CS1010E Lecture 2
Simple C Programs

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
`www.comp.nus.edu.sg/~joxan`
`cs1010e@comp.nus.edu.sg`

Semester II, 2016/2017

Lecture Outline

- **Program Structure.**
- Constants and Variables.
- Assignment Statements.

Our First C Program

```c
/*--------------------------------------------------------------------*/
/*  Program chapter1_1                                                */
/*                                                                    */
/*  This program computes the                                         */
/*  distance between two points.                                      */

#include <stdio.h>
#include <math.h>

int main(void)
{
   /*  Declare and initialize variables.  */
   double x1=1, y1=5, x2=4, y2=7,
          side_1, side_2, distance;

   /*  Compute sides of a right triangle.  */
   side_1 = x2 - x1;
   side_2 = y2 - y1;
   distance = sqrt(side_1*side_1 + side_2*side_2);

   /*  Print distance.  */
   printf("The distance between the two points is "
          "%5.2f \n",distance);

   /*  Exit program.  */
   return 0;
}
/*--------------------------------------------------------------------*/
```

Comments

- The first five lines of this program contain **comments**
- These give the program a name (`chapter1_1`) and then document its purpose

```
/*--------------------------------------------------------*/
/*  Program chapter1_1                                    */
/*                                                        */
/*  This program computes the                             */
/*  distance between two points.                          */
```

- Comments begin with the characters `/*` and end with the characters `*/`

Comments

- A comment can be on a line by itself
- A comment can also extend over several lines
- Good style requires that comments be used throughout the program
- Improves readability and to document the computations
- Should also use initial comments to give a name and describe the purpose of the program

Preprocessor Directives

- **Preprocessor directives** provide instructions that are performed before the program is compiled

- Preprocessor directives being with a `#`.

- The `#include` directive inserts additional statements in the program

```
#include <stdio.h>
#include <math.h>
```

- Statements in the files `stdio.h` and `math.h` should be included in place of these two statements before the program is compiled

Preprocessor Directives

- The $<$ and $>$ characters indicate that the files are included with the **Standard C library** that comes with an ANSI C compiler
- The `stdio.h` file contains information related to the output statement used in this program
- The `math.h` file contains information related to the function used to compute the square root
- The `.h` extension specifies that they are header files
- Usually placed after initial comments

Main Function

- Every C program contains a set of statements called a `main` function
- The keyword `int` indicates that the function returns an integer value to the operating system
- The keyword `void` indicates that the function is not receiving any information from the operating system
- The body of the function is enclosed by braces, `{ }`

Main Function

- These braces are placed on lines by themselves to easily identify the body of the function

- The following two lines specify the beginning of the main function.

```
int main(void)
{
```

Declarations

- The main function contains two types of commands: declarations and statements

- The **declarations** define the memory locations that will be used by the statements

- Declarations must therefore precede the statements

- The declarations may or may not give **initial values** to be stored in the memory locations

```
/*  Declare and initialize variables.  */
double x1=1, y1=5, x2=4, y2=7,
  side_1, side_2, distance;
```

Declarations

- The program will use seven variables named `x1`, `y1`, `x2`, `y2`, `side_1`, `side_2`, and `distance`
- The term `double` indicates that all the variables will store double-precision floating-point values
- These variables can store values such as 12.5 and -0.0005 with many digits of precision.
- Also, `x1` should be initialized (given an initial value) to the value 1
- `y1` should be initialized to the value 5

Declarations

- `x2` should be initialized to the value 4
- `y2` should be initialized to the value 7
- Initial values of `side_1`, `side_2`, and `distance` are not specified and should not be assumed to be initialized to zero
- The declaration was too long for one line, so we split it over two lines
- Indenting of the second line indicates that it is a continuation of the previous line. Contributes to good style guidelines

Statements

- The **statements** that specify the operations to be performed are:

```
/*  Compute sides of a right triangle.  */
side_1 = x2 - x1;
side_2 = y2 - y1;
distance = sqrt(side_1*side_1 + side_2*side_2);

/*  Print distance.  */
printf("The distance between the two points is "
  "%5.2f \n",distance);
```

- Compute the lengths of the two sides of the right triangle

- Compute the length of the hypotenus

Statements

- Distance is printed with the `printf` statement.
- The output statement is too long for a single line, so we separate the statement into two lines.
- Indenting of the second line indicates that it is a continuation of the previous line.
- Declarations and statements are all required to end with a semicolon.
- Details discussed later in the course.

End Execution

- To end execution of the program and return control to the operating system, we use a `return 0;` statement

```
/*  Exit program.  */
return 0;
```

- This statement returns a value of 0 to the operating system

- A value of zero indicates a successful end of execution

End of Main

- The body of the `main` function then ends with the right brace on a line by itself
- A comment line delineates the end of the main function.

```
}
/*----------------------------------------------------------*/
```

White Space

- Note that we have also included blank lines (called white space) in the program to separate different components
- White space makes a program more readable and easy to modify
- The declarations and statements within the main function were indented to show the structure of the program
- This spacing provides a consistent style, and makes our programs easier to read

General Form

- Here is the **general form** of a C program.

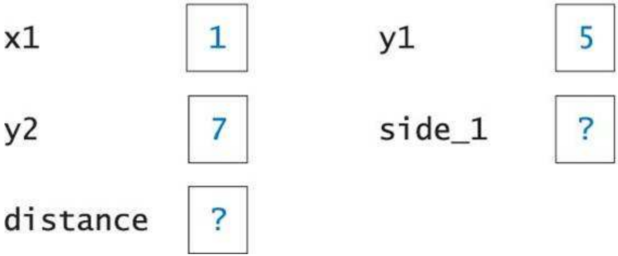
```
preprocessing directives
int main(void)
{
   declarations;
   statements;
}
```

Constants and Variables

- Constants and variables represent values that we use in our program.
- **Constants** are specific values, such as 2, 3.1416, -1.5, ‘a’, or “hello”, that we include in the C statements
- **Variables** are memory locations that are assigned a name or **identifier**
- The identifier is used to reference the value stored in the memory location
- Analogy: mailbox has a person’s name (identifier) on it; the mailbox contains a value

Memory Snapshot

```
double x1=1, y1=5, x2=4, y2=7,
       side_1, side_2, distance;
```



Constants and Variables

- Values of variables that were not given initial values are unspecified
- Indicated with a question mark; sometimes these values are called **garbage values**
- The diagram on the previous slide is called a **memory snapshot** because it shows the contents of a memory location at a specified point in the execution of the program
- The preceding memory snapshot shows the variables and contents as specified by the declaration statement

Identifiers

- Rules for selecting a valid identifier are:
  - An identifier must begin with an alphabetic character or the underscore character (_);
  - Alphabetic characters in an identifier can be lowercase or uppercase letters;
  - An identifier can contain digits, but not as the first character; and
  - An identifier can be of any length.

Case Sensitive

- C is **case sensitive**, thus uppercase letters are different from lowercase letters
- `Total`, `TOTAL`, and `total` represent three different variables
- C also includes **keywords** with special meaning to the C compiler that cannot be used for identifiers

Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Valid Identifiers

- Examples of valid identifiers.
  - `distance`
  - `x_1`
  - `X_sum`
  - `average_measurement`
  - `initial_time`

Invalid Identifiers

- Examples of invalid identifiers.
  - `1x` (begins with a digit)
  - `switch` (is a keyword)
  - `$sum` (contains an invalid character, `$`)
  - `rate%` (contains an invalid character, `%`)

Identifier Names

- Identifier names should be carefully selected.
- It must reflect the contents of the variable
- The name should also indicate the units of measurement
- If a variable represents a temperature measurement in degrees Fahrenheit, use an identifier such as `temp_F` or `degrees_F`
- If a variable represents an angle, name it `theta_rad` to indicate that the angle is measured in radians or `theta_deg` if the angle is measured in degrees

Declarations

- The declarations at the beginning of the `main` function must include all the identifiers of the variables that we plan to use in the main function.

- The declarations must also specify the types of values that will be stored in the variables.

- These apply also to other C functions

Scientific Notation

- A **floating-point** value is one that can represent both integer and noninteger values such as 2.5, -0.004, and 15.0
- A floating-point value expressed in **scientific notation** is rewritten as a mantissa times a power of 10, where the mantissa has an absolute value greater than or equal to 1.0 and strictly less than 10.0
- Example: in scientific notation, 25.6 is written as 2.56×10^1, -0.004 is written as -4.0×10^{-3}, and 1.5 is written as 1.5×10^0

Exponential Notation

- In **exponential notation**, the letter **e** is used to separate the mantissa from the exponent of the power of 10

- Example: in exponential notation, 25.6 is written as `2.56e1`, -0.004 is written as `-4.0e-3`, and 1.5 is written as `1.5e0`

Precision and Range

- The number of digits allowed by the computer for the decimal portion of the mantissa determines the precision
- The number of digits allowed for the exponent determines the range
- Values with two digits of precision and an exponent range of -8 to 7 could include values such as 2.33×10^5 $(233,000)$ and 5.92×10^{-8} (0.0000000592)
- This precision and exponent range would not be sufficient for many of the types of values that we use in engineering problem solutions

Precision and Range

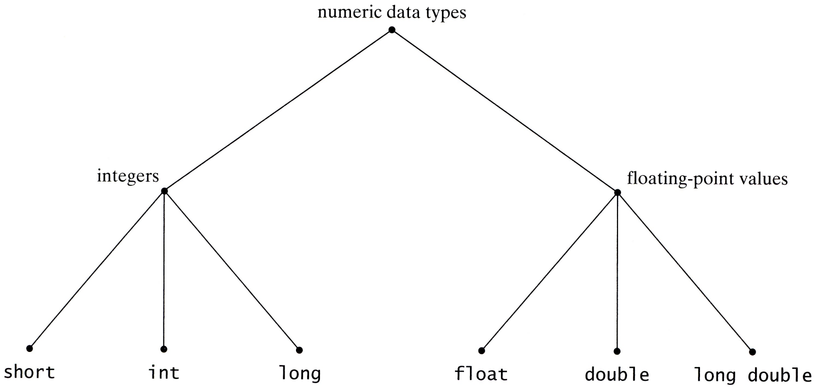
- For example, the distance in miles from Mars to the Sun is $141,517,510$ or 1.4151751×10^8

- To represent this value, we would need at least seven digits of precision and an exponent range that included the integer 8

Numeric Data Types

- For example, the distance in miles from Mars to the Sun is $141,517,510$ or 1.4151751×10^8
- To represent this value, we would need at least seven digits of precision and an exponent range that included the integer 8

Numeric Data Types



Signed Integers

- The **type specifiers** for signed integers are `short`, `int`, and `long`, for short integer, integer, and long integer, respectively
- The specific range of values are **system dependent**; ranges can vary from one system to another
- The short integer data type ranges from $-32,768$ to $32,767$
- The integer and long integer data types range from $-2,147,483,648$ to $2,147,483,647$

Unsigned Integers

- C also allows an `unsigned` qualifier to be added to integer specifiers. An unsigned integer represents only non-negative values
- Signed and unsigned integers can represent the same number of values, but the ranges are different
- An `unsigned short` has the range of values from 0 to $65,535$; a `short` integer has the range of values from $-32,768$ to $32,767$
- Both variables can represent a total of $65,536$ values

Floating-Point Values

- The type specifiers for floating-point values are `float` (single precision), `double` (double precision), and `long double` (extended precision)
- The following statement from program `chapter1_1` defines seven variables that all contain double-precision floating-point values

```
double x1=1, y1=5, x2=4, y2=7,
  side_1, side_2, distance;
```

- The difference between the three types relate to the precision (accuracy) and the range of the values represented

Floating-Point Values

- The precision and range are system dependent
- On most systems, a `double` data type stores about twice as many decimal digits of precision as are stored with a `float` data type
- A `double` value will have a wider range of exponent values than a `float` value
- The `long double` value may have more precision and a still wider exponent range, but this is system dependent

Example Data-Type Limit

Integers	
`short`	Maximum = 32,767
`int`	Maximum = 2,147,483,647
`long`	Maximum = 2,147,483,647
Floating Point	
`float`	6 digits of precision Maximum exponent 38 Maximum value 3.402823e+38
`double`	15 digits of precision Maximum exponent 308 Maximum value 1.797693e+308
`long double`	15 digits of precision Maximum exponent 308 Maximum value 1.797693e+308

*Microsoft Visual C++ 6.0 compiler.

Floating-Point Values

- A floating-point constant such as `2.3` is assumed to be a `double` constant
- To specify a `float` constant or a `long double` constant, the letter (or suffix) `F` or `L` must be appended to the constant
- Thus, `2.3F` and `2.3L` represent a `float` constant and a `long double` constant, respectively

Character Data

- All information stored in a computer is represented internally as sequences of binary digits (0 and 1)
- Each **character** corresponds to a **binary code** value
- The most commonly used binary codes are **ASCII** (American Standard Code for Information Interchange) and **EBCDIC** (Extended Binary Coded Decimal Interchange Code)
- We assume that the ASCII code is used to represent characters

ASCII Codes

Character	ASCII Code	Integer Equivalent
newline, \n	0001010	10
%	0100101	37
3	0110011	51
A	1000001	65
a	1100001	97
b	1100010	98
c	1100011	99

ASCII Codes

- The character ‘a’ is represented by the binary value 1100001, which is equivalent to the integer value of 97.
- A total of 128 characters can be represented in the ASCII code
- Complete ASCII code tables can be found in Appendix B of Etter’s book and most programming books. Google is your friend!

Character Data

- Character data can be represented by constants or by variables
- A character constant is enclosed in single quotes, such as `'A'`, `'b'`, and `'3'`
- A variable that is going to contain a character is defined as an integer or a character data type
- The type specifier for characters is `char`

Characters and Integers

- Once a character is stored in memory as a binary value, the binary value can be interpreted as a character or as an integer
- The binary ASCII representation for a *character* digit is not equal to the binary representation for an *integer* digit
- From the ASCII table, the ASCII binary representation for the character digit 3, or `'3'`, is 0110011, which is equivalent to the binary representation of the integer value 51
- Character `'3'` has integer value 51, while the integer 3 has integer value 3 (of course)

Characters and Integers

```
/*------------------------------------------------------------
/*  Program chapter2_1
/*
/*  This program prints two values
/*  as characters and integers.

#include <stdio.h>

int main(void)
{
   /*  Declare and initialize variables.  */
   char ch='a';
   int i=97;

   /*  Print both values as characters.  */
   printf("value of ch: %c; value of i: %c \n",ch,i);

   /*  Print both values as integers.  */
   printf("value of ch: %i; value of i: %i \n",ch,i);

   /*  Exit program.  */
   return 0;
}
/*------------------------------------------------------------
```

Characters and Integers

- The output from the program is:
  - ```
    value of ch:  a; value of i:  a
    value of ch:  97; value of i:  97
    ```
    ```

Symbolic Constants

- A **symbolic constant** is defined with a preprocessor directive that assigns an identifier to the constant

- The directive can appear anywhere in a C program

- The compiler will replace each occurrence of the directive identifier with the constant value in all statements that follow the directive

- Engineering constants such as π or g (the acceleration due to gravity) are good candidates for symbolic constants

Symbolic Constants

- The following preprocessor directive assigns the value 3.141593 to the variable `PI`:
  - `#define PI 3.141593`

- Statements that need to use the value of π would then use the symbolic constant identifier `PI` instead of 3.141593.

- The following statement computes the area of a circle:
  - `area = PI*radius*radius;`

Symbolic Constants

- Symbolic constants are usually defined with uppercase identifier (as in `PI` instead of `pi`) to indicate that they are symbolic constants
- Identifiers should be well selected and easy to remember
- Separate `#define` directives are needed for each symbolic constant required
- Preprocessor directives do not end with a semicolon

Assignment Statements

- An **assignment statement** is used to assign a value to an identifier
- The general form of the assignment statement is:
  - `identifier = expression;`
- An **expression** can be a constant, another variable, or the result of an operation
- Example 1:
  - ```
    double sum=10.5;
    int x1=3;
    char ch='a';
    ```
    ```

Assignment Statements

- Example 2:
  - ```
    double sum;
    int x1;
    char ch;
    ...
    sum = 10.5;
    x1 = 3;
    ch = 'a';
    ```

- After each set of statements is executed, the value of `sum` is 10.5, the value of `x1` is 3, and the value of `ch` is ‘a’
    ```

Memory Snapshot

sum	10.5	x1	3	ch	'a'

Assignment Statements

- In Example 1, the statements define and initialize the variables at the same time

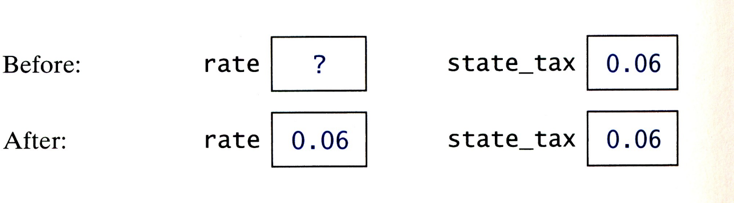
- In Example 2, the assignment statements could be used at any point in the program. The assignment statements may be used to change (not just initialize) the values in variables

- Multiple assignments are allowed. The following statement assigns a value of zero to each of the variables `x`, `y`, and `z`:
  - `x = y = z = 0;`

Assignment Statements

- We can assign a value from one variable to another with:
    - `rate = state_tax;`
- The equal sign should be read as "is assigned the value of".
- The statement says "`rate` is assigned the value of `state_tax`"
- If `state_tax` contains the value 0.06, then `rate` also contains the value 0.06 after the statement is executed
- The value in `state_tax` is not changed

Memory Snapshot



Conversions

- If we assign a value to a variable that has a different data type, then a conversion must occur during the execution of the statement

- Sometimes the conversion can result in information being lost.

- Consider the following declaration and assignment statement:
  - ```
    int a;
    ...
    a = 12.8;
    ```
    ```

Conversions

- Because `a` is defined as an integer, it cannot store a value with a nonzero decimal portion
- Therefore, `a` will contain the value 12 and not 12.8
- Note that the value is truncated and not rounded up

Numeric Conversions

- To determine whether a numeric conversion will work properly or not, we use the following order (from high to low):

high:	long double
	double
	float
	long integer
	integer
low:	short integer

Numeric Conversions

- If a value is moved to a data type that is higher in order, no information will be lost
- If a value is moved to a data type that is lower in order, information may be lost
- Moving an integer to a double will work properly
- Moving a float to an integer may result in the loss of some information or an incorrect result
- Use only assignments that do not cause potential conversion problems

Arithmetic Operators

- An assignment statement can be used to assign the result of an arithmetic operation to a variable
- The following statement computes the area of a square:
  - `area_square = side*side;`
- $\star$ is used to indicate multiplication
- The symbols $+$, $-$, and $/$ are used to indicate addition, subtraction, and division respectively

Arithmetic Operators

- Each of the following statements is a valid computation for the area of a triangle:
  - `area_triangle = 0.5*base*height;`
  - `area_triangle = (base*height)/2;`
- The use of parentheses in the second statement is not required, but is used for readability

Increment

- Consider this assignment statement:
  - `x = x + 1;`
- In algebra, this statement is invalid, because a value cannot be equal to itself plus 1
- This assignment statement should not be read as an equality.
- It should be read as "`x` is assigned the value of `x` plus 1"

Increment

- Therefore, the statement indicates that the value stored in the variable `x` is incremented by 1

- If the value of `x` is 5 before this statement is executed, then the value of `x` will be 6 after the statement is executed

Modulus

- C also includes a **modulus** operator (`%`) that is used to compute the remainder in a division between two integers
- For example, `5%2` is equal to `1`
- `6%3` is equal to `0`
- `2%7` is equal to `2`
    - The quotient of `2/7` is zero with a remainder of `2`

Division

- If `a` and `b` are integers, then the expression `a/b` computes the integer quotient, whereas the expression `a%b` computes the integer remainder
- Thus, if `a` is equal to `9` and `b` is equal to `4`, the value of `a/b` is 2, and the value of `a%b` is 1
- An execution error occurs if the value of `b` is equal to zero in either `a/b` or `a%b` because the computer cannot perform division by zero
- If either of the integer values in `a` and `b` is negative, the result of `a%b` is system dependent

Modulus

- The modulus operator is useful in determining if an integer is a multiple of another number
- If `a%2` is equal to zero, then `a` is even; otherwise, `a` is odd
- If `a%5` is equal to zero, then `a` is a multiple of 5
- We will use the modulus operator frequently in the development of engineering solutions

Unary Operators

- The five operators (`+`, `-`, `*`, `/`, `%`) are **binary operators** — operators that operate on two values
- C also includes **unary operators** — operators that operate on a single value
- Plus and minus signs can be unary operators when they are used in an expression such as `-x`

Binary Operators

- The result of a binary operation with values of the same type is another value of the same type
- If `a` and `b` are `double` values, then the result of `a/b` is also a `double` value
- If `a` and `b` are integers, then the result of `a/b` is also an integer
- Integer division can sometimes produce unexpected results, because any decimal portion of the integer division is dropped

Binary Operators

- The result of a binary operation with values of the same type is another value of the same type
- If `a` and `b` are `double` values, then the result of `a/b` is also a `double` value
- If `a` and `b` are integers, then the result of `a/b` is also an integer
- Integer division can sometimes produce unexpected results, because any decimal portion of the integer division is dropped.

Mixed Operation

- An operation between values of different types is a **mixed operation**
- Before the operation is performed, the value with the lower type is converted or promoted to the higher type
- Thus the operation is performed with values of the same type.
- If an operation is specified between an `int` and a `float`, the `int` will be converted to a `float` before the operation is performed
- The result will be a `float`

Cast Operator

- Suppose we want to compute the average of a set of integers.

- If the sum and the count of the integers have been stored into integer variables `sum` and `count`, the following appears to be correct:
  - ```
    int sum, count;
    float average;
    ...
    average = sum/count;
    ```

- However, the division between two integers gives an integer result that is then converted to a `float` value
    ```

Cast Operator

- If `sum` is 18 and `count` is 5, then the value of `average` is 3.0, not 3.6
- We must use a **cast operator** — a unary operator that allows us to specify a type change in the value before the next computation
- The cast `(float)` is applied to `sum`:
  - `average = (float)sum/count;`
- The value of `sum` is converted to a `float` value before the division is performed

Cast Operator

- The division becomes a mixed operation between a `float` value and an integer

- So the value of `count` is converted to a `float` value; the result of the division is then a `float` value that is stored in `average`

- If `sum` is 18 and `count` is 5, then the value of `average` is now correctly computed to be 3.6

- Note that the cast operator affects only the value used in the computation; it does not change the value stored in the variable `sum`

Priority of Operators

- In an expression that contains more than one operator, we must know the order in which the operations are performed
- We must know the **precedence** of operators, which matches the standard algebraic precedence
- Operations within parentheses are always evaluated first.
- If the parentheses are nested, the operations within the innermost parentheses are evaluated first

Arithmetic Operator Precedence

Precedence	Operator	Associativity
1	Parentheses: ( )	Innermost first
2	Unary operators: + − (type)	Right to left
3	Binary operators: * / %	Left to right
4	Binary operators: + −	Left to right

Priority of Operators

- Unary operators are evaluated before the binary operations `*`, `/`, and `%`
- Binary addition and subtraction are evaluated last
- If there are several operators of the same precedence level in an expression, the variables or constants are grouped (or associated) with the operators in a specific order

Priority of Operators

- Consider the following expression:
  - `a*b + b/c*d`
- Because multiplication and division have the same precedence level, and because the **associativity** (the order for grouping the operations) is from left to right, this expression will be evaluated as if it contained the following:
  - `(a*b) + ((b/c)*d)`

Overflow and Underflow

- The values stored in a computer have a wide range of allowed values
- However, if the result of a computation exceeds the range of allowed values, an error occurs
- Assume that the exponent range of a floating-point value is from -38 to 38
- This range should accommodate most computations, but it is possible for the results of an expression to be outside of this range

Overflow

- Suppose we execute the following commands:
  - ```
    x = 2.5e30;
    y = 1.0e30;
    z = x*y;
    ```
- The values of `x` and `y` are within the allowable range
- The value of `z` should be `2.5e60`, but this value exceeds the range
- This error is called exponent **overflow**
    ```

Overflow

- The exponent of the result of an arithmetic operation is too large to store in the memory assigned to the variable

- The action generated by an exponent overflow is system dependent

Underflow

- Suppose we execute the following commands:
  - ```
    x = 2.5e-30;
    y = 1.0e30;
    z = x/y;
    ```

- The values of `x` and `y` are within the allowable range

- The value of `z` should be `2.5e-60`, but this value exceeds the range

- This error is called exponent **underflow**
    ```

Underflow

- The exponent of the result of an arithmetic operation is too small to store in the memory assigned to the variable
- The action generated by an exponent underflow is system dependent

Increment and Decrement

- The C language contains unary operators for incrementing and decrementing variables
- These operators cannot be used with constants or expressions.
- The operators are the increment operator `++` and the decrement operator `--`
- Both these operators can be applied either in a **prefix** position (before the identifier), or in a **postfix** position (after the identifier)
- Example of prefix position: `++count`
- Example of postfix position: `count++`

Increment and Decrement

- If an increment or decrement operator is used by itself, it is equivalent to an assignment statement that increments or decrements the value

- The statement:
  - `y--;`

- is equivalent to the statement:
  - `y = y - 1;`

Increment and Decrement

- If the increment or decrement operator is used in an expression, then the expression must be evaluated carefully
- If the increment or decrement operator is in a prefix position, the identifier is modified, and then the new value is used in evaluating the rest of the expression
- If the increment or decrement operator is in a postfix position, the old value of the identifier is used to evaluate the rest of the expression, and then the identifier is modified

Increment and Decrement

- Execution of the statement:
  - `w = ++x - y;`
- is equivalent to the execution of this pair of statements:
  - ```
    x = x + 1;
    w = x - y;
    ```
- If the value of `x` is 5 and the value of `y` is 3, then the value of `x` increases to 6 and the value of `w` becomes 3
    ```

Increment and Decrement

- Similarly, execution of the statement:
  - `w = x++ - y;`
- is equivalent to the execution of this pair of statements:
  - ```
    w = x - y;
    x = x + 1;
    ```
- If the value of `x` is 5 and the value of `y` is 3, then the value of `w` becomes 2 and `x` increases to 6
    ```

Increment and Decrement: IMPORTANT NOTE

- Some expressions using $++$ and $--$ are ambiguous
- `x = 0; y = ++x + ++x`
  Is the final value of `y` equal to 2 or 3?
- The key question is when the first increment is visible
- $\boxed{\text{No general solution for this}}$
- A sufficient solution: use $++$ and $-$ only once for each variable in a statement

More Assignment Operators

- C allows simple assignment statements to be abbreviated

- Each of the following pair of statements contain equivalent statements:
  - `x = x + 3;`
    `x += 3;`
  - `sum = sum + x;`
    `sum += x;`
  - `d = d/4.5;`
    `d /= 4.5;`
  - `r = r%2;`
    `r %= 2;`

More Assignment Operators

- In general, any statement of the form:
  - `identifier = identifier operator expression;`
- can be written in this form:
  - `identifier operator= expression;`
- **Abbreviated assignment** statements are usually used because they are shorter

More Assignment Operators

- Earlier we used the following **multiple-assignment** statement:
  - `x = y = z = 0;`
- The interpretation of the above statement is clear
- But the interpretation of the following statement is not as clear:
  - `a = b += c + d;`
- To evaluate this properly, we use the table on the next slide

More Operator Precedence

Precedence	Operator	Associativit
1	Parentheses: ( )	Innermost fir
2	Unary operators: + − ++ −− (type)	Right to left
3	Binary operators: * / %	Left to right
4	Binary operators: + −	Left to right
5	Assignment operators: = += −= *= /= %=	Right to left

More Assignment Operators

- The assignment operators are evaluated last and their associativity is right to left.

- Thus, the statement is equivalent to:
  - `a = (b += (c + d));`

- If we replace the abbreviated forms with the longer forms of the operations, we have:
  - `a = (b = b + (c + d));`

- or:
  - ```
    b = b + (c + d);
    a = b;
    ```
    ```

More Assignment Operators

- While this gives us good practice with the precedence / associativity table, statements used in a program should be more readable
- Using abbreviated assignment statements in a multiple-assignment statement is not recommended
- Note that the spacing convention we use inserts spaces around abbreviated operators and multiple-assignment operators because these operators are evaluated after the arithmetic operators

References

Etter Sections 2.1 to 2.3

Next Lecture

Simple C Programs – Part 2