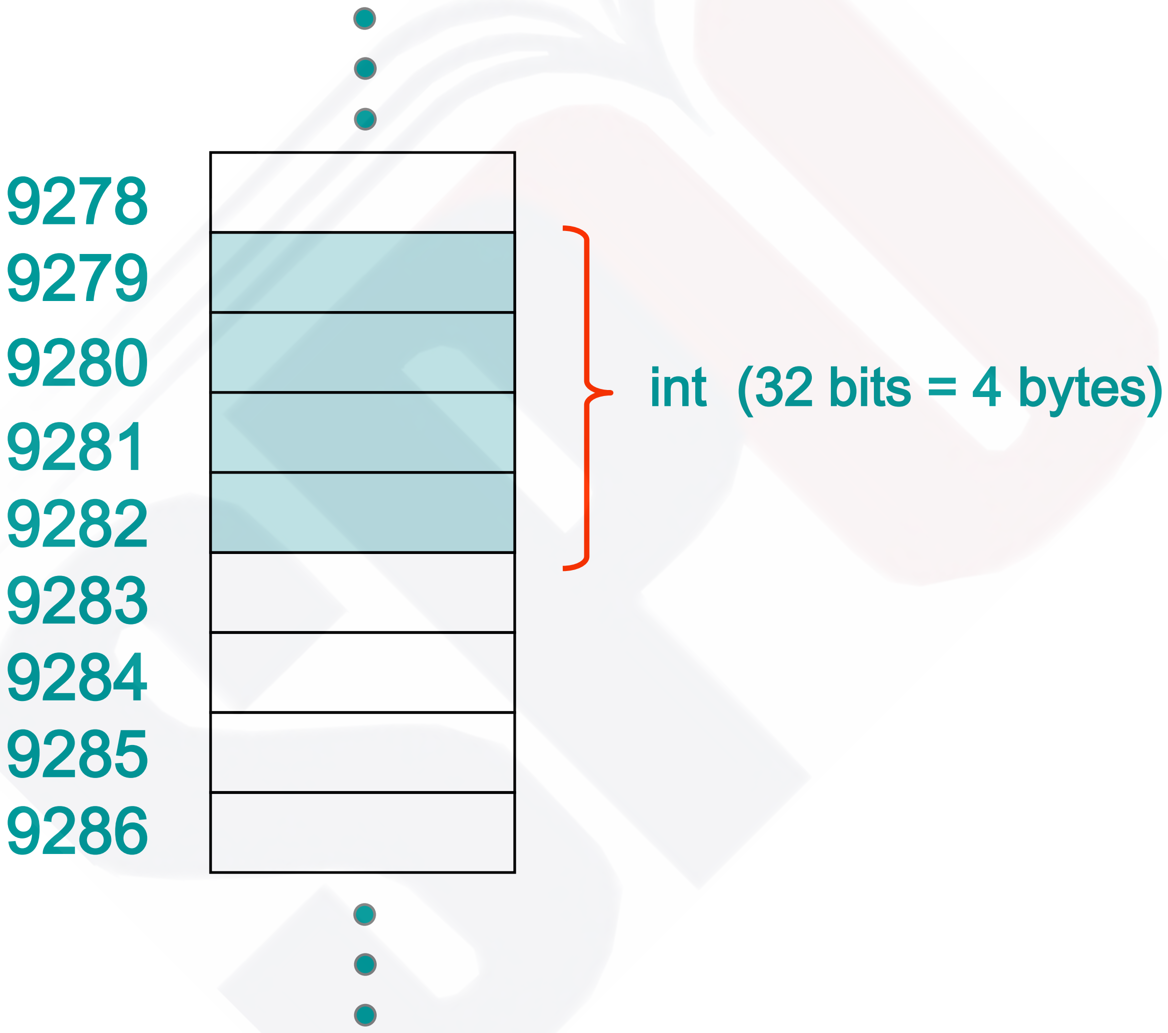
9286

char (8 bits = 1 byte)

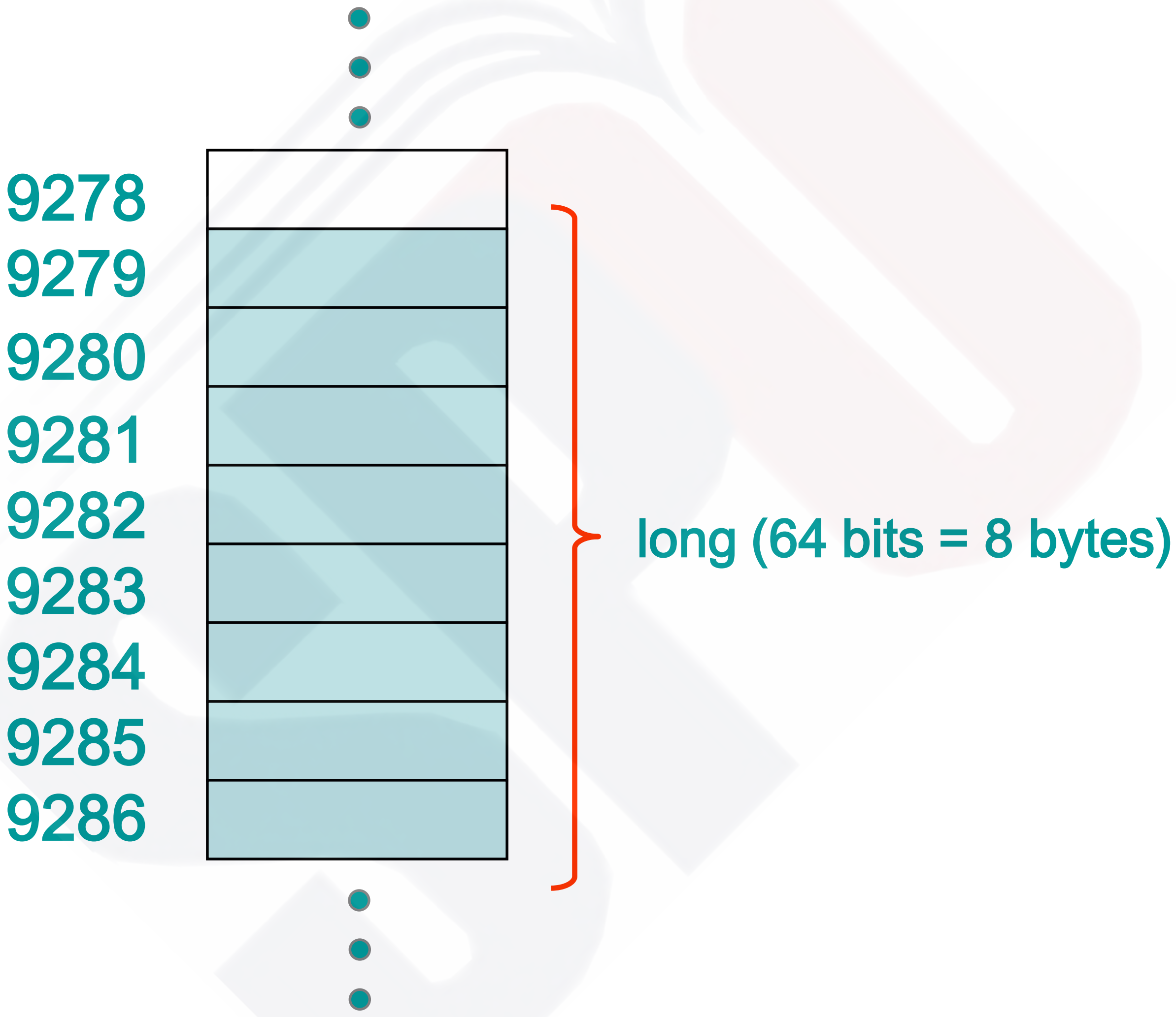
Storing a short



Storing an int



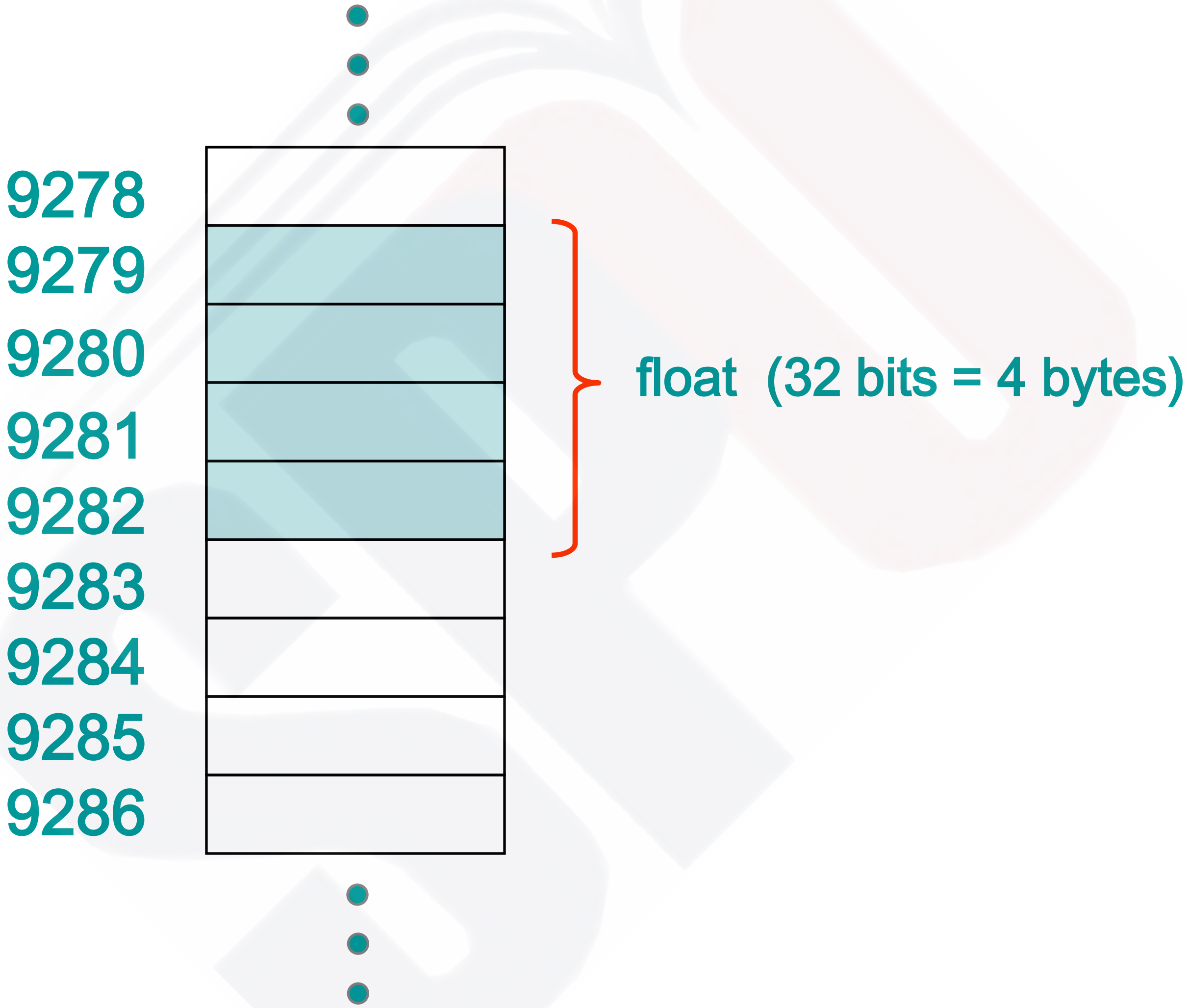
Storing a long



9278
9279
9280
9281
9282
9283
9284
9285
9286

long (64 bits = 8 bytes)

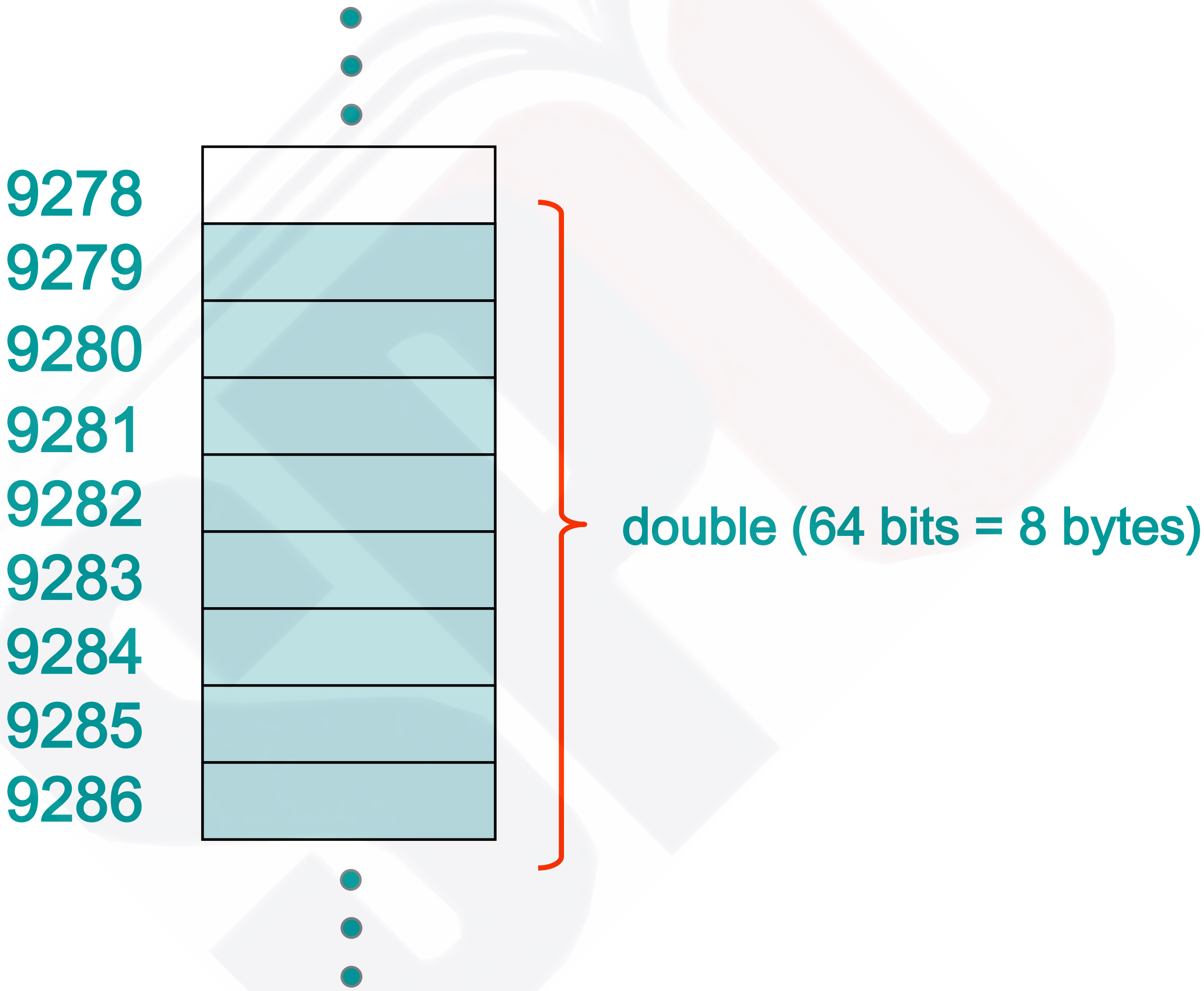
Storing a float



9278
9279
9280
9281
9282
9283
9284
9285
9286

float (32 bits = 4 bytes)

Storing a double



9278
9279
9280
9281
9282
9283
9284
9285
9286

double (64 bits = 8 bytes)

Storing a Double

Address 0x08

Address 0x0C

Character Strings

- A string of characters can be represented as a *string literal* by putting double quotes around the text:
- Examples:

```
"This is a string literal."
"123 Main Street"
"X"
```

Characters

- A char variable stores a single character
- Character literals are delimited by single quotes:

```
'a'   'X'    '7'    '$'    ','    '\n'
```

- Example declarations:

```
char topGrade = 'A';

char terminator = ';', separator = ' ';
```

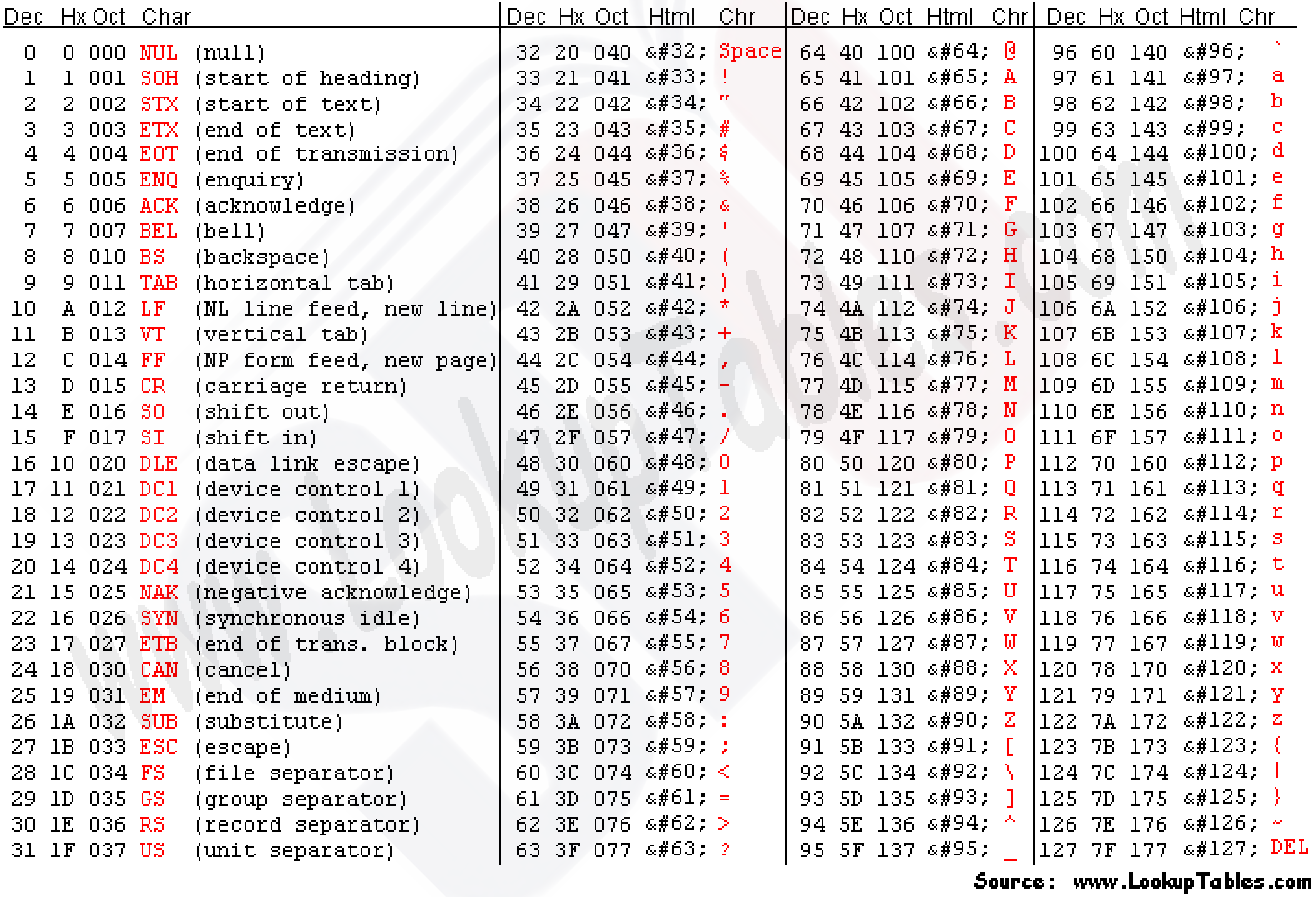
- Note the distinction between a primitive character variable, which holds only one character, and a `String` object, which can hold multiple characters

Characters

- The *ASCII character set* is older and smaller than Unicode, but is still quite popular
- The ASCII characters are a subset of the Unicode character set, including:

uppercase letters	A, B, C, …
lowercase letters	a, b, c, …
punctuation	period, semi-colon, …
digits	0, 1, 2, …
special symbols	&, |, \, …
control characters	carriage return, tab, ...

ASCII Table



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Escape Sequences

- What if we wanted to print a the quote character?
- The following line would confuse the compiler because it would interpret the second quote as the end of the string

```
printf ("I said "Hello" to you.");
```

- An *escape sequence* is a series of characters that represents a special character
- An escape sequence begins with a backslash character (\)

```
printf ("I said \"Hello\" to you.");
```

Escape Sequences

- **Some C escape sequences:**

Escape Sequence	Meaning
`\b`	backspace
`\t`	tab
`\n`	newline
`\r`	carriage return
`\a`	beep
`\"`	double quote
`\'`	single quote
`\\`	backslash

printf() function

- printf(“format string”, variable1, variable2, …);
- printf(“For int use %d”, myInteger);
- printff(“For float use %f”, myFloat);
- printf(“For double use %lf”, myDouble);
- printf(“For float or double %g”, myF_or_D);
- printf(“int=%d double %lf”, myInteger, myDouble);

scanf() function

- **scanf(“format string”, &variable1, &variable2, …);**
- **scanf(“%d”, &myInteger);**
- **scanf(“%f”, &myFloat);**
- **scanf(“%lf”, &myDouble);**
- **scanf(“%d%f”, &myInteger, &myFloat);**

Common Bugs

- **Using & in a printf function call.**

  ```
  printf(“For int use %d”, &myInteger); // wrong
  ```

- **Using the wrong string in printf**

  ```
  printf(“This is a float %d”, myFloat);  // use %f  not %d
  ```

- **Not using & in a scanf() function call.**

  ```
  scanf(“%d”,  myInteger);   // Wrong
  ```

- **Using the wrong string in scanf()**

  ```
  scanf(“%d”, &myFloat);   // wrong;  use %f instead of %d
  ```

PROBLEM SOLVING & PROGRAM DESIGN

Two phases involved in the design of any program:

- **Problem Solving Phase**
  - **Define the problem**
  - **Outline the solution**
  - **Develop the outline into an algorithm**
  - **Test the algorithm for correctness**
- **Implementation Phase**
  - **Code the algorithm using a specific programming language**
  - **Run the program on the computer**
  - **Document and maintain the program**

Structured Programming Concept

- Structured programming techniques assist the programmer in writing effective error free programs.

The elements of structured of programming include:

- Top-down development
- Modular design.

The Structure Theorem:

It is possible to write any computer program by using only three (3) basic control structures, namely:

- **Sequential**
- **Selection (if-then-else)**
- **Repetition (looping, DoWhile)**

ALGORITHMS

An algorithm is a sequence of precise instructions for solving a problem in a finite amount of time.

Properties of an Algorithm:

- **It must be precise and unambiguous**
- **It must give the correct solution in all cases**
- **It must eventually end.**

Developing an Algorithm

- Understand the problem

  (Do problem by hand. Note the steps)

- Devise a plan

  (look for familiarity and patterns)

- Carry out the plan (trace)

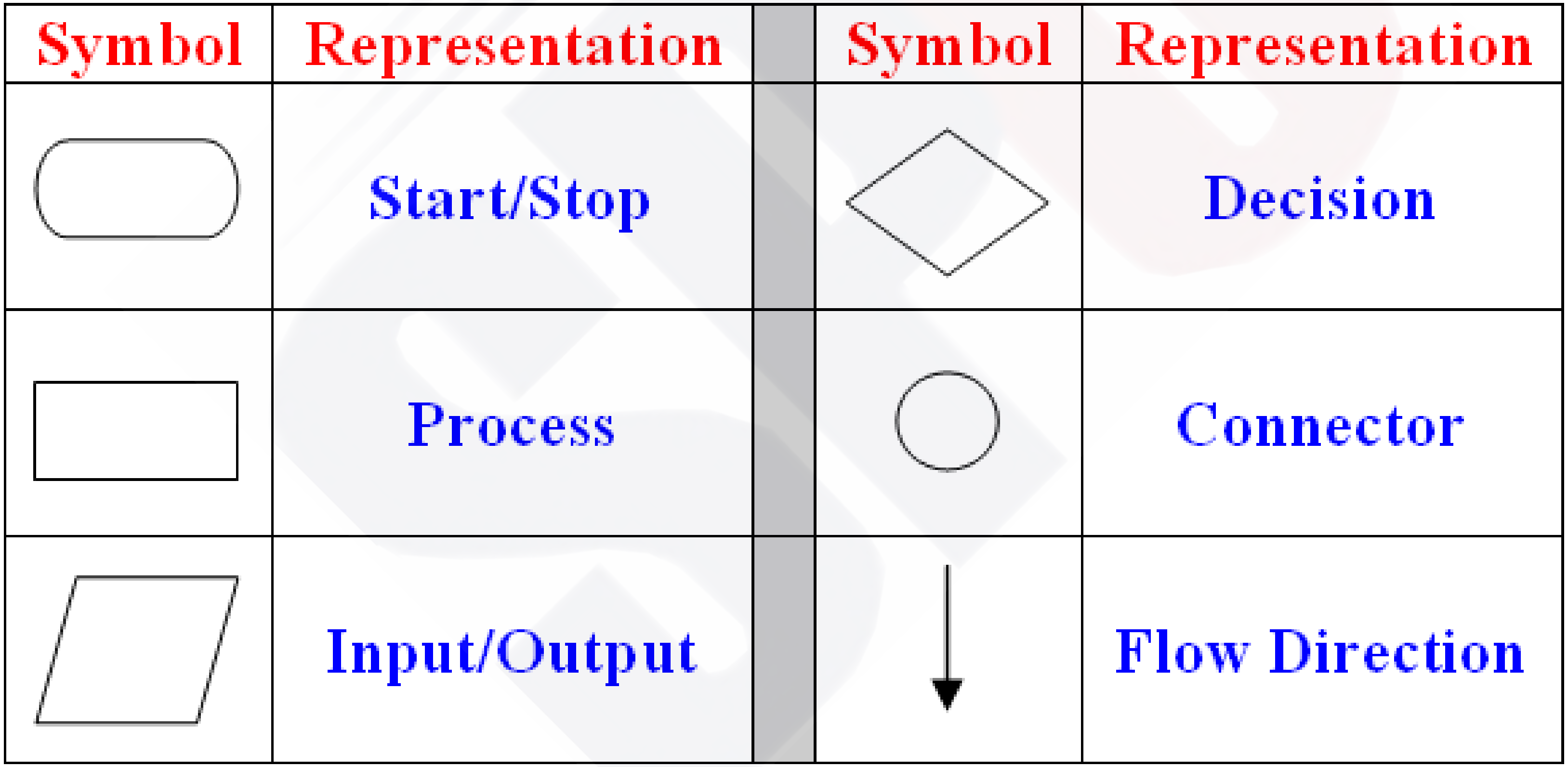
- Review the plan (refinement)

Understanding the Algorithm

Possibly the simplest and easiest method to understand the steps in an algorithm, is by using the flowchart method. This algorithm is composed of block symbols to represent each step in the solution process as well as the directed paths of each step.

Understanding the Algorithm

The most common block symbols are:



Symbol	Representation		Symbol	Representation
⬭	Start/Stop		◇	Decision
▭	Process		○	Connector
▱	Input/Output		↓	Flow Direction

Understanding the Algorithm

Problem Example

Find the average of a given set of numbers.

Understanding the Algorithm - Problem Example

Solution Steps - Proceed as follows:

1. Understanding the problem

   (i) Write down some numbers on paper and find the average manually, noting each step carefully.

   e.g. Given a list say: 5, 3, 25, 0, 9

Understanding the Algorithm - Problem Example

Solution Steps - Proceed as follows:

1. **Understanding the problem**
   - **(i) Write down some numbers on paper**
   - **(ii) Count numbers | i.e. How many? 5**
   - **(iii) Add them up | i.e. 5 + 3 + 25 + 0 + 9 = 42**
   - **(iv) Divide result by numbers counted | i.e. 42/5 = 8.4**

Understanding the Algorithm - Problem Example

Solution Steps - Proceed as follows:

2. **Devise a plan:**

   Make note of NOT what you did in steps (i) through (iv) above, but HOW you did it.

   In doing so, you will begin to develop the algorithm.

For Example:

How do we count the numbers?

Starting at 0 we set our COUNTER to 0.

Look at first number and add 1 to COUNTER.

Look at 2nd number and add 1 to COUNTER.

…and so on,

until we reach the end of the list.

For Example:

How do we add numbers?

Let SUM be the sum of numbers in list.

i.e. Set SUM to 0

Look at 1st number and add number to SUM.

Look at 2nd number and add number to SUM.

…and so on,

until we reach end of list.

For Example:

How do we compute the average?

Let AVE be the average.

then $$\text{AVE} = \frac{\text{total sum of items}}{\text{number of items}}$$

$$= \frac{\text{SUM}}{\text{COUNTER}}$$

Understanding the Algorithm - Problem Example

Solution Steps - Proceed as follows:

3. Identify patterns, repetitions and familiar tasks.

Familiarity: **Unknown number of items?**
i.e. n item

Patterns : **look at each number in the list**

Repetitions: **Look at a number**
Add number to sum
Add 1 to counter

Understanding the Algorithm - Problem Example

Solution Steps - Proceed as follows:

4. Carry out the plan

Check each step

Consider special cases

Check result

Check boundary conditions:

e.g. What if the list is empty?

Division by 0?

Are all data values within specified range?

Understanding the Algorithm - Problem Example

Solution Steps - Proceed as follows:

5. Review the plan:
    - Can you derive the result differently?
    - Can you make the solution more general?
    - Can you use the solution or method for another problem?

e.g. average temperature or average grades

Understanding the Algorithm - Problem Example

A flowchart representation of the algorithm for the above problem can be as follows: