Creative Software Design

C Overview

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Many of slides are taken from
https://www.comp.nus.edu.sg/~cs1010

Topics Covered

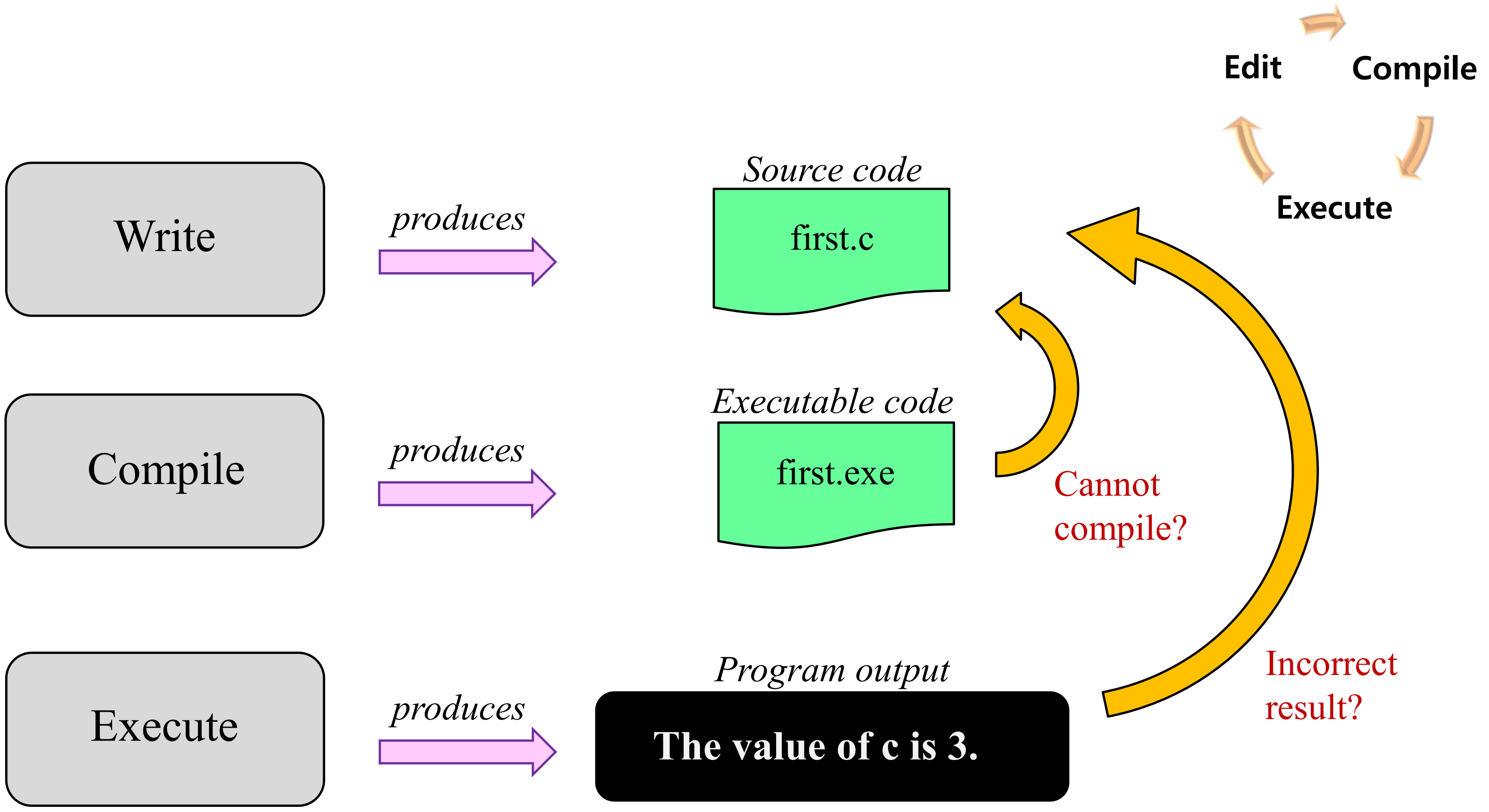
1. A Simple C Program
2. Variables and Data Types
3. Program Structure
   - Preprocessor directives
   - Input
   - Compute
   - Output
4. Pointers

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

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

Quick Review: Edit, Compile, Execute



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)

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) {
    double 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

```
Enter distance in miles: 10.5
That equals     16.89 km.
```

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) {
   double 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

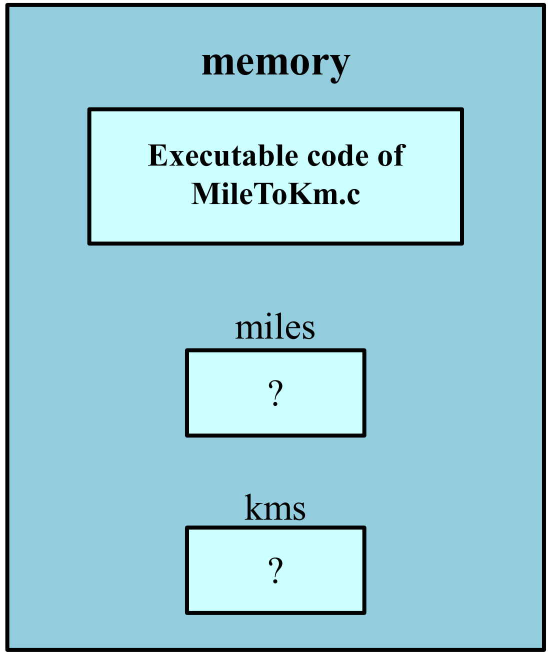
comments

functions

special symbols

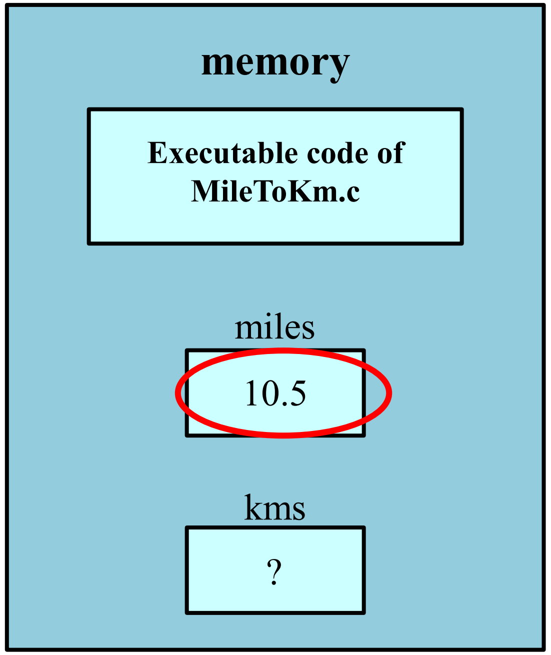
punctuations

What Happens in the Computer Memory



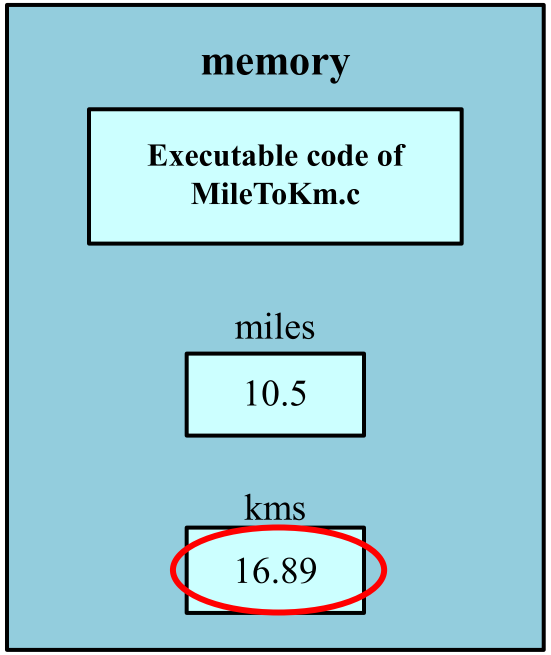
At the beginning

Do not assume that uninitialised variables contain zero! **(<u>Very</u> 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
  - Each variable actually has an address too
- 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

- 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'

https://en.cppreference.com/w/c/language/type

Type of Errors

- 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

Easiest to spot – the compiler helps you!

- 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

Moderately easy to spot

- Logic errors
  - Program produces incorrect result

Hard to spot

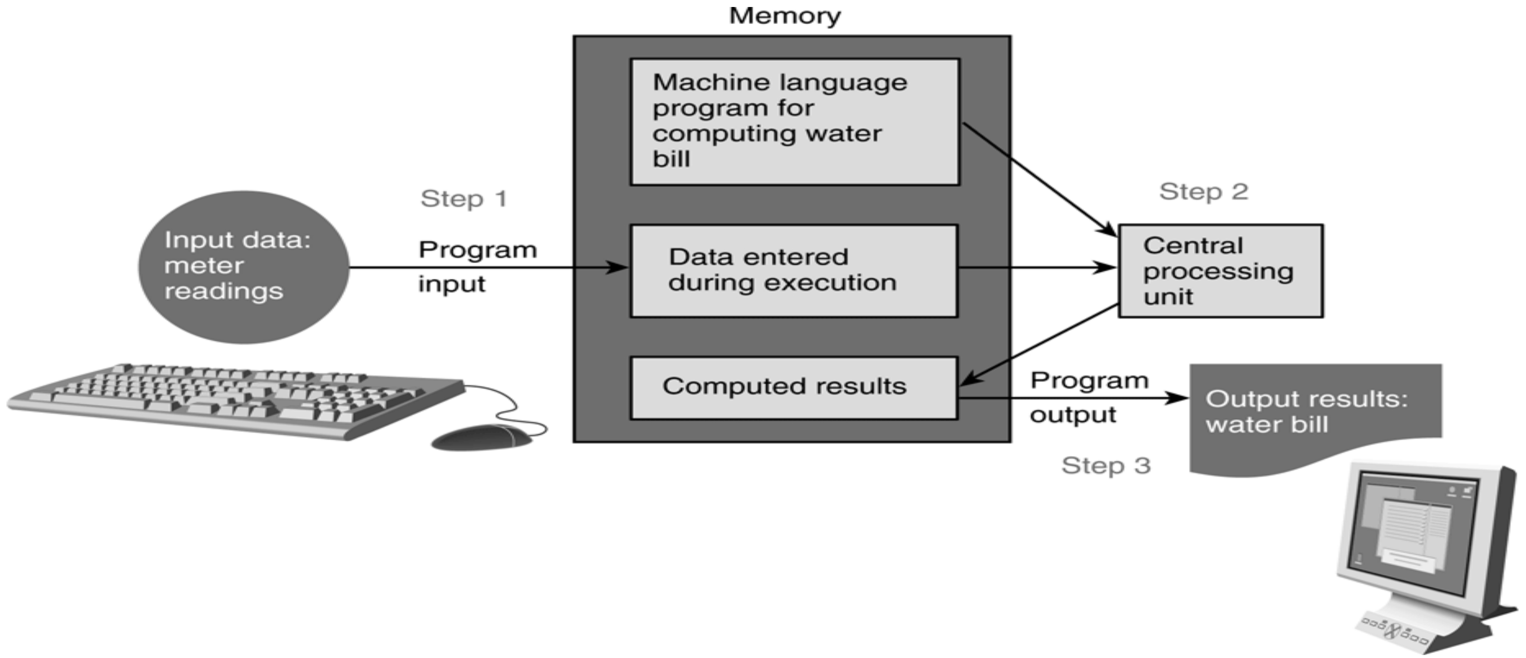
- Undetected errors
  - Exist if we do not test the program thoroughly enough

May never be spotted!

> 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



Program Structure: Preprocessor Directives (1/2)

Preprocessor
Input
Compute
Output

- The C preprocessor provides the following
  - Inclusion of header files
  - Macro expansions
  - Conditional compilation

- 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;
}
```

Program Structure: Input/Output (1/3)

Preprocessor
Input
Compute
Output

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

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);
```

Program Structure: 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)

- 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 <u>without</u> indicating width, decimal places, etc.

Program Structure: 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 %

Program Structure: Compute (1/9)

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

```
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

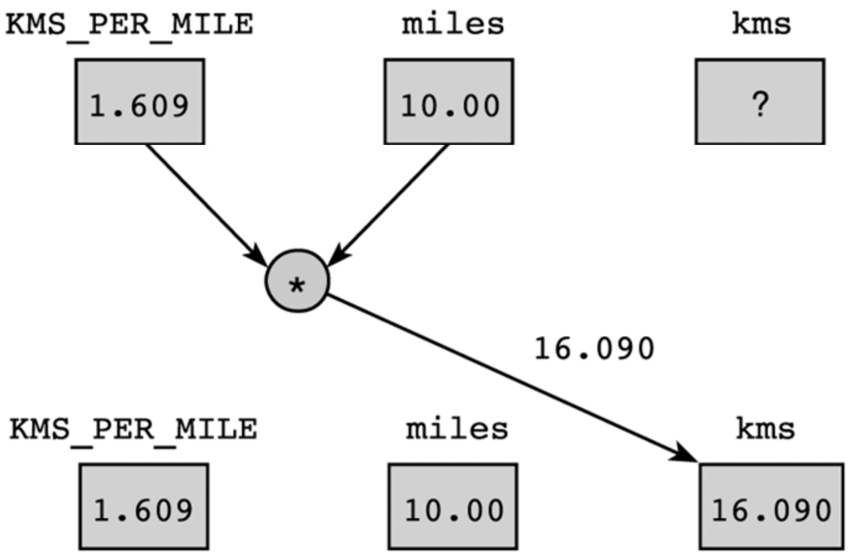
Program Structure: Compute (3/9)

- Reserved words (or keywords)
  - Have special meaning in C
  - Eg: int, void, double, return
  - Complete list: https://en.cppreference.com/w/c/keyword
  - 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

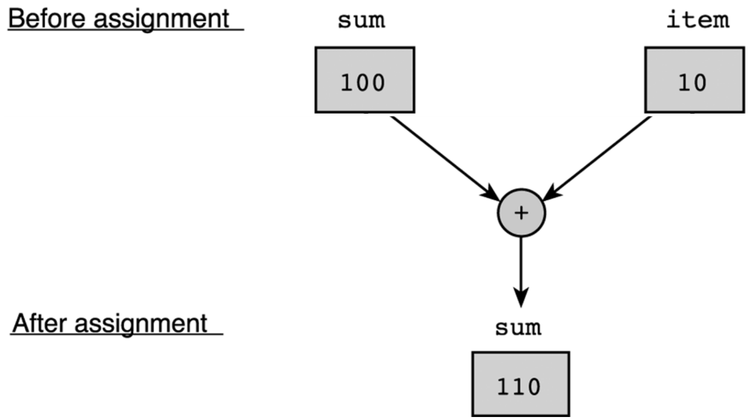
Eg: kms = KMS_PER_MILE * miles;



Program Structure: Compute (5/9)

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;
```

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.

Program Structure: Compute (7/9)

Preprocessor
Input
Compute
Output

- 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.

Program Structure: Compute (8/9)

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

Program Structure: Compute (9/9)

- Mixed-Type Arithmetic Operations

```
int    m = 10/4;       means  m = 2;
double p = 10/4;       means  p = 2.0;
int    n = 10/4.0;     means  n = 2;
double 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; double ff = 15.8;
double pp = (double) aa / 4;    means  pp = 1.5;
int    nn = (int) ff / aa;      means  nn = 2;
double qq = (double) (aa / 4);  means  qq = 1.0;
```

if - else if - else

Suppose that only condition3 is true

```
if(condition1)
{
    // condition1 is true
}
else if(condition2)
{
    // condition1 is false and conditions2 is true
}
else if(condition3)
{
    // only condition3 is true
}
else
{
    // all the conditions are false
}
```

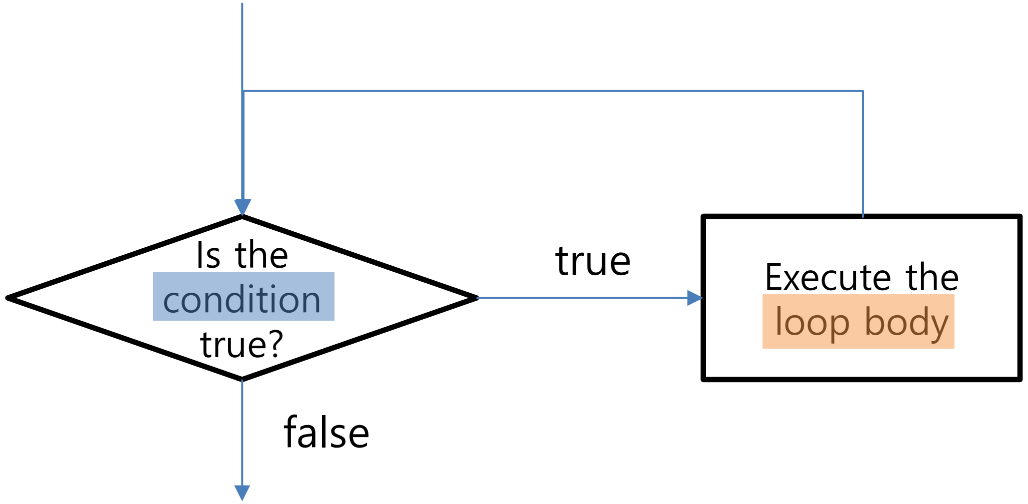
You can use
- only if
- if and else if
- if and else
- if, else if, and else

switch

```
switch(n)
{
case 1:        if(n==1)
    printf("1");
    printf("11");
    break;
case 2:        else if(n==2)
    printf("2");
    printf("22");
    break;
default:       else
    printf("default")
}
```

while

```
while(num < 5)
{
    printf("%d\n", num);
    num++;
}
```



for

```
      1)             2)       3)
for(int i=0; i<3; i++)
{
      printf("%d\n", i);
}     4)
```

1st iteration
1) → 2) → 4) → 3) : i==1

2nd iteration
2) → 4) → 3) : i==2

3rd iteration
2) → 4) → 3) : i==3

4th iteration
2)

- 1) initial clause: is executed once before the first evaluation of condition expression
- 2) condition expression: is evaluated before the loop body. If the result of the expression is zero (i.e., false), the loop statement is exited immediately
- 3) iteration expression: is executed after the loop body

break & continue

- break : exits the current loop immediately

```
int main(void)
{
    ...
    while(1)
    {
        if(x > 20)
            break;
        ...
    }
    ...
}
```

- continue : skips the remained loop body and jumps to the starting of the loop

```
int main(void)
{
    ...
    while(1)
    {
        if(x/2 == 1)
            continue;
        ...
    }
    ...
}
```

Variable and Its Address (1/2)

- A variable has a unique name (identifier) in the function it is declared in, it belongs to some data type, and it contains a value of that type

Data type Name

```
int a;
a = 123;
```

May only contain integer value

- A variable occupies some space in the memory, and hence it has an address
- The programmer usually does not need to know the address of the variable (she simply refers to the variable by its name), but the system keeps track of the variable's address

a

123

Where is variable a located in the memory?

Variable and Its Address (2/2)

- You may refer to the address of a variable by using the address operator: & (ampersand)

```
int a = 123;
printf("a = %d\n", a);
printf("&a = %p\n", &a);
```

```
a = 123
&a = ffbff7dc
```

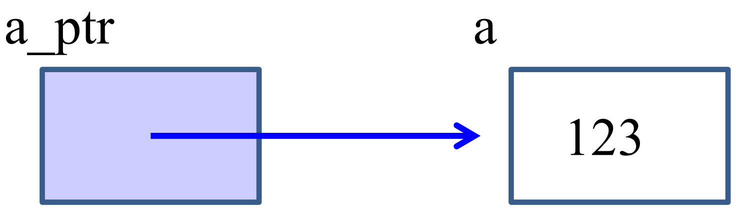
- %p is used as the format specifier for addresses
- Addresses are printed out in hexadecimal (base 16) format
- The address of a variable varies from run to run, as the system allocates any free memory to the variable

Pointer Variable

- A variable that contains the address of another variable is called a pointer variable, or simply, a pointer.
- Example: a pointer variable a_ptr is shown as a blue box below. It contains the address of variable a.

Assuming that variable a is located at address ffbff7dc.

- Variable a_ptr is said to be pointing to variable a.
- If the address of a is immaterial, we simply draw an arrow from the blue box to the variable it points to.



Declaring a Pointer

Syntax:

```
type *pointer_name;
```

- pointer_name is the name (identifier) of the pointer
- type is the data type of the variable this pointer may point to

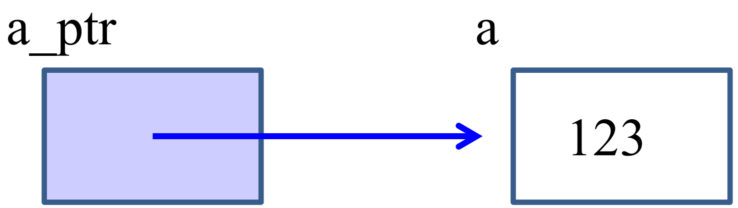
- Example: The following statement declares a pointer variable a_ptr which may point to any int variable
- Good practice to name a pointer with suffix _ptr or _p

```
int *a_ptr;
```

Assigning Value to a Pointer

- Since a pointer contains an address, only addresses may be assigned to a pointer
- Example: Assigning address of a to a_ptr

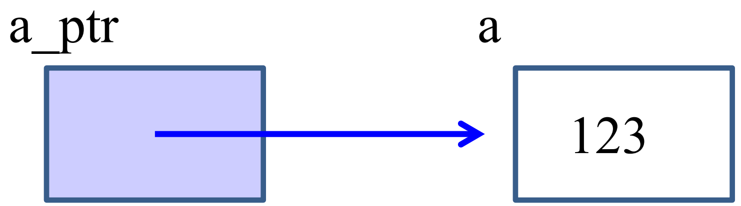
```
int a = 123;
int *a_ptr; // declaring an int pointer

a_ptr = &a;
```



- We may initialise a pointer during its declaration:

```
int a = 123;
int *a_ptr = &a; // initialising a_ptr
```

Accessing Variable Through Pointer



- Once we make a_ptr points to a (as shown above), we can now access a directly as usual, or indirectly through a_ptr by using the indirection operator (also called dereferencing operator): *

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

=

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

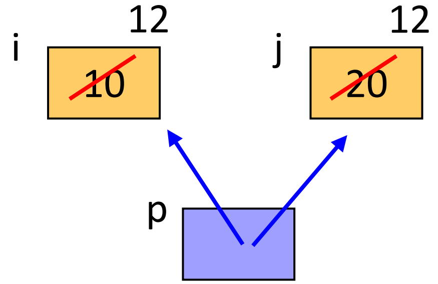
```
*a_ptr = 456;
```

=

```
a = 456;
```

Hence, *a_ptr is synonymous with a

Example #1




```c
int i = 10, j = 20;
int *p; // p is a pointer to some int variable

p = &i; // p now stores the address of variable i
```

Important! Now **`*p`** is equivalent to **`i`**

```c
printf("value of i is %d\n", *p);
```

```
value of i is 10
```

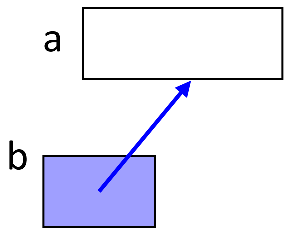
```c
// *p accesses the value of pointed/referred variable
*p = *p + 2; // increment *p (which is i) by 2
             // same effect as: i = i + 2;
p = &j; // p now stores the address of variable j
```

Important! Now **`*p`** is equivalent to

```c
*p = i;  // value of *p (which is j now) becomes 12
         // same effect as: j = i;
```

Example #2 (1/2)

```c
#include <stdio.h>

int main(void) {
   double a, *b;

   b = &a;
   *b = 12.34;
   printf("%f\n", a);

   return 0;
}
```



Can you draw the picture?
What is the output?

`12.340000`

What is the output if the **printf()** statement is changed to the following?

```c
printf("%f\n", *b);
```

`12.340000`

```c
printf("%f\n", b);
```

Compile with warning

```c
printf("%f\n", *a);
```

Error

What is the proper way to print a pointer?
(sometimes need to do this.)

```c
printf("%p\n", b);
```

`ffbff6a0`

Value in hexadecimal; varies from run to run.

Example #2 (2/2)

- How do we interpret the declaration?

```
double *a, b;
```

- The above is equivalent to

```
double *a; // this is straight-forward: a is a double variable
double b;
```

- Two declare two (or more) pointers..

```
double *a, *b;
```

- The following are equivalent:

```
double a;
double *b;
b = &a;
```

```
double a;
double *b = &a;
```

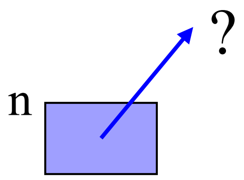
But this is not the same as above (and it is not legal):

```
double a;
double b = &a;
```

Common Mistake

```c
#include <stdio.h>

int main(void) {
   int *n;

   *n = 123;
   printf("%d\n", *n);

   return 0;
}
```

What's wrong with this?
Can you draw the picture?



- Where is the pointer n pointing to?
- Where is the value 123 assigned to?
- Result: Unexpected termination with an error message

Why Do We Use Pointers?

- It might appear that having a pointer to point to a variable is redundant since we can access the variable directly
- The purpose of pointers is apparent when we pass the address of a variable into a function, in the following scenarios:
  - To pass the address of the first element of an array to a function so that the function can access all elements in the array
  - To pass the addresses of two or more variables to a function so that the function can pass back to its caller new values for the variables
  - To access the memory hardware directly

Summary

- We briefly review the C programming language
  - Control structure
  - Data types,
  - Pointers
  - Etc

- The following are also important, but not covered due to the time constraint
  - User-defined types such as struct and typedef

- If you are not familiar with today's topic, please reconsider taking this course seriously

Next Time

- Labs in this week:
  - Lab1: C++ development environment setup & PA 2-1
  - Lab2: PA 2-2

- Next lecture:
  - 3. Introduction to C++