CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #2

Overview of C Programming

NUS National University of Singapore | School of Computing

Lecture #2: Overview of C Programming (1/2)

1. A Simple C Program
2. von Neumann Architecture
3. Variables
4. Data Types
5. Program Structure
   - 5.1 Preprocessor Directives
   - 5.2 Input/Output
   - 5.3 Compute
     - Arithmetic operators
     - Assignment statements
     - Typecast operator

Lecture #2: Overview of C Programming (2/2)

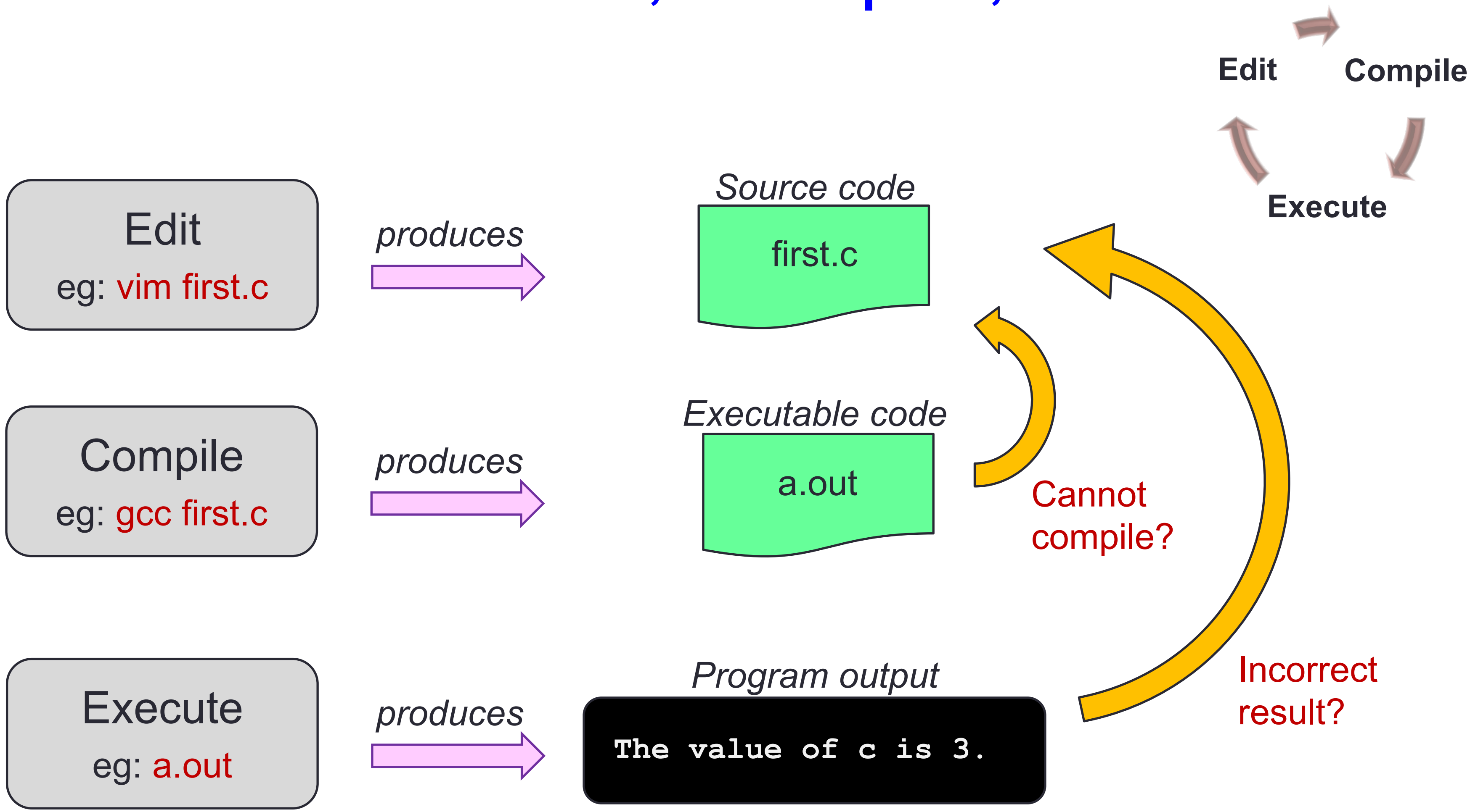
6. Selection Statements
   - 6.1 Condition and Relational Operators
   - 6.2 Truth Values
   - 6.3 Logical Operators
   - 6.4 Evaluation of Boolean Expressions
   - 6.5 Short-Circuit Evaluation
7. Repetition Statements
   - 7.1 Using ‘break’ in a loop
   - 7.2 Using ‘continue’ in a loop

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



1. A Simple C Program (1/3)

General form

preprocessor directives

main function header
{
    declaration of variables
    executable statements
}

"Executable statements" usually consists of 3 parts:

- Input data
- Computation
- Output results

1. A Simple C Program (2/3)

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 MileToKm.c
$ a.out
Enter distance in miles: 10.5
That equals     16.89 km.
```

(Note: All C programs in the lectures are available on LumiNUS as well as the CS2100 website. Python versions are also available.)

1. A Simple C Program (3/3)

```
// 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

(Only /* … */ is ANSI C)

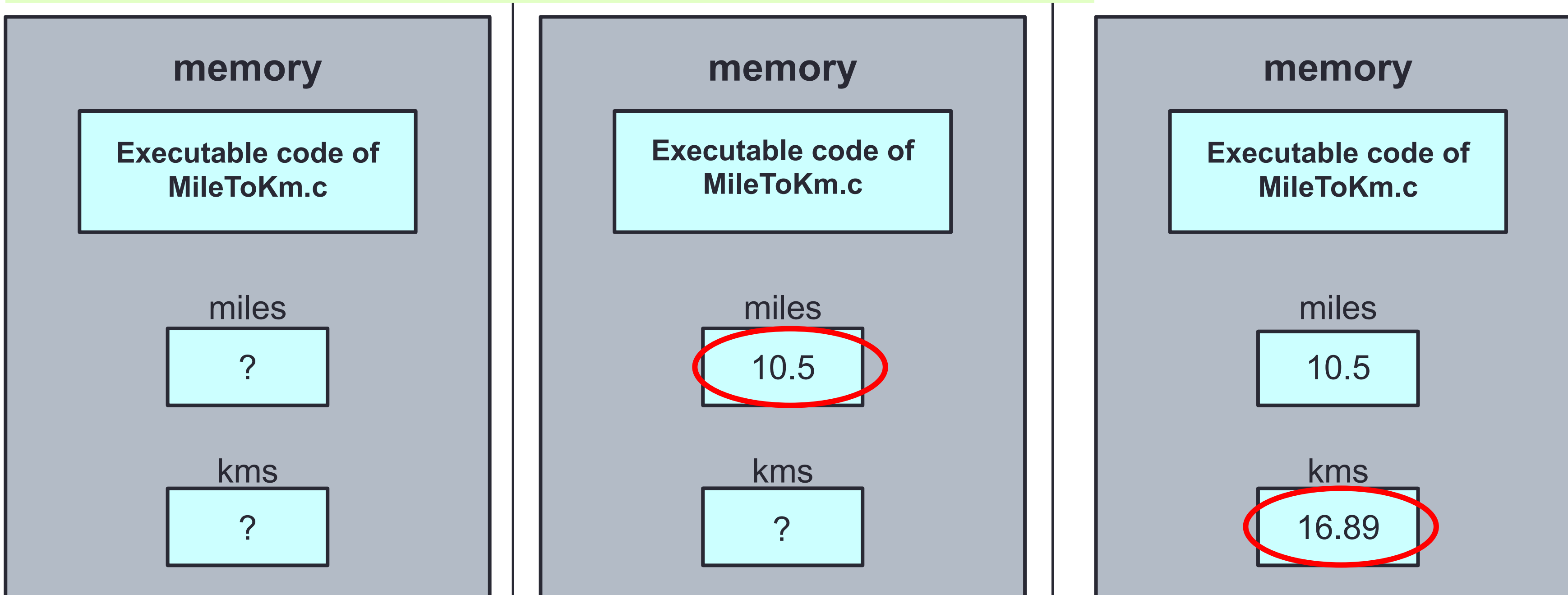
functions

special symbols

In C, semi-colon (;) terminates a statement.
Curly bracket { } indicates a block.
In Python: block is by indentation

2. von Neumann Architecture (1/2)

What happens in the computer memory?



At the beginning

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

After user enters: 10.5 to

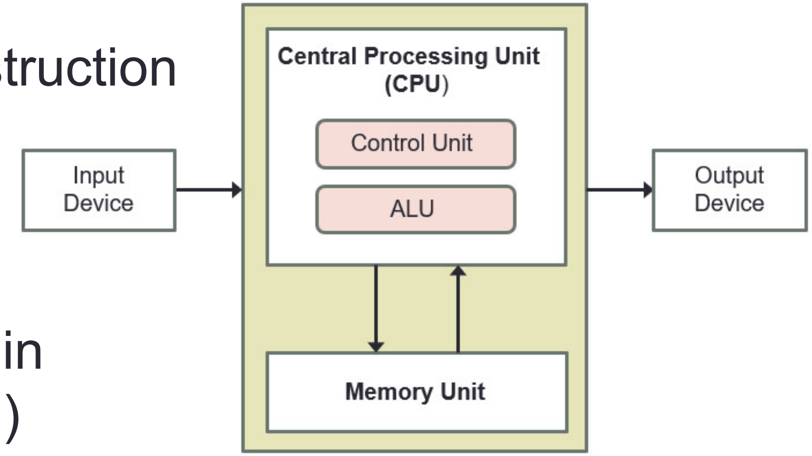
```
scanf("%f", &miles);
```

After this line is executed:

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

2. von Neumann Architecture (2/2)

- John von Neumann (1903 – 1957)
- von Neumann architecture* describes a computer consisting of:
  - Central Processing Unit (CPU)
    - Registers
    - A control unit containing an instruction register and program counter
    - An arithmetic/logic unit (ALU)
  - Memory
    - Stores both program and data in random-access memory (RAM)
  - I/O devices



(* Also called *Princeton architecture*, or *stored-program architecture*)

3. Variables

```
float miles, kms ;
```

- 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
- (Each variable actually has an address too, but for the moment we will skip this until we discuss pointers.)
- A variable is declared with a data type
  - `int count; // variable 'count' of type 'int'`
- Variables may be initialized during declaration:
  - `int count = 3; // 'count' is initialized to 3`
- Without initialization, the variable contains an unknown value (Cannot assume that it is zero!)

Python

Declaration via assignment in function/global

```
count = 3
```

3. Variables: Mistakes in Initialization

- No initialization

  -Wall option turns on all warnings

InitVariable.c

```
int main(void) {
    int count;
    count = count + 12;
    return 0;
}
```

Python

Cannot declare without initialization

```
$ gcc -Wall InitVariable.c
InitVariable.c: In function 'main':
InitVariable.c:3:8: warning: 'count' is used
uninitialized in this function
  count = count + 12;
        ^
```

- Redundant initialization

```
int count = 0;
count = 123;
```

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

4. Data Types (1/3)

```
float miles, kms;
```

- Every variable must be declared with a data type
  - To determine the type of data the variable may hold
- Basic data types in C:
  - 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 characters
    - Enclosed in a pair of single quotes
    - Eg: `'A'`, `'z'`, `'2'`, `'*'`, `' '`, `'\n'`

Python	JS
`int`	`number`
`float`	`number`
`str`	`string`

4. Data Types (2/3)

- A programming language can be strongly typed or weakly typed
  - Strongly typed: every variable to be declared with a data type. (C: `int count; char grade;`)
  - Weakly typed: the type depends on how the variable is used (JavaScript: `var count; var grade;`)
  - The above is just a simple explanation.
    - Much subtleties and many views and even different definitions. Other aspects include static/dynamic type checking, safe type checking, type conversions, etc.
    - Eg: Java, Pascal and C are strongly typed languages. But Java /Pascal are more strongly typed than C, as C supports implicit type conversions and allows pointer values to be explicitly cast.
    - One fun video: https://www.youtube.com/watch?v=bQdzwJWYZRU

4. Data Types (3/3)

DataTypes.c

```
// This program checks the memory size
// of each of the basic data types
#include <stdio.h>

int main(void) {
   printf("Size of 'int' (in bytes): %d\n", sizeof(int));
   printf("Size of 'float' (in bytes): %d\n", sizeof(float));
   printf("Size of 'double' (in bytes): %d\n", sizeof(double));
   printf("Size of 'char' (in bytes): %d\n", sizeof(char));

   return 0;
}
```

Python

Use sys.getsizeof

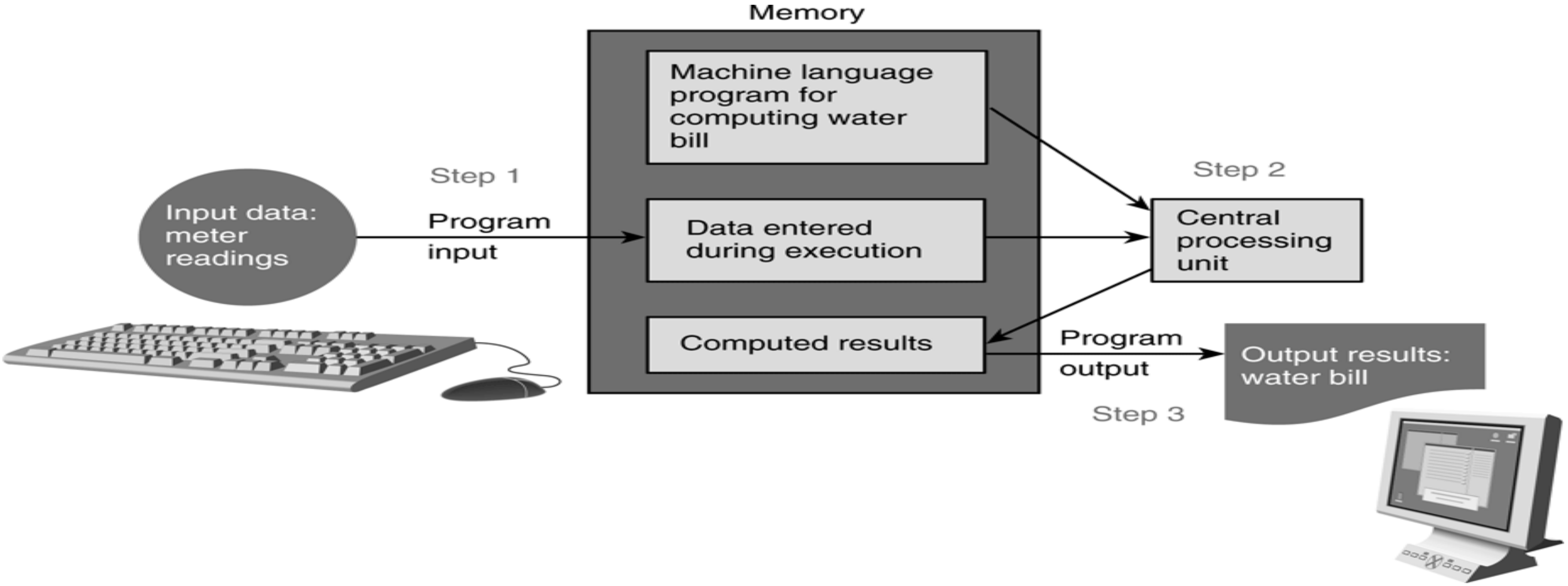
```
import sys
sys.getsizeof(1)
```

```
$ gcc DataTypes.c -o DataTypes
$ DataTypes
Size of 'int' (in bytes): 4
Size of 'float' (in bytes): 4
Size of 'double' (in bytes): 8
Size of 'char' (in bytes): 1
```

-o option specifies name of executable file (default is ‘a.out’)

5. 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 and assignment statements
  - Output: through stdout (using printf), or file output



5.1 Preprocessor Directives (1/2)

Preprocessor
Input
Compute
Output

- The C preprocessor provides the following
  - Inclusion of header files
  - Macro expansions
  - Conditional compilation
  - We will focus on inclusion of header files and simple application of macro expansions (defining constants)
- 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 functions from certain libraries, you need to include the respective header file, examples:
    - To use mathematical functions, `#include <math.h>` (In sunfire, need to compile with –lm option)
    - To use string functions, `#include <string.h>`

5.1 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.

In Python, there is no parallel, but closest is simply declare global variable

```
PI = 3.142
areaCircle = PI * radius * radius
volCone = PI * radius * height / 3.0
```

What the compiler sees:

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

5.2 Input/Output (1/3)

Preprocessor
Input
Compute
Output

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

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 averag
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);
```

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);
```

InputOutputV2.c

5.2 Input/Output (2/3)

Preprocessor
Input
Compute
Output

- %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)

Python

All inputs are read as `string`

- 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
- Note: For scanf(), just use the format specifier without indicating width, decimal places, etc.

5.2 Input/Output (3/3)

Preprocessor
Input
Compute
Output

- \n is an example of escape sequence
- Escape sequences are used in printf() function for certain special effects or to display certain characters properly
- 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 %

Try out TestIO.c and compare with TestIO.py

5.3 Compute (1/10)

Preprocessor
Input
Compute
Output

- 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

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

Python

```python
def main():
  # statements
  return 0
if __name__ == "__main__":
  main()
```

5.3 Compute (2/10)

Preprocessor
Input
Compute
Output

- 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`

5.3 Compute (3/10)

Preprocessor
Input
Compute
Output

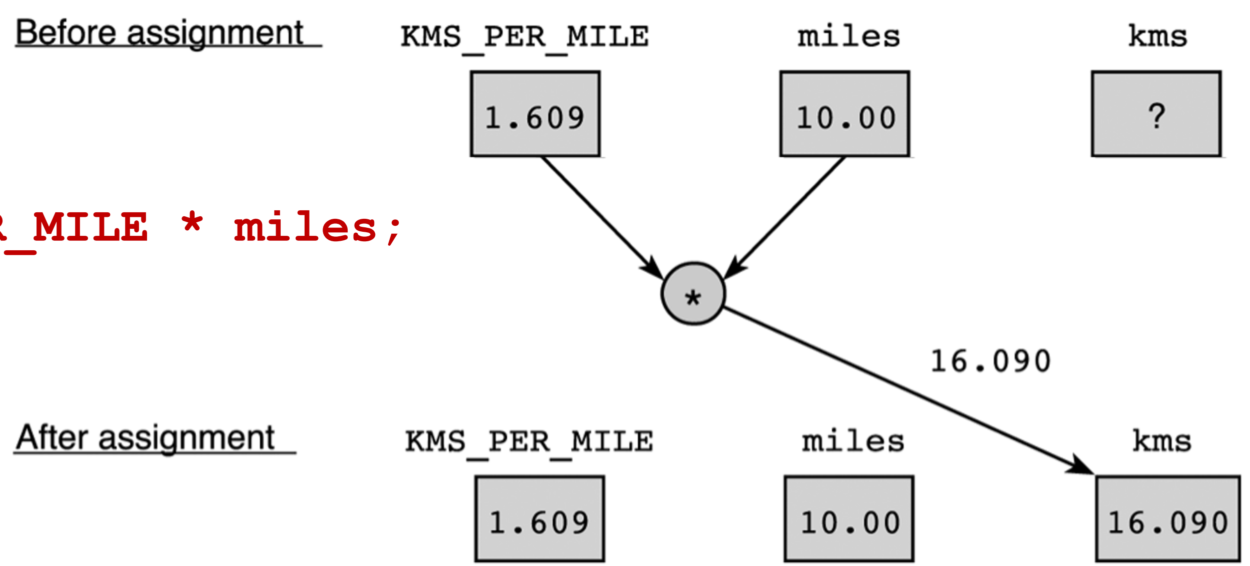
- Reserved words (or keywords)
  - Have special meaning in C
  - Eg: `int`, `void`, `double`, `return`
  - Complete list: http://c.ihypress.ca/reserved.html
  - Cannot be used for user-defined identifiers (names of variables or functions)
- Standard identifiers
  - Names of common functions, such as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

5.3 Compute (4/10)

Preprocessor
Input
Compute
Output

- 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

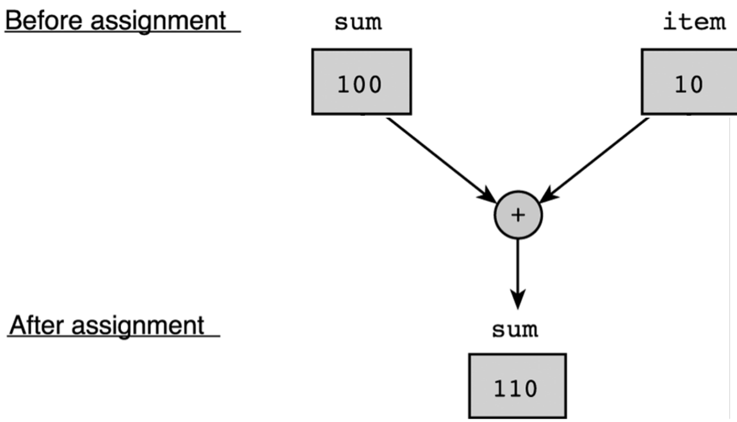
Eg: `kms = KMS_PER_MILE * miles;`



5.3 Compute (5/10)

Preprocessor
Input
Compute
Output

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;
```

Python

Can write: `a = b = c = 3 + 6`

CANNOT: `a = 5 + (b = 3)`

5.3 Compute (6/10)

Preprocessor
Input
Compute
Output

- 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.

5.3 Compute (7/10)

Preprocessor
Input
Compute
Output

- Arithmetic operations
  - Binary Operators: +, –, *, /, % (*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 (slide 30)
    - addition, subtraction are lower in precedence than multiplication, division, and remainder
  - Truncate result if result can't be stored (slide 31)
    - `int n;  n = 9 * 0.5;`   results in 4 being stored in n.

5.3 Compute (8/10)

Preprocessor
Input
Compute
Output

ArithOps.c

```c
// To illustrate some arithmetic operations in C
#include <stdio.h>
int main(void) {
   int x, p, n;

   // to show left associativity
   printf("46 / 15 / 2 = %d\n", 46/15/2);
   printf("19 %% 7 %% 3 = %d\n", 19%7%3);

   // to show right associativity
   x = -23;
   p = +4 * 10;
   printf("x = %d\n", x);
   printf("p = %d\n", p);

   // to show truncation of va
   n = 9 * 0.5;
   printf("n = %d\n", n);

   return 0;
}
```

```
$ gcc ArithOps.c -o ArithOps
$ ArithOps
46 / 15 / 2 = 1
19 % 7 % 3 = 2
x = -23
p = 40
n = 4
```

5.3 Compute (9/10)

Preprocessor
Input
Compute
Output

- Arithmetic operators: Associativity & Precedence

Operator Type	Operator	Associativity
Primary expression operators	`( )` *expr*`++` *expr*`--`	Left to right
Unary operators	`* & + -` `++`*expr* `--`*expr* `(`*typecast*`)`	Right to left
Binary operators	`* / %`	Left to right
	`+ -`	
Assignment operators	`= += -= *= /= %=`	Right to left

Python

`expr++`, `expr--`, `++expr`, `--expr` are not available

5.3 Compute (10/10)

Preprocessor
Input
Compute
Output

- 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 TypeCast.c

5.3 Compute: Difference with Python

- Python Floor Division

`a = 10/4`	means	`a = 2.5`
`b = 10//4`	means	`b = 2`
`c = -10/4`	means	`c = -2.5`
`d = -10//4`	means	`d = -3`

- Modulo
  - Python % is modulo

    `a = 10%4` → `a = 2`

    `b = -10%4` → `b = 2`
  - C % is remainder

    `a = 10%4` → `a = 2`

    `b = -10%4` → `b = -2`
  - NOTE: be careful with negative values for % operation

Try out Modulo.c and compare with Modulo.py

6. Selection Structures (1/2)

- C provides two control structures that allow you to select a group of statements to be executed or skipped when certain conditions are met.

if … else …

```
if (condition) {
   /* Execute these statements if TRUE */
}
```

```
if (condition) {
   /* Execute these statements if TRUE  */
}
else {
   /* Execute these statements if FALSE */
}
```

```
if condition:
   # Statement
```

```
if condition:
   # Statement
elif condition:
   # Statement
else:
   # Statement
```

6. Selection Structures (2/2)

switch

Python

No counterpart

```
/* variable or expression must be of discrete type */
switch ( <variable or expression> ) {
   case value1:
       Code to execute if <variable or expr> == value1
       break;

   case value2:
       Code to execute if <variable or expr> == value2
       break;
   ...

   default:
       Code to execute if <variable or expr> does not
       equal to the value of any of the cases above
       break;
}
```

6.1 Condition and Relational Operators

- A condition is an expression evaluated to `true` or `false`.
- It is composed of expressions combined with relational operators.
  - Examples: `(a <= 10)`, `(count > max)`, `(value != -9)`

Relational Operator	Interpretation
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Python

Allows
`1 <= x <= 5`

6.2 Truth Values

- Boolean values: true or false.
- There is no Boolean type in ANSI C. Instead, we use **integers**:
  - 0 to represent false
  - Any other value to represent true (1 is used as the representative value for true in output)
- Example:

Python

NOTE*: only integers!*
In Python and JavaScript you have truthy and falsy values, but not in C

TruthValues.c

```c
int a = (2 > 3);
int b = (3 > 2);

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

```
a = 0; b = 1
```

6.3 Logical Operators

- Complex condition: combining two or more Boolean expressions.
- Examples:
  - If temperature is greater than 40C or blood pressure is greater than 200, go to A&E immediately.
  - If all the three subject scores (English, Maths and Science) are greater than 85 and mother tongue score is at least 80, recommend taking Higher Mother Tongue.
- Logical operators are needed: && (and), || (or), ! (not).

A	B	A && B	A || B	!A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Python

```
A || B → A or B
A && B → A and B
!A → not A
```

6.4 Evaluation of Boolean Expressions (1/2)

- The evaluation of a Boolean expression is done according to the precedence and associativity of the operators.

Operator Type	Operator	Associativity
Primary expression operators	( ) [ ] . -> expr++ expr--	Left to Right
Unary operators	* & + - ! ~ ++expr --expr (typecast) sizeof	Right to Left
Binary operators	* / %	Left to Right
	+ -	
	< > <= >=	
	== !=	
	&&	
	||	
Ternary operator	?:	Right to Left
Assignment operators	= += -= *= /= %=	Right to Left

Python

```
cond ? expr1 : expr2 →
expr1 if cond else cond2
```

6.4 Evaluation of Boolean Expressions (2/2)

- What is the value of x?

```
int x, y, z,
    a = 4, b = -2, c = 0;
x = (a > b || b > c && a == b);
```

x is true (1)

gcc issues warning (why?)

- Always good to add parentheses for readability.

```
y = ((a > b || b > c) && a == b);
```

y is false (0)

- What is the value of z?

```
z = ((a > b) && !(b > c));
```

z is true (1)

Try out EvalBoolean.c

6.5 Short-Circuit Evaluation

- Does the following code give an error if variable a is zero?

```
if ((a != 0) && (b/a > 3)) {
    printf(. . .);
}
```

- Short-circuit evaluation
  - expr1 || expr2: If <u>expr1 is true</u>, skip evaluating expr2 and return true immediately, as the result will always be true.
  - expr1 && expr2: If <u>expr1 is false</u>, skip evaluating expr2 and return false immediately, as the result will always be false.

7. Repetition Structures (1/2)

- C provides three control structures that allow you to select a group of statements to be executed repeatedly.

```
while ( condition )
{
    // loop body
}
```

```
do
{
    // loop body
}  while ( condition );
```

```
for ( initialization; condition; update )
{
    // loop body
}
```

Initialization: initialize the **loop variable**

Condition: repeat loop while the condition on **loop variable** is true

Update: change value of **loop variable**

7. Repetition Structures (2/2)

- Example: Summing from 1 through 10.

Sum1To10_While.c

```
int sum = 0, i = 1;
while (i <= 10) {
    sum = sum + i;
    i++;
}
```

Sum1To10_DoWhile.c

```
int sum = 0, i = 1;
do {
    sum = sum + i;
    i++;
}
while (i <= 10);
```

Sum1To10_For.c

```
int sum, i;
for (sum = 0, i = 1; i <= 10; i++) {
    sum = sum + i;
}
```

7.1 Using 'break' in a loop (1/2)

BreakInLoop.c

```
// without 'break'
printf ("Without 'break':\n");
for (i=1; i<=5; i++) {
   printf("%d\n", i);
   printf("Ya\n");
}
```

```
// with 'break'
printf ("With 'break':\n");
for (i=1; i<=5; i++) {
   printf("%d\n", i);
   if (i==3)
      break;
   printf("Ya\n");
}
```

```
Without 'break':
1
Ya
2
Ya
3
Ya
4
Ya
5
Ya
```

```
With 'break':
1
Ya
2
Ya
3
```

7.1 Using 'break' in a loop (2/2)

BreakInLoop.c

```
// with 'break' in a nested loop
printf("With 'break' in a nested loop:\n");
for (i=1; i<=3; i++) {
   for (j=1; j<=5; j++) {
      printf("%d, %d\n", i, j);
      if (j==3)
         break;
      printf("Ya\n");
   }
}
```

```
With 'break' in …
1, 1
Ya
1, 2
Ya
1, 3
2, 1
Ya
2, 2
Ya
2, 3
3, 1
Ya
3, 2
Ya
3, 3
```

- In a nested loop, **break** only breaks out of the inner-most loop that contains the **break** statement.

7.2 Using 'continue' in a loop (1/2)

ContinueInLoop.c

```
// without 'continue'
printf ("Without 'continue':\n");
for (i=1; i<=5; i++) {
   printf("%d\n", i);
   printf("Ya\n");
}
```

```
Without 'continue':
1
Ya
2
Ya
3
Ya
4
Ya
5
Ya
```

```
// with 'continue'
printf ("With 'continue':\n");
for (i=1; i<=5; i++) {
   printf("%d\n", i);
   if (i==3)
      continue;
   printf("Ya\n");
}
```

```
With 'continue':
1
Ya
2
Ya
3
4
Ya
5
Ya
```

7.2 Using 'continue' in a loop (2/2)

ContinueInLoop.c

```
// with 'continue' in a nested loop
printf("With 'continue' in a nested loop:\n");
for (i=1; i<=3; i++) {
   for (j=1; j<=5; j++) {
      printf("%d, %d\n", i, j);
      if (j==3)
         continue;
      printf("Ya\n");
   }
}
```

```
With ...
1, 1
Ya
1, 2
Ya
1, 3
1, 4
Ya
1, 5
Ya
2, 1
Ya
2, 2
Ya
2, 3
2, 4
Ya
2, 5
Ya
```

```
3, 1
Ya
3, 2
Ya
3, 3
3, 4
Ya
3, 5
Ya
```

- In a nested loop, `continue` only skips to the next iteration of the inner-most loop that contains the `continue` statement.

End of File