PROGRAMMING METHODOLOGY
(PHƯƠNG PHÁP LẬP TRÌNH)

UNIT 3: Overview of C Programming Language

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Currently, there are no modification on these contents.

Unit 3: Overview of C Programming

Objectives:

- Learn basic C constructs, interactive input, output, and arithmetic operations
- Learn some data types and the use of variables to hold data
- Understand basic programming style

References:

- Chapter 2 Variables, Arithmetic Expressions and Input/Output
- Chapter 3 Lessons 3.1 Math Library Functions and 3.2 Single Character Data

Unit 3: Overview of C Programming

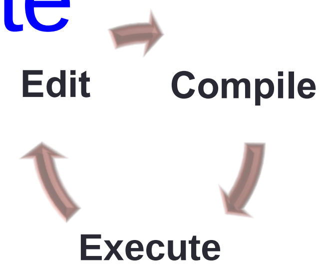
1. A Simple C Program
2. Variables and Data Types
3. Program Structure
   - Preprocessor directives
   - Input
   - Compute
   - Output
4. Math Functions
5. Programming Style
6. Common Mistakes

Introduction

- C: A general-purpose computer programming language developed in 1972 by Dennis Ritchie (1941 – 2011) at Bell Telephone Lab for use with the UNIX operation System
- We will follow the ANSI C (C90) standard

  http://en.wikipedia.org/wiki/ANSI_C

Quick Review: Edit, Compile, Execute



Test, test, and test!

Edit eg: vim first.c	*produces* →	*Source code* first.c
Compile eg: gcc first.c	*produces* →	*Executable code* a.out
Execute eg: a.out	*produces* →	*Program output* `The value of c is 3.`

Cannot compile?

Incorrect result?

A Simple C Program (1/3)

- General form of a simple C program

preprocessor directives
main function header
{
    declaration of variables
    executable statements
}

"Executable statements" usually consists of 3 parts:
- Input data
- Computation
- Output results

A Simple C Program (2/3)

Unit3_MileToKm.c

```
// Converts distance in miles to kilometres.
#include <stdio.h>   /* printf, scanf definitions */
#define KMS_PER_MILE 1.609 /* conversion constant */

int main(void) {
    float miles,   // input - distance in miles
          kms;     // output - distance in kilometres

    /* Get the distance in miles */
    printf("Enter distance in miles: ");
    scanf("%f", &miles);

    // Convert the distance to kilometres
    kms = KMS_PER_MILE * miles;

    // Display the distance in kilometres
    printf("That equals %9.2f km.\n", kms);

    return 0;
}
```

Sample run

```
$ gcc -Wall Week2_MileToKm.c
$ a.out
Enter distance in miles: 10.5
That equals     16.89 km.
```

A Simple C Program (3/3)

```c
// Converts distance in miles to kilometres.

#include <stdio.h>    /* printf, scanf definitions */
#define KMS_PER_MILE 1.609 /* conversion constant */

int main(void) {
    float miles,    // input - distance in miles
          kms;      // output - distance in kilometres

    /* Get the distance in miles */
    printf("Enter distance in miles: ");
    scanf("%f", &miles);

    // Convert the distance to kilometres
    kms = KMS_PER_MILE * miles;

    // Display the distance in kilometres
    printf("That equals %9.2f km.\n", kms);

    return 0;
}
```

preprocessor directives

standard header file

constant

reserved words

variables

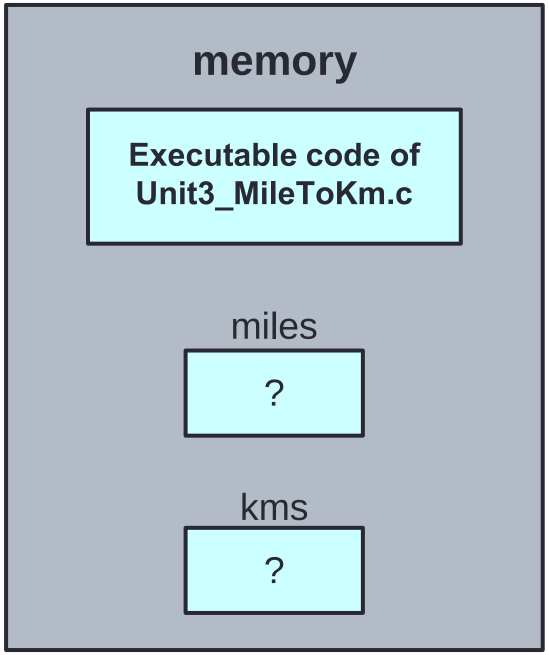
comments

functions

special symbols

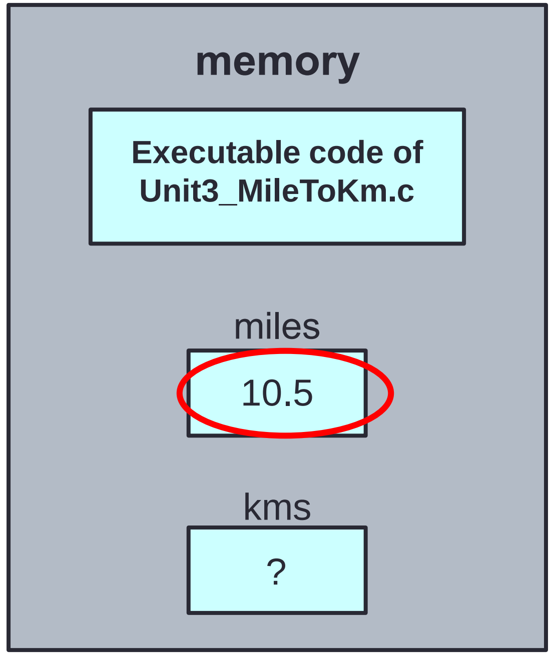
punctuations

What Happens in the Computer Memory



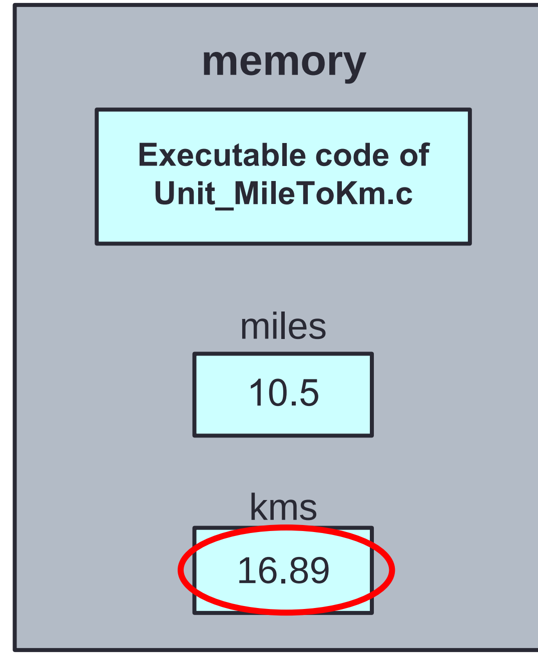
At the beginning

Do not assume that uninitialised variables contain zero! **(Very common mistake.)**



After user enters: 10.5 to

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



After this line is executed:

```
kms = KMS_PER_MILE * miles;
```

Variables

- Data used in a program are stored in variables
- Every variable is identified by a name (identifier), has a data type, and contains a value which could be modified
- A variable is declared with a data type
  - Eg: `int count; // variable 'count' of type 'int'`
- Variables may be initialized during declaration:
  - Eg: `int count = 3; // count is initialized to 3`
- Without initialization, the variable contains an unknown value (Cannot assume that it is zero)

Variables: Mistakes in Initialization

- Incorrect: No initialization

```
int count;

count = count + 12;
```

Does ‘count’ contain 12 after this statement?

- Redundant initialization

```
int count = 0;

count = 123;
```

Initialization here is redundant.

Data Types

- To determine the type of data a variable may hold
- Basic data types in C (more will be discussed in class later):
  - int: For integers
    - 4 bytes (in sunfire); -2,147,483,648 (-2^{31}) through +2,147,483,647 ($2^{31} - 1$)
  - float or double: For real numbers
    - 4 bytes for float and 8 bytes for double (in sunfire)
    - Eg: 12.34, 0.0056, 213.0
    - May use scientific notation; eg: 1.5e-2 and 15.0E-3 both refer to 0.015; 12e+4 and 1.2E+5 both refer to 120000.0
  - char: For individual characters
    - Enclosed in a pair of single quotes
    - Eg: `'A'`, `'z'`, `'2'`, `'*'`, `' '`, `'\n'`

http://www.tutorialspoint.com/ansi_c/c_basic_datatypes.htm

Notes (1/2)

- Basic steps of a simple program
  1. Read inputs (scanf)
  2. Compute
  3. Print outputs (printf)
- For now we will use interactive inputs
  - Standard input stream (stdin) – default is keyboard
  - Use the scanf() function
- Assume input data always follow specification
  - Hence no need to validate input data (for now)
- Outputs
  - Standand output stream (stdout) – default is monitor
  - Use the printf() function

Notes (2/2)

- Include header file <stdio.h> to use scanf() and printf()
  - Include the header file (for portability sake) even though some systems do no require this to be done
- Read
  - Lessons 1.6 – 1.9
- Important! (CodeCrunch issue)
  - Make sure you have a newline character (‘\n’) at the end of your last line of output, or CodeCrunch may mark your output as incorrect.

```
printf("That equals %9.2f km.\n", kms);
```

Type of Errors

Easiest to spot – the compiler helps you!

- Syntax errors (and warnings)
  - Program violates syntax rules
  - Warning happens, for example, incomparable use of types for output
  - Advise to use **gcc –Wall** to compile your programs

Moderately easy to spot

- Run-time errors
  - Program terminates unexpectedly due to illegal operations, such as dividing a number by zero, or user enters a real number for an integer data type

Hard to spot

- Logic errors
  - Program produces incorrect result

May never be spotted!

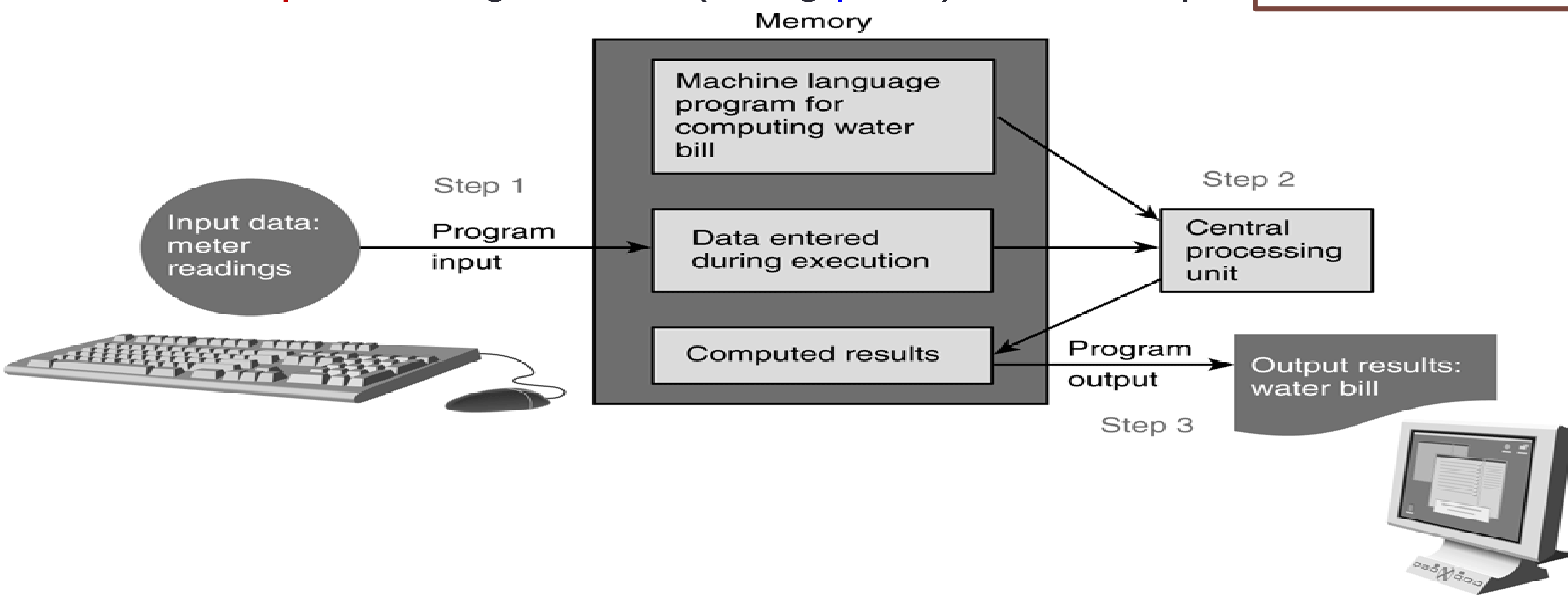
- Undetected errors
  - Exist if we do not test the program thoroughly enough

The process of correcting errors in programs is called debugging.
This process can be very time-consuming!

Program Structure

- A basic C program has 4 main parts:
  - Preprocessor directives:
    - eg: #include <stdio.h>, #include <math.h>, #define PI 3.142
  - Input: through stdin (using scanf), or file input
  - Compute: through arithmetic operations
  - Output: through stdout (using printf), or file output

We will learn file input/output later.



Program Structure: Preprocessor Directives (1/2)

Preprocessor
Input
Compute
Output

- The C preprocessor provides the following
  - Inclusion of header files
  - Macro expansions
  - Conditional compilation
  - For now, we will focus on inclusion of header files and simple application of macro expansions
- Inclusion of header files
  - To use input/output functions such as scanf() and printf(), you need to include <stdio.h>: `#include <stdio.h>`
  - To use mathematical functions, you need to include <math.h>: `#include <math.h>`

Program Structure: Preprocessor Directives (2/2)

Preprocessor
Input
Compute
Output

- Macro expansions
  - One of the uses is to define a macro for a constant value
  - Eg: `#define PI 3.142 // use all CAP for macro`

```
#define PI 3.142

int main(void) {
   ...
   areaCircle = PI * radius * radius;
   volCone = PI * radius * radius * height / 3.0;
}
```

Preprocessor replaces all instances of PI with 3.142 before passing the program to the compiler.

What the compiler sees:

```
int main(void) {
   ...
   areaCircle = 3.142 * radius * radius;
   volCone = 3.142 * radius * radius * height / 3.0;
}
```

Preprocessor
Input
Compute
Output

Program Structure: Input/Output (1/3)

- Input/output statements:
  - printf (format string, print list);
  - printf (format string);
  - scanf(format string, input list);

age

age
20

Address of variable 'age' varies each time a program is run.

"age" refers to value in the variable age.
"&age" refers to (address of) the memory cell where the value of age is stored.

One version:

```
int age;
double cap; // cumulative average point
printf("What is your age? ");
scanf("%d", &age);
printf("What is your CAP? ");
scanf("%lf", &cap);
printf("You are %d years old, and your CAP is %f\n", age, cap);
```

Unit3_InputOutput.c

Another version:

```
int age;
double cap; // cumulative average point
printf("What are your age and CAP? ");
scanf("%d %lf", &age, &cap);
printf("You are %d years old, and your CAP is %f\n", age, cap);
```

Unit3_InputOutputV2.c

Preprocessor
Input
Compute
Output

Program Structure: Input/Output (2/3)

- %d and %lf are examples of format specifiers; they are placeholders for values to be displayed or read

Placeholder	Variable Type	Function Use
%c	char	printf / scanf
%d	int	printf / scanf
%f	float or double	printf
%f	float	scanf
%lf	double	scanf
%e	float or double	printf (for scientific notation)

- Examples of format specifiers used in printf():
  - %5d: to display an integer in a width of 5, right justified
  - %8.3f: to display a real number (float or double) in a width of 8, with 3 decimal places, right justified
- See Table 2.3 (page 65) for sample displays
- Note: For scanf(), just use the format specifier <u>without</u> indicating width, decimal places, etc.

Preprocessor
Input
Compute
Output

Program Structure: Input/Output (3/3)

- \n is an example of escape sequence
- Escape sequences are used in printf() function for certain special effects or to display certain characters properly
- See Table 1.4 (pages 32 – 33)
- These are the more commonly used escape sequences:

Escape sequence	Meaning	Result
`\n`	New line	Subsequent output will appear on the next line
`\t`	Horizontal tab	Move to the next tab position on the current line
`\"`	Double quote	Display a double quote "
`%%`	Percent	Display a percent character %

Note the error in Table 1.4. It should be `%%` and not `\%`

Exercise #3: Distance Conversion (1/2)

- Convert distance from miles to kilometres
  - Unit3_MileToKm.c
  - The program is given (which you can copy to your directory as earlier instructed), but for this exercise we want you to type in the program yourself as a practice in using vim
  - The program is shown in the next slide

Exercise #3: Distance Conversion (2/2)

Unit3_MileToKm.c

```
// Unit3_MileToKm.c
// Converts distance in miles to kilometers.
#include <stdio.h>
#define KMS_PER_MILE 1.609

int main(void) {
   float miles,  // input - distance in miles.
         kms;    // output - distance in kilometers

   /* Get the distance in miles */
   printf("Enter distance in miles: ");
   scanf("%f", &miles);

   // Convert the distance to kilometres
   kms = KMS_PER_MILE * miles;

   // Display the distance in kilometres
   printf("That equals %9.2f km.\n", kms);

   return 0;
}
```

Preprocessor
Input
Compute
Output

Program Structure: Compute (1/9)

- Computation is through function
  - So far, we have used one function: int main(void)

    main() function: where execution of program begins
- A function body has two parts
  - Declarations statements: tell compiler what type of memory cells needed
  - Executable statements: describe the processing on the memory cells

```
int main(void) {
    /* declaration statements */
    /* executable statements */
    return 0;
}
```

Program Structure: Compute (2/9)

- Declaration Statements: To declare use of variables

```
int count, value;
```

Data type

Names of variables

- User-defined Identifier
  - Name of a variable or function
  - May consist of letters (a-z, A-Z), digits (0-9) and underscores (_), but MUST NOT begin with a digit
  - Case sensitive, i.e. count and Count are two distinct identifiers
  - Guideline: Usually should begin with lowercase letter
  - Must not be reserved words (next slide)
  - Should avoid standard identifiers (next slide)
  - Eg: *Valid identifiers:* maxEntries, _X123, this_IS_a_long_name
    Invalid: 1Letter, double, return, joe's, ice cream, T*S

Preprocessor
Input
Compute
Output

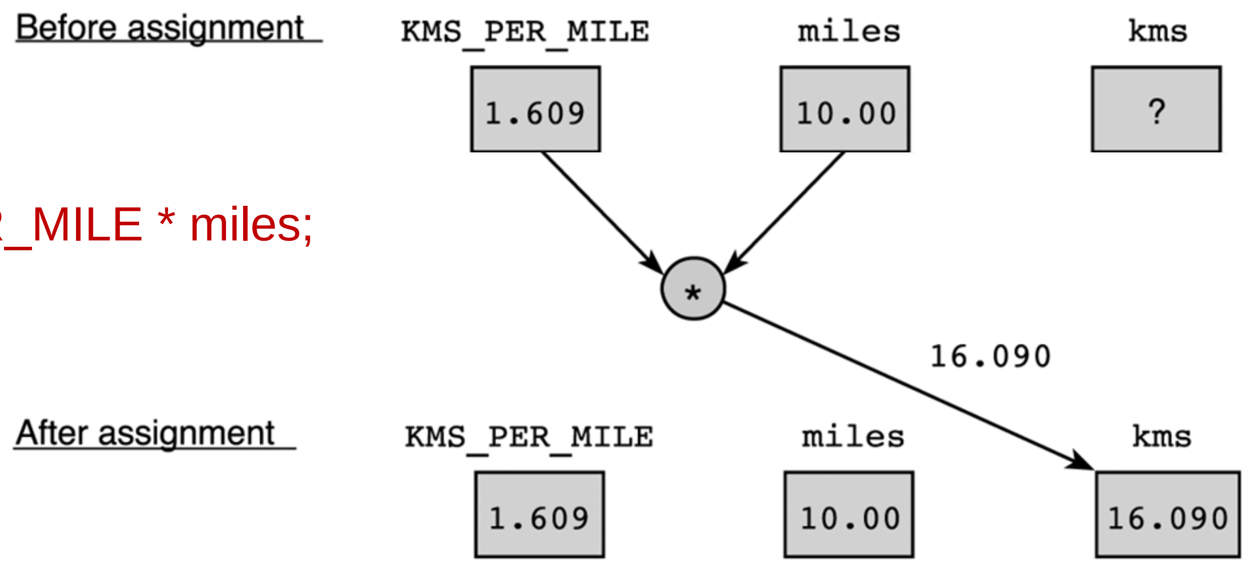
Program Structure: Compute (3/9)

- Reserved words (or keywords)
  - Have special meaning in C
  - Eg: int, void, double, return
  - Complete list: http://c.ihypress.ca/reserved.html
  - Cannot be used for user-defined identifiers (names of variables or functions)
- Standard identifiers
  - Names of common functions, such as printf, scanf
  - Avoid naming your variables/functions with the same name of built-in functions you intend to use

Program Structure: Compute (4/9)

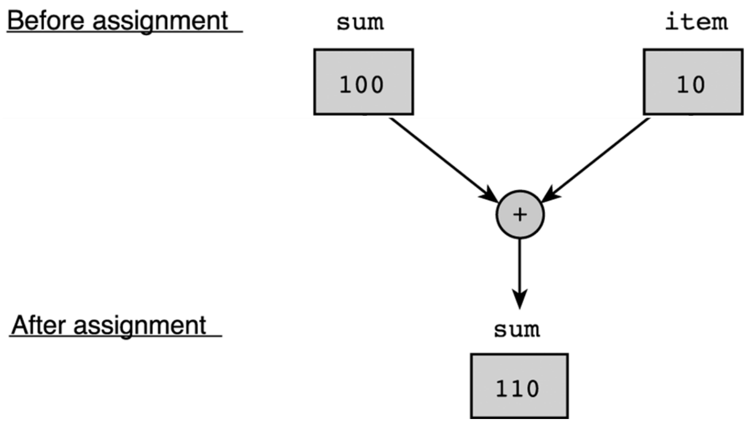
- Executable statements
  - I/O statements (eg: printf, scanf)
  - Computational and assignment statements
- Assignment statements
  - Store a value or a computational result in a variable
  - (Note: '=' means **'assign value on its right to the variable on its left'**; it does NOT mean equality)
  - Left side of '=' is called lvalue

Eg: kms = KMS_PER_MILE * miles;



Program Structure: Compute (5/9)

Eg: sum = sum + item;



- Note: lvalue must be assignable

- Examples of invalid assignment (result in compilation error "lvalue required as left operand of assignment"):
  - `32 = a; // '32' is not a variable`
  - `a + b = c; // 'a + b' is an expression, not variable`

- Assignment can be cascaded, with associativity from right to left:
  - `a = b = c = 3 + 6;` // 9 assigned to variables c, b and a
  - The above is equivalent to: `a = (b = (c = 3 + 6));`
    which is also equivalent to:

```
c = 3 + 6;
b = c;
a = b;
```

Program Structure: Compute (6/9)

- Side Effect:
  - An assignment statement does not just assigns, it also has the side effect of returning the value of its right-hand side expression
  - Hence `a = 12;` has the side effect of returning the value of 12, besides assigning 12 to `a`
  - Usually we don't make use of its side effect, but sometimes we do, eg:

    ```
    z = a = 12;  // or z = (a = 12);
    ```

  - The above makes use of the side effect of the assignment statement `a = 12;` (which returns 12) and assigns it to `z`
  - Side effects have their use, but avoid convoluted codes:

    ```
    a = 5 + (b = 10);  // assign 10 to b, and 15 to a
    ```

  - Side effects also apply to expressions involving other operators (eg: logical operators). We will see more of this later.

Preprocessor
Input
Compute
Output

Program Structure: Compute (7/9)

- Arithmetic operations
  - **Binary Operators**: +, –, *, /, % (modulo or remainder)
    - Left Associative (from left to right)
      - 46 / 15 / 2 → 3 / 2 → 1
      - 19 % 7 % 3 → 5 % 3 → 2
  - **Unary operators**: +, –
    - Right Associative
      - x = – 23 p = +4 * 10
  - Execution from left to right, respecting parentheses rule, and then precedence rule, and then associative rule (next page)
    - addition, subtraction are lower in precedence than multiplication, division, and remainder
  - Truncated result if result can't be stored (the page after next)
    - int n; n = 9 * 0.5; results in 4 being stored in n.

Try out **Unit3_ArithOps.c**

Program Structure: Compute (8/9)

Preprocessor
Input
Compute
Output

- Arithmetic operators: Associativity & Precedence

Operator Type	Operator	Associativity
Primary expression operators	( ) expr++ expr--	L to R
Unary operators	* & + - ++expr --expr (typecast)	R to L
Binary operators	* / %	L to R
	+ -	
Assignment operators	= += -= *= /= %=	R to L

Preprocessor
Input
Compute
Output

Program Structure: Compute (9/9)

- Mixed-Type Arithmetic Operations

```
int   m = 10/4;       means   m = 2;
float p = 10/4;       means   p = 2.0;
int   n = 10/4.0;     means   n = 2;
float q = 10/4.0;     means   q = 2.5;
int   r = -10/4.0;    means   r = -2;
```

Caution!

- Type Casting
  - Use a cast operator to change the type of an expression
    - syntax: (*type*) expression

```
int aa = 6; float ff = 15.8;
float pp = (float) aa / 4;     means  pp = 1.5;
int   nn = (int) ff / aa;      means  nn = 2;
float qq = (float) (aa / 4);   means  qq = 1.0;
```

Try out **Unit3_MixedTypes.c** and **Unit3_TypeCast.c**

Exercise #4: Temperature Conversion

- Instructions will be given out in class
- We will use this formula

$$celsius = \frac{5}{9} \times (fahrenheit - 32)$$

Exercise #5: Freezer (1/2)

- Write a program freezer.c that estimates the temperature in a freezer (in °C) given the elapsed time (hours) since a power failure. Assume this temperature (*T*) is given by

$$T = \frac{4t^2}{t+2} - 20$$

where *t* is the time since the power failure.

- Your program should prompt the user to enter how long it has been since the start of the power failure in hours and minutes, both values in integers.
- Note that you need to convert the elapsed time into hours in real number (use type float)
  - For example, if the user entered 2 30 (2 hours 30 minutes), you need to convert this to 2.5 hours before applying the above formula.

Exercise #5: Freezer (2/2)

- Refer to the sample run below. Follow the output format.

```
Enter hours and minutes since power failure: 2 45
Temperature in freezer = -13.63
```

- How long does it take the freezer to get to zero degree? Which of the following is the closest answer?
  a) 3 hours
  b) 4 hours 10 minutes
  c) 6 hours 30 minutes
  d) 8 hours

- This exercise is mounted on CodeCrunch as a practice exercise.

Math Functions (1/2)

- In C, there are many libraries offering functions for you to use.
- Eg: scanf() and printf() – requires to include <stdio.h>
- In Exercise #5, for t^2 you may use t*t, or the pow() function in the math library: pow(t, 2)
  - pow(x, y) // computes x raised to the power of y
- To use math functions, you need to
  - Include <math.h> AND
  - Compile your program with –lm option (i.e. gcc –lm …)
- See Tables 3.3 and 3.4 (pages 88 – 89) for some math functions

Math Functions (2/2)

- Some useful math functions
  - Function abs(x) from <stdlib.h>; the rest from <math.h>

Function	Arguments	Result
abs(x)	int	int
ceil(x)	double	double
cos(x)	double (radians)	double
exp(x)	double	double
fabs(x)	double	double
floor(x)	double	double
log(x)	double	double
log10(x)	double	double
ceil(x)	double	double
pow(x, y)	double, double	double
sin(x)	double (radians)	double
sqrt(x)	double	double
tan(x)	double (radians)	double

Function prototype:

double pow(double x, double y)

function return type

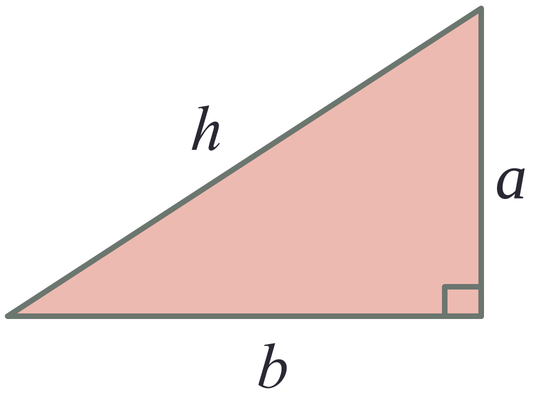
Q: Since the parameters x and y in pow() function are of double type, why can we call the function with pow(t, 2)?

A: Integer value can be assigned to a double variable/parameter.

Math Functions: Example (1/2)

- Program Unit3_Hypotenuse.c computes the hypotenuse of a right-angled triangle given the lengths of its two perpendicular sides

$$h = \sqrt{(a^2 + b^2)}$$



Math Functions: Example (2/2)

Unit3_Hypotenuse.c

Remember to compile with –lm option!

```
// Unit3_Hypotenuse.c
// Compute the hypotenuse of a right-angled triangle.
#include <stdio.h>
#include <math.h>

int main(void) {
  float hypot, side1, side2;

  printf("Enter lengths of the 2 perpendicular sides: ");
  scanf("%f %f", &side1, side2);

  hypot = sqrt(side1*side1 + side2*side2);
  // or hypot = sqrt(pow(side1, 2) + pow(side2, 2));

  printf("Hypotenuse = %6.2f\n", hypot);

  return 0;
}
```

Exercise #6: Freezer (version 2)

- Instructions will be given out in class

Programming Style

- Identifier naming for variables and functions
  - Use lower-case with underscore or capitalise first character of every subsequent word (Eg: celsius, sum, second_max, secondMax; NOT Celsius, SUM, SecondMax)
  - Must be descriptive (Eg: numYears instead of ny, abc, xbrt)
- User-defined constants
  - Use upper-case with underscore (Eg: KMS_PER_MILE, DAYS_IN_YEAR)
- Consistent indentation
- Appropriate comments
- Spacing and blank lines
- And many others

In vim, typing

gg=G

would auto-indent your program nicely!

Common Mistakes (1/2)

- Not initialising variables

EXTREMELY COMMON MISTAKE

  - Program may work on some machine but not on another!

Cannot assume that the initial value of b is zero!

```
int a, b;
a = b + 3; // but what is the value of b?
```

- Unnecessary initialisation of variables

```
int x = 0;
x = 531;
```

```
int x = 0;
scanf("%d", &x);
```

- Forgetting & in a scanf() statement

```
int x;
scanf("%d", x);
```

✗

```
int x;
scanf("%d", &x);
```

✓

Common Mistakes (2/2)

- Forgetting to compile with –lm option when the program uses math functions.
- Forgetting to recompile after modifying the source code.

Sometimes when your program crashes, a "core dump" may happen. Remove the file "core" (UNIX command: rm core) from your directory as it takes up a lot of space.

Summary

- In this unit, you have learned about
  - The use of variables in a program and the basic data types in C
  - The basic structure of a simple C program which includes: preprocessor directives, input statements, computation, and output statements.
  - Using Math functions
  - Good programming style
  - Common mistakes made by beginners

End of File