Problem Solving Techniques

by

Liew Shiuh Deh and Wong See Wan

March 1, 2020

Chapter 1

Class introduction

1.1 Course structure

- Course objectives
- Learning outcome
- Class rules
- Marks distribution
- A practical subject
- Hardware and software

1.1.1 Course objectives

1. Introduction to heuristic and algorithmic approach to problem solving
2. Learn logic and number systems
3. Learn algorithm and flow chart
4. Learn to use programming environments
5. Learn programming languages
6. Learn to solve problems using microcontrollers

1.1.2 Learning Outcome

1. identify and understand problems, and using heuristic approach to solve the problems
2. define data types and create algorithms to solve problems
3. write programs using GUI programming environments
4. problem solving using microcontrollers

1.1.3 Class rules

1.1.4 Marks distribution

Class assessment	20%
Project	40%
Term examination	40%

Table 1.1: Marks distribution

1.1.5 A Practical Subject

1. The class put great emphasis on practical hands-on
2. This class is designed to be skill-oriented

Chapter 2

Introduction to Problem Solving

2.1 Problem solving in everyday life

You may not be aware that you are solving problems all the time. For example, an everyday life problem is when you feel hungry, what would you do?

Below are some possible options. Discuss what other problems you may encounter after choosing an option below, and what decision you would make.

- Cook your own food.
- Go to the canteen and find something to eat.
- Go to Mc Donal's.
- Skip the meal.

Discussion

- When you cook your own food, what problems will you face?
- If you were to go to the canteen, what problems will you be facing?
- If you were to go to Mc Donal's, what will be your problems?
- What happens when you skip your meal?

Six steps problem-solving process

1	Identify the problem	Find out the problem before you can solve it.
2	Understand the problem	Knowledge on the problem is required.
3	Identify alternatives	Other acceptable methods to solve the problem.
4	Select the best way to solve the problem	Identify and evaluate the pros and cons of each possible solution before selecting the best solution.
5	List the instructions	Step-by-step instructions to solve the problem.
6	Evaluate the solutions	Check to see if the problem is solved correctly to satisfy the needs.

Discussion

You have some knowledges on how to solve problems systematically, now try to solve the meal problem using these steps.

2.2 Types of problem solving methods

2.2.1 Heuristic solutions

- Based on knowledge and prior experience
- Go through a process of trial and error
- The six-steps problem-solving processes can be used
- Solutions that cannot be reached through a direct set of steps
- Correctness and appropriateness far less certain
- May need to iterate many times

- Same solutions may not work all the time

2.2.2 Algorithmic solutions

- Solving problems by following some rules or series of actions, such as baking a cake, balancing a chequebook,
- The first five steps of problem-solving processes can be used

Most problems require a combination of the two methods

2.3 Problem solving with computers

Algorithmic solutions can be solved with computers, and effectively. For example, calculation problems, performing repetitive task problem, and controlling devices such as TV, air-conditional, robot.

Heuristic solutions can be solved effectively by people. For example, problems such as playing a chess game, doing house chores, speaking a language. These problems can be solved using computer, but require high computing power computer to play chess game, artificial intelligence to make a robot that does house chores, and speak a language.

The focus of this class is to learn computer algorithms, utilising sensors to detect variables, and write programs on microcontroller to solve problems.

Chapter 3

Program design

3.1 Steps in program development

Computer programming is an art. Anybody given the right tools(programming development environment, compiler) and steps to follow as discuss below can write well-designed programs.

3.1.1 Define the problem

Understand the problem completely and divide the problem into the following components:

- Inputs
- Outputs
- Processing steps to produce the outputs

3.1.2 MakeCode

https://makecode.microbit.org

MakeCode environment

Familiar with MakeCode environment. Select some Basic blocks from Toolbox, and learn to connect the blocks.

on start

Use the `on start` block to display scrolling texts.
Display Hello! using `show string` block in `on start` block.

forever

Repeat the above in `forever` block and observe the output.

3.1.3 Outline the solution

Break down the solution into smaller tasks, and establish a solution outline. The rough draft of the outline are:

- the major processing steps involved
- the major subtasks (if any)
- the user interface (if any)
- the major control structures (e.g. repetition loops)
- the major variables and record structures
- the mainline logic

Repetition loops

Perform a countdown begin from 3 using `show string` and `pause` blocks.

User interface

Include user input to start counting when button A is pressed.
Use `on button A pressed`.

3.1.4 Develop the outline into an algorithm

The solution outline is expanded into an algorithm, which is a set of precise steps and order to be carried out in a program.

Pseudocode (a form of structured English) is used to represent the solution algorithm.

Below is an example of a pseudocode for maintaining an air-conditional temperature.

```
Set temperature
DO WHILE air-conditional is on
Obtain room temperature
If room temperature < set temperature
   Switch off compressor
else if room temperature > set temperature
  Switch on compressor
else
  Keep compressor running
END DO
```

Flowchart is a pictorial method of algorithm representation.

Below is an example of a flowchart for maintaining an air-conditional temperature.

```
[Set temperature]
            |
            v                               Yes
+--> (if room temperature < set temperature) -->  Switch off compressor
|           | No
|           v                               Yes
|    (if room temperature > set temperature) -->  Switch on compressor
|           | No
|           v
+-- Keep compressor running
```

3.1.5 Test the algorithm for correctness

- Identify logic errors so that they can be easily corrected at early stage
- Walk through the algorithm as if the computer executes the instructions

3.1.6 Code the algorithm into a specific programming language

Below are examples of programming languages.

Javascript

```
input.onButtonPressed(Button.A, function () {
    basic.showString("3")
    basic.pause(1000)
    basic.showString("2")
    basic.pause(1000)
    basic.showString("1")
    basic.pause(1000)
    basic.showString("0")
})
```

C

```
int main()
{
    printf("3\n");
    sleep(1);
    printf("2\n");
    sleep(1);
    printf("1\n");
    sleep(1);
    printf("0\n");
    sleep(1);

    return 0;
}
```

Introduce MakeCode programming environment and Arduino Integrated Development Environment. Look at the similarities between MakeCode `on start` block and `forever` block, and Arduino `setup` and `loop` functions.

3.1.7 Compile and run the program

- To compile a program means using a compiler to compile or convert a program written in human readable programming language to computer executable program or application.

- If errors in the program exist, then correct the errors or bugs and repeat this step.

3.1.8 Document and maintain the program

- Program documentation is to describe how a program is written for future reference or other programmers to refer to in order to be able to maintain the program.
- Program documentation consists of two parts, internal documentation or inline documentation which describes the purpose of the line of computer code or the function of that part of the code.
- External documentation is a detail description of how a program is designed, which includes flowcharts, algorithms, and data format.
- The purpose of the documentation is for people to understand how the program is designed so that the program can be modified or upgraded.

Example of inline documentation

```
/*-------Input ports------*/
#define RS 2      // right sensor
#define LS 3      // left sensor


/*-------Outputs ports------*/
#define LM1 5       // left motor
#define LM2 4       // left motor
#define RM1 6      // right motor
#define RM2 7       // right motor
```

An example program that increments a value using MakeCode

- In `on start`, insert `set variable` block and set a value, say 0.
- In `forever` block, insert `change variable to` block to a value, say 1.
- Run the program and see what happens.
- Move the `set variable` block and put on the first item in the `forever` block, then run the program and see what happens.

Chapter 4

Program data

One thing that computer can do very efficiently is processing data. Other problems that computers can help solve efficiently are problems involving mathematical processing, and logical processing.

We need to understand how computers define and process data in order to write efficient programs for computers to execute.

Data are input into computers and be processed then output as information. Below is an example of how a bank computer keeps track of your bank account balance.

Data	Input	Computer	Output	Information
Deposits and withdrawals	$\rightarrow$	Program calculates the balance	$\rightarrow$	Balance

Data are input or provided to a computer, and the program process the data based upon its pre-programmed algorithm. The above example is to calculate deposits balance, and the result or output of the calculation is the balance.

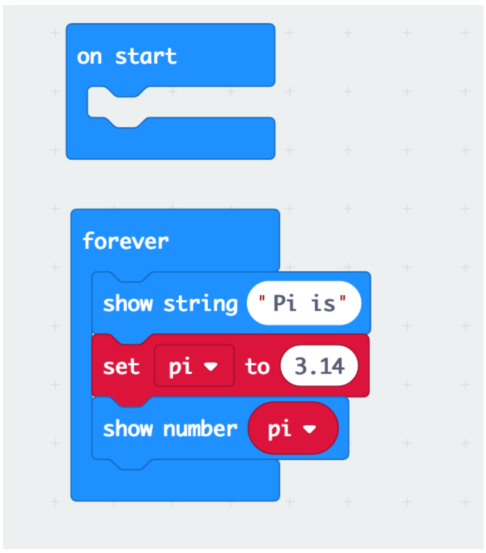
4.1 Constants

Computers process data into information, and constants are part of the data.

- Constants are data or fixed values that are embedded in a program, and these data are not changed while a program is executing.

- Every constant value is given a name, so that the value can be referred to.
- Constant values are stored in computer memory, and these values are used whenever they are needed in a program.
- Constant values are often used in formulas, such as the area of a circle is πr^2 and π is a constant value of 3.142857142857143 and this value does not change.

Demonstrate how the π value is shown using MakeCode.



```
on start

forever
    show string "Pi is"
    set pi to 3.14
    show number pi
```

Figure 4.1: MakeCode Show Pi

Another type of constant is a name or a label. For example, the name of a person or description of a label. The "Pi is" label and `pi` number in the exercise above are constant values.

4.2 Variables

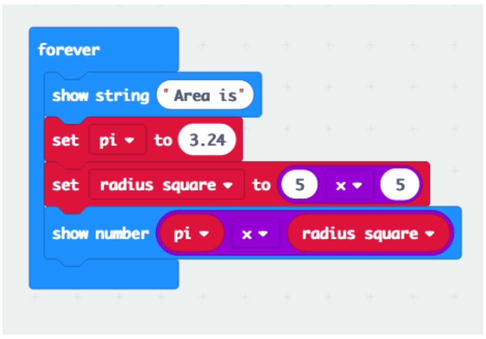
Variables are data that change while a program is executing. These data are stored in computer memory and are refered to through the names provided.

Each piece of data or variable is provided a unique name, so that we can refer to the correct piece of data.

Let's use the πr^2 formula to calculate different sizes of circles using a computer. What we need to provide the computer are the formula, the π value, and the radius. In this case, π is a constant, and radius is a variable. We provide a relevant name to radius r and called it `radius`.

In computer languages, we use `pi` $*$ `r` $*$ `r` to represent the πr^2 formula, where π is the constant value of 3.1428, the asterisk $*$ symbol represents multiply, and `r` is the radius of the circle.

Demonstrate how the π formula is applied using MakeCode `Variables` blocks.



```
forever
    show string "Area is"
    set pi to 3.24
    set radius square to 5 x 5
    show number pi x radius square
```

Figure 4.2: MakeCode Area of Circle

4.3 Constant and variable in C

We use an online C compiler https://www.onlinegdb.com to demonstrate how constant and variable are used in the C programming language.

In Figure 4.3, `const` is called a *quantifier* that specifies floating point or real number `pi` is a constant value, and this value will not change.

The `int` declares that `radius` is a variable value, and this value can be changed to calculate different areas of circles. In the example, the radius of first circle is 5, and radius of second circle is 8. The `radius` is declared once, but has been set to different radius for calculating different area of circles.

`float area` without the `const` *quantifier* is declaring `area` as a variable, and `area` is *assigned* the result of the `pi` $*$ `r`^{2} formula. Since `area` is declared

```
#include <stdio.h>

int main()
{
   const float pi = 3.1428;  /* Declare pi as a constant number */
   int radius = 5;           /* Declare radius as a variable which can be changed */
   float area;

   area = pi * radius * radius;

    /* Display the result after computing the area */
    printf("Area of first circle is %f\n", area);

    radius = 8;
    area = pi * radius * radius;

    /* Display the result after radius is set to a different value */
    printf("Area of second circle is %f\n", area);

    return 0;
}
```

Figure 4.3: Constants and variables in C

as a variable, so it is reused to store different area of circles and the data is stored in memory then printed as output of the program.

Below are output of the program.

```
Area of first circle is 78.570007
Area of second circle is 201.139206
```

4.4 Data types

Data are unorganised facts in the form of numbers or characters that have not yet been processed into information. Data are input into computer, and be processed by a program based upon an algorithm to become information. The information is either be displayed on the computer screen, printed as hard copy, stored in the hard drive for retrieval in future, or transmitted to other computers.

4.4.1 bits and bytes

Before we learn data types, we need to know bits and bytes. Data types are grouping of contiguous bytes to hold a piece of data. A bit is the smallest unit of storage in a computer, it is either a 1 or a 0. A group of 8 bits is a byte. The smallest value of a byte is 0000 0000 or 0, and the largest value of a byte is 1111 1111 or 255.

We will learn more on bits and bytes in Section 4.5.

4.4.2 `char`

`char` data type or character constant is for storing a single character, such as 'a', 'b', 'C', '1', or symbols such as '@', '#', '&', etc. Each character is 1 byte which means we have 256 different characters or symbols in `char` data type.

Below shows the ASCII characters between 0 and 127. Numbers on the left represent the predefined characters or symbols on the right. For example, the character 'a' is represented by number 97 which means we see 'a' on the computer screen, but it is number 97 to the computer.

```
  0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack    7 bel
  8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so    15 si
 16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn   23 etb
 24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs    31 us
 32 sp    33  !    34  "    35  #    36  $    37  %    38  &    39  '
 40  (    41  )    42  *    43  +    44  ,    45  -    46  .    47  /
 48  0    49  1    50  2    51  3    52  4    53  5    54  6    55  7
 56  8    57  9    58  :    59  ;    60  <    61  =    62  >    63  ?
 64  @    65  A    66  B    67  C    68  D    69  E    70  F    71  G
 72  H    73  I    74  J    75  K    76  L    77  M    78  N    79  O
 80  P    81  Q    82  R    83  S    84  T    85  U    86  V    87  W
 88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^    95  _
 96  `    97  a    98  b    99  c   100  d   101  e   102  f   103  g
104  h   105  i   106  j   107  k   108  l   109  m   110  n   111  o
112  p   113  q   114  r   115  s   116  t   117  u   118  v   119  w
120  x   121  y   122  z   123  {   124  |   125  }   126  ~   127 del
```

The 10 decimal numbers from 0 to 9 are in the character set as well, and each of these 'numbers' has a numeric value. The number '1' has a numeric value of 49. These numeric characters are mainly for display purposes and not for performing computation. We will discuss how to make use of the numeric values in future chapters.

4.4.3 `int`

Computer must be told that a character represents a number or an alphabet, and whether a number is for computation or for display. For example, we have two numbers 1 and 2, and add them together such as $1 + 2 = 3$. These are numbers for computation, so we need to tell the computer or *declare* these numeric values as numbers. Another example, the '1' in, say "Figure 1", is not for computation but for showing that is the first figure; thus, this '1' is part of a string of characters, and we do not add "Figure 1" and Figure 2" to produce "Figure 3".

We use `int` to represent integer number, and an `int` is 2 or 4 bytes in size depending upon the microcontroller or microprocessor that is used to execute the program. For a two 2 bytes integer, we can have 2^{16} bits to store 65,536 different numbers.

Integers can be declared as sign or unsign, a signed integer means it is a positive or negative whole number. A 16 bits signed `int` can have a range of numbers between -32,768 and 32,767; whereas, a 16 bits `unsigned int` stores only positive numbers between 0 and 65,535.

4.4.4 `long`

The largest value an `int` can hold is 2^{16} or 2^{32}. This value is too small for some applications, so `long` is a data type for holding large value integers. The size of `long` is four or eight bytes in size or 2^{32} or 2^{64} bits.

One may ask why not declare all numbers as `long`? The answer is we do not use large numbers all the time. The number of bytes to hold a `long` is double the size of an `int` which takes up computer memory, and may take up more computing power.

Similar to `int` data type, `long` can be declared as signed or unsigned. A signed 8 bytes or 2^{64} `long` has a range of numbers between
-9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.

4.4.5 `float`

Numbers that have decimal points are real numbers, and are also called floating-point numbers. These numbers are declared as `float` in C and C++. `float` data type is 4 bytes in size for storing 6 to 9 significant digits of real numbers. The number of digits can be stored in a `float` data type is much smaller than an `int` or a `long` data type, this is because the 4 bytes of `float` is used to store significant digits and exponential values to make up a real number.

`float` values are signed and cannot be compared for same or different in value; for instance, $\frac{1}{3}$ is not the same as 0.33333.

4.4.6 Character string

When a sequence of characters, such as "Hello world", are joined together to form a phrase or a label, this is called an *array* of characters. There are several ways to declare a memory space for holding an array of characters. The first method is declared a memory space as a character array just large enough to hold all the characters, and the data in this piece of array will not be changed.

The second method is to declare a fixed size array, say an array of 128 characters, for keeping texts of different lengths shorter than the array size.

Figure below depicts how characters are kept in a character array. The character array is declared as `char s[] = "Hello world";`. The array begins from the left side, and the first character has an index number 0, and this character space holds the '`H`' character. The next character space has index number 1, and keeps the '`e`' character. The last character in the array is "`\n`", which is a carriage return. This character is included by the compiler to specify the end of a string. Carriage return is a non-printable character, so it is not displayed on the screen but is stored in the array.

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]	s[10]	s[11]
H	e	l	l	o		w	o	r	l	d	\n

The third method is using the C++ `string` class to declare strings of characters. Once a string is declared using the `string` class, it can be shorten, attach to another string, insert texts into the string, etc.

We will discuss other methods of allocating memory space for storing data, array made up of other data types, and learn how to manipulate each element in an array in future chapters.

4.4.7 Exercises

Checking the size of different data types

Type the program in Figure 4.4 and compile it using https://www.onlinegdb.com online compiler. Change the **Laguage** option on the top right corner to **C**, then click on the ▷ Run green colour button to compile and execute the program.

```
#include <stdio.h>

int main()
{
        char c;
        int i;
        long l;
        float f;
        char s[] = "Hello world";

        printf("The size of char is %ld\n", sizeof(c));
        printf("The size of int is %ld\n", sizeof(i));
        printf("The size of long is %ld\n", sizeof(l));
        printf("The size of float is %ld\n", sizeof(f));
        printf("The size of s character array is %ld\n", sizeof(s));

        return 0;
}
```

Figure 4.4: Size of data types

Assigning values to data types

Figure 4.5 shows how to assign values to the data types.

```c
#include <stdio.h>

int main()
{
        char c = 'A';
        int i = 123;
        long l = 123456;
        float f = 1.23456;
        char s[] = "Hello world";

        printf("char c contains %c\n", c);
        printf("integer i contains %d\n", i);
        printf("long l contains %ld\n", l);
        printf("float f contains %f\n", f);
        printf("character array s contains %s\n", s);

        return 0;
}
```

Figure 4.5: Populating the data types

4.5 Number systems

Decimal number

The base-10 or Arabic numeral system consists of 10 numeric symbols ranging from 0 to 9, such as 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. These symbols can be combined to form numbers of different values. The unit number is denoted by $n \times 10^0$, where n can be any of the base-10 symbols and 10^0 is 1. The 10th position value is denoted by $n \times 10^1$, and the 100th position value is denoted by $n \times 10^2$, and so on.

Table below shows how to calculate the value in base-10 number from a set of numbers, say 12345. In this case, 5 is worth five ones, 4 is worth four tens and we get 40, 3 is worth three hundred and we get 300, 2 is worth two thousands and we get 2,000, and 1 is worth 10 times of a thousand and we get 10,000. When we add the numbers together, we obtain 12,345 in decimal as shown in 4.1.

10,000th	1,000th position	100th position	10th position	1st unit
10^4	10^3	10^2	10^1	10^0
1×10^4	2×10^3	3×10^2	4×10^1	5×10^0
10,000	2,000	300	40	5

$$10,000 + 2,000 + 300 + 40 + 5 = 12,345 \tag{4.1}$$

Binary number

Binary number or base-2 number consists of two numeric symbols 0 and 1. Each symbol is one bit and start counting from right to left. Table 4.1 shows the value of each bit, which can be represented by $2^{(n-1)}$ where n is the number of bits. The right most bit is the LSB (Least Significant Bit), and the left most bit is the MSB (Most Significant Bit).

Let's say we have 8 bits of 10101010 in binary, we can convert it to base-10 decimal value using table 4.1. What we need to do is placing the bits according to the position of the bits, then add up the $2^{(n-1)}$ numbers on the columns that have a 1 and we get $128 + 32 + 8 + 2 = 170$.

On the other hand, we can convert a base-10 number, say 123, to binary number by dividing the base-10 number by 2 repeatedly and the remainders are the binary number equivalent to the base-10 number. In this case, 123 in base-10 is equivalent to 1111011 in binary or base-2.

MSB 8th bit	7th bit	6th bit	5th bit	4th bit	3rd bit	2nd bit	LSB 1st bit
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	1	0	1	0	1	0

Table 4.1: The value in a byte

Base-10 Divisor	Base-10	Base-10 Remainder	Base-2
2	123		
2	61	$123 - (2 \times 61) = 1$	1
2	30	$61 - (2 \times 30) = 1$	1
2	15	$30 - (2 \times 15) = 0$	0
2	7	$15 - (2 \times 7) = 1$	1
2	3	$7 - (2 \times 3) = 1$	1
2	1	$3 - (2 \times 1) = 1$	1
	0	$1 - (2 \times 0) = 1$	1

Octal number

Octal number or base-8 number system consists of the first 8 numbers of the base-10 number symbols, such as 0, 1, 2, 3, 4, 5, 6, and 7. Each digit in octal number is eight to the power - 1 or 8^{n-1}. Table below shows the conversion from a base-8 number 012345(octal numbers are preceded by 0) to base-10 number, and in binary number. An octal number can be represented by groups of three bits.

Base-n	8^4	8^3	8^2	8^1	8^0	Values
Base-8	1	2	3	4	5	012345
	1×8^4	2×8^3	3×8^2	4×8^1	5×8^0	
Base-10	1 x 4096	2 x 512	3 x 64	4 x 8	5 x 1	5349
Base-2	001	010	011	100	101	001 010 011 100 101

We can convert the base-10 number 5349 back to base-8 by using the divide method. In this case, we use 8 as the divisor and divide 5349 repeatedly until 0 remains. The remainders 12345 is octal 012345 equivalent to 5349 in decimal.

Base-10 Divisor	Base-10	Base-10 Remainder	Base-8
8	5349		
8	668	$5349 - (8 \times 668) = 5$	5
8	83	$668 - (8 \times 83) = 4$	4
8	10	$83 - (8 \times 10) = 3$	3
8	1	$10 - (8 \times 1) = 2$	2
	0	$1 - (8 \times 0) = 1$	1

Hexadecimal number

Hexadecimal is base-16 or there are 16 base number symbols. The first 10 symbols are the same as base-10 symbols, and the 11th symbol or number 10 is A, the 12th symbol or number 11 is B, and the 16th symbol or number 15 is F, as shown below.

Base-10	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base-16	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

A hexadecimal base number can be represented by 4 bits of binary number as shown in the tables below. Hexadecimal numbers are preceded by 0x followed by the number, such as 0xC3A, which can be represented by 1100 0011 1010 in binary.

Base-16	7	6	5	4	3	2	1	0
Base-2	0111	0110	0101	0100	0111	0010	0001	0000

Base-16	F	E	D	C	B	A	9	8
Base-2	1111	1110	1101	1100	1011	1010	1001	1000

Conversion from hexadecimal to decimal is similar to conversion from other number systems to decimal introduced above. Table below shows how to convert hexadecimal number to decimal number and binary number.

Base-n	16^3	16^2	16^1	16^0	Values
Base-16	1	2	3	4	0x1234
	1×16^3	2×16^2	3×16^1	4×16^0	
Base-10	1×4096	2×256	3×16	4×1	4660
Base-2	0001	0010	0011	0100	0001 0010 0011 0100

To convert a based-10 number, say 3130, to hexadecimal number simply divide the base-10 number by 16 repeatedly until 0 remains. The remainders are C3A, which is hexadecimal number 0xC3A. The conversion is as shown below.

Base-10 Divisor	Base-10	Base-10 Remainder	Base-16
16	3130		
16	195	$(3130 - 16 \times 195) = 10$	A
16	12	$(195 - 16 \times 12) = 3$	3
	0	$(12 - 16 \times 0) = 12$	C

4.6 Operators

4.6.1 Arithmetic operators

Arithmetic operators are for computation, such as addition, subtraction, multiplication, and division. These operations are denoted by +, -, *, and / respectively. The program in Figure 4.6 demonstrates how the operators are applied. Division only works with `float` data type; otherwise, the decimal numbers are ignored.

```
1   #include <stdio.h>
2
3   int main()
4   {
5       int a = 1;
6       int b = 2;
7       float c = 3.0;
8       float d = 4.5;
9
10      printf("%d + %d = %d\n", a, b, a + b);
11      printf("%d - %d = %d\n", a, b, a - b);
12      printf("%f * %f is %f\n", c, d, c * d);
13      printf("%f / %f is %f\n", c, d, c / d);
14
15      return 0;
16  }
```

Figure 4.6: Arithmetic operators

To print an `int` value such as in `printf("Integer %d\n", 123);` use `"%d"`, and the `"\n"` symbol is to place a new line on the text. Use `"%ld"` for `long` values such as `printf("Long integer %ld\n", 123456);`. Use `"%f"` for decimal numbers such as `printf("Float number %f\n", 12.3456);`

Exercises

1. Change line 5 to `int a = 1.5;` , then run the program and observe the results.

2. Change line 7 to `float c = 3;` , then run the program and observe the results.

3. Change the addition on line 10 to division, and observe the results.

4.6.2 Relational operators

Relational operators are for comparing true or false conditions, such as one value is larger than or equal to the other. The operators are > larger than, >= larger than or equal to, < less than, <= less than or equal to.

The comparison using relational operator in 4.2 has a false result because 1 is not larger than 1, but comparison in 4.3 is true because 1 is not larger than but equal to the other 1.

`1 > 1` (4.2)

`1 >= 1` (4.3)

Both comparisons in 4.4 and 4.5 are true because 1 is less than 2. The comparison in 4.6 is true because 2 is not less than but equal to the other 2.

`1 < 2` (4.4)

`1 <= 2` (4.5)

`2 <= 2` (4.6)

The *equality* operators `==` and `!=` are for comparing whether two items are equal or not equal respectively. For example, if `A` has a value of 1 and `B` has a value of 2, and we compare them as shown in comparison 4.7 the answer is false, but the answer is true in comparison 4.8 .

`A == B` (4.7)

`A != B` (4.8)

4.6.3 Logical operators

Logical operators `&&` AND, and `||` OR are for evaluating *true* or *falsehood* of expressions. These operators are *connectors* of expressions and equations. During data processing, different expressions are evaluated through logical operators to produce answers. The expressions are *operands* and the answer is *resultant*. The *resultant* is always `TRUE` or `FALSE`.

We make decisions all the time, for instance you may consider going to the beach if today "is a sunny day" AND "you are free". In this case, "is a sunny day" is an *operand*, and "you are free" is another *operand*. These two *operands* are compared with AND operator, only if the *resultant* is TRUE, then you go to the beach.

Another example is if today "is a sunny day" OR "the wave is up", you will go to the beach. In this case, as long as one operand is TRUE, then you will go to the beach. When it is sunny but no waves you will go to the beach, when the wave is up doesn't matter it is sunny or raining you will go to the beach, and when it is sunny and the wave is up you sure will go to the beach. The *resultant* is FALSE only if both *operands* are FALSE, which means it is not sunny nor wave is up.

Exercises

1. Write an expression using AND operator to compare two operands for a TRUE resultant.
2. Write an expression using OR operator to compare two operands for a TRUE resultant.
3. Write an expression using AND and OR operators to compare operands for a TRUE resultant.
4. Write an expression using an AND, a OR, and an AND operators to compare operands for a TRUE resultant.
5. Write an expression using a OR, an AND, and a OR operators to compare operands for a TRUE resultant.

Chapter 5

Problem Solving with Decisions

5.1 The Decision Logic Structure

We make a decision when we need to accomplish something or to acquire something, but due to limited resources or certain circumstances to meet we can only select from the best choice out of a list of options. `if` is the condition statement that we use to evaluate the options to obtain the result.

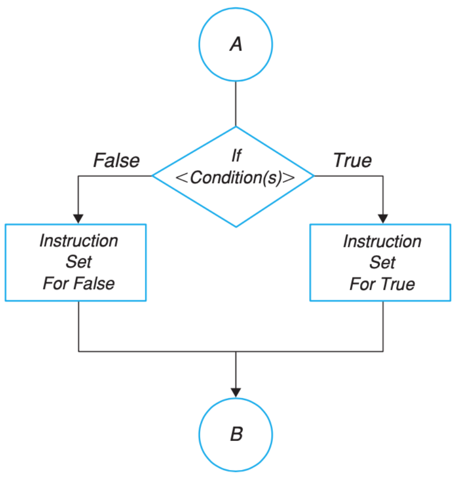


Figure 5.1: If condition

The outcome of an `if` condition has only two states, either TRUE or FALSE. As shown in Figure 5.1, when the comparison of some conditions turns out to be TRUE, then we do something; otherwise, we do something else. The condition could be ”is a sunny day” or ”is examination period”. We may make a decision if it is a sunny day, then we go to the beach as shown in the pseudocode below.

```
if (is a sunny day) then
   go to the beach
```

We can add ELSE to the IF statement to become IF something happens THEN do something, ELSE do something else as shown below.

```
if (is a sunny day) then
   go to the beach
else
   stay home
```

5.2 `if` statement

We use `if` statement to evaluate conditions in computer programming. The pseudocode examples in the previous section can be converted into a program as shown in Figure 5.2. The condition is `sunny_day` which is either a 1 for TRUE or 0 for FALSE.

```c
#include <stdio.h>

int main()
{
    int sunny_day = 1;

    if (sunny_day)
       printf("Go to the beach\n");
    else
       printf("Stay home\n");

    return 0;
}
```

Figure 5.2: If sunny day

We can have multiple conditions combined with operators in an `if` statement. As shown in Figure 5.3, two conditions are compared, first condition is `sunny_day`, and second condition is `surf_up`. The results of the `if` statement

```
#include <stdio.h>

int main()
{
    int sunny_day = 1;
    int surf_up = 1;

    if (sunny_day && surf_up)
       printf("Go to the beach\n");
    else
       printf("Stay home\n");

    return 0;
}
```

Figure 5.3: If sunny day and surf's up

is TRUE only if both `sunny_day` and `surf_up` are TRUE, and the statement right below if statement is executed. Either one of the conditions is FALSE the statement below `else` is executed.

5.3 Nested if statement

We can have `if` statement inside another `if` statement, and this is called nested `if` statement. This is appled to evaluating some conditions only when specific condition occurs. The example in Figure 5.4 shows that only if it is a sunny day then the condition inside the if statement is evaluated. In this case, the `if (exam)` condition is evaluated. This nested `if` condition is to decide staying at the beach for an hour or till sun set.

We also can evaluate conditions in `else`, where the `if` condition returns FALSE. In the example, the `if (exam)` condition is again evaluate, but the outcome is different. This time is deciding to study for an hour or play game.

We can use negation `!` not or `!=` not equal to relational operator in `if` statement as shown below.

```
if (!sunny_day) {
   printf("Stay home\n");
   printf("and do homework\n");
}
```

When there are more than one statements need to be executed after an if evaluation, place a pair of `{}` before and at the end of the statements to

```c
#include <stdio.h>

int main()
{
    int sunny_day = 1;
    int exam = 1;

    if (sunny_day) {
       printf("Go to the beach\n");
       if (exam)
          printf("Stay for one hour\n");
       else
          printf("Stay till sun set\n");
    } else {
       printf("Stay home\n");
       if (exam)
          printf("Study for one hour\n");
       else
          printf("Play game\n");
    }

    return 0;
}
```

Figure 5.4: If sunny day and exam

specify that the statements inside the pair of `{}` are in a group and to be executed together.