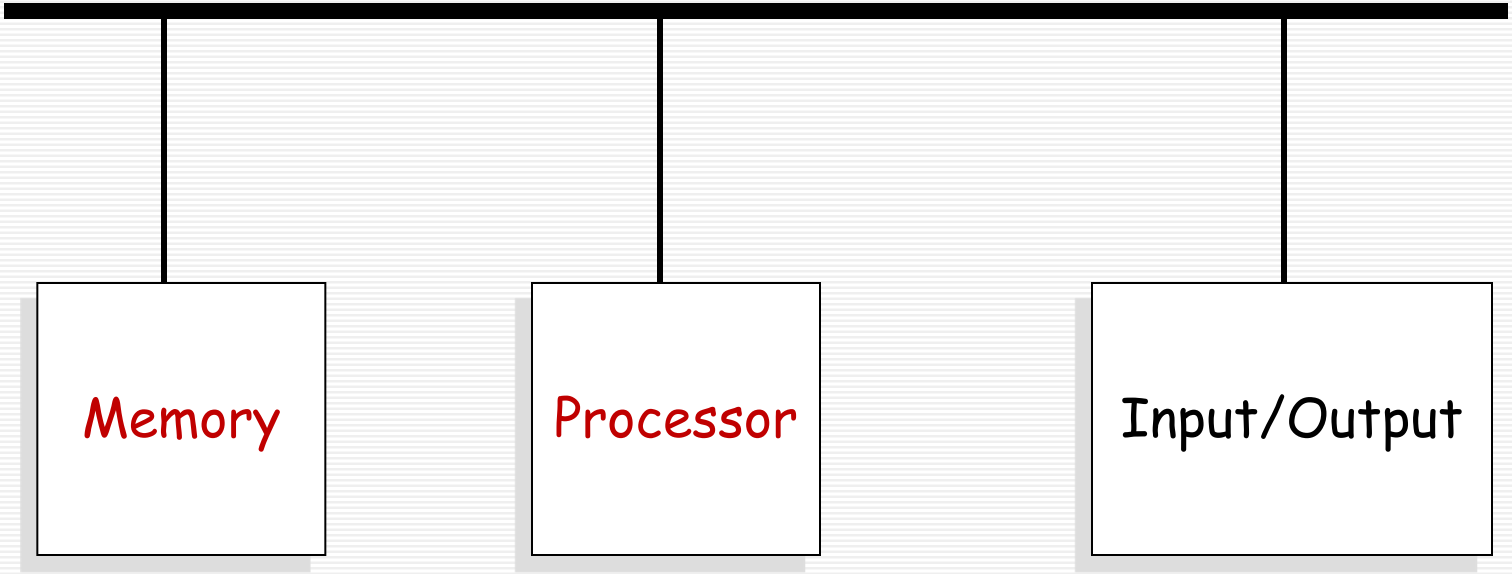
An Overview of C

Algorithmic Thinking

- Very diligent
- But, not so smart
- Can do a few of simple operations (instructions)
- Complex operation: a series of simple operations

- Must be told in detail what to do
  - understandable to computer
  - for all possible cases
- Algorithmic Thinking
  - Algorithms == Recipes

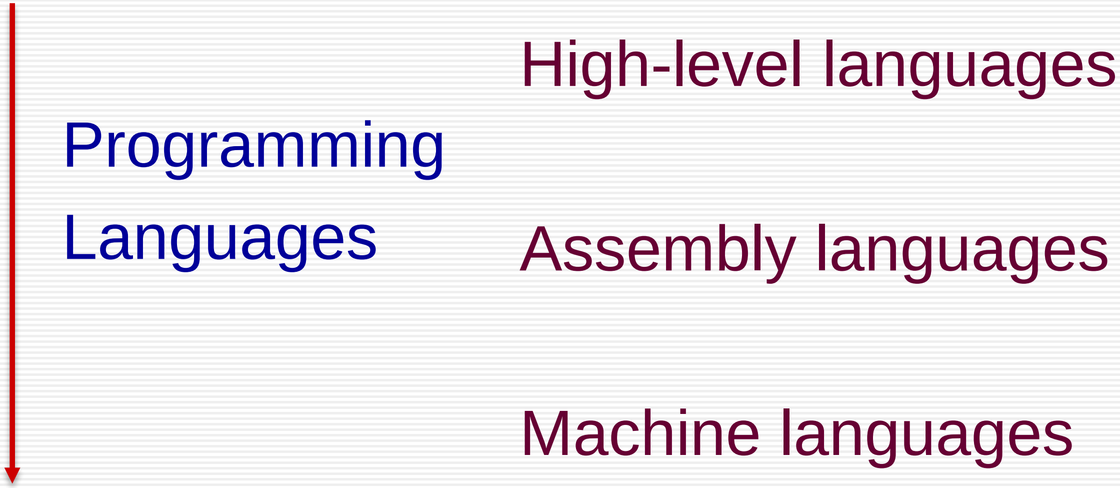
Von Neumann Architecture



- Stored Program Concept

Programming Languages

- Algorithms: Developed by people



Programming Languages

High-level languages

Assembly languages

Machine languages

- Computers: Execute algorithms

How to Learn Programming

- Learn by doing
  - Do exercises/practices.
  - Lectures will give you basic tools only.

- In the lectures, you will learn:
  - Language syntax
  - Algorithmic thinking

- Read “An Overview of C” & Try by yourself
  - A Book on C

Warning!!

- Lectures
  - seem easy
- Textbook: An Overview of C
  - seems that you understand well
- Programming assignments
  - more difficult than it seems

- Expect many bugs in your programs

Programming maturity comes with p.r.a.c.t.i.c.e!!

C Programming Language (1/2)

- Born in the early 1970s with UNIX
- C is
  - Small
    - Fewer keywords
  - Portable
    - Code written on one machine easily moved to another
  - Terse
    - A very powerful set of operators
    - Able to access the machine in the bit level
  - Widely used
  - The basis for C++ and Java

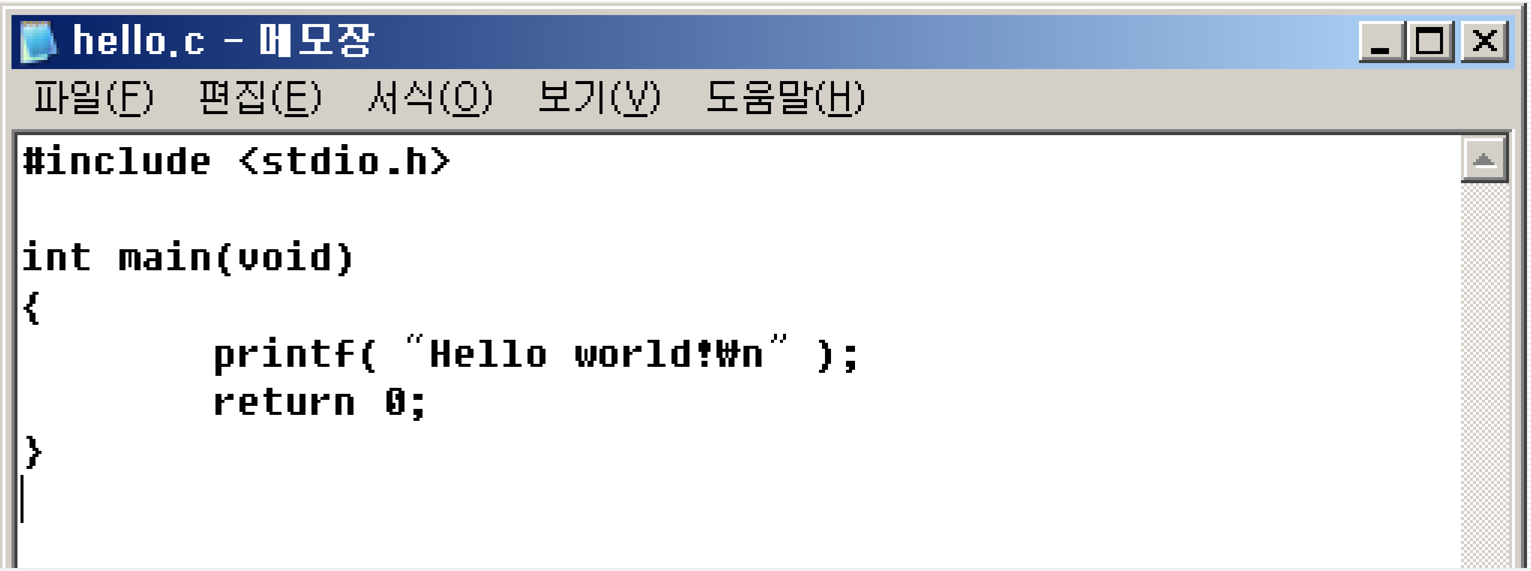
C Programming Language

- Criticism
  - Complicated syntax
  - No automatic array bounds checking
  - Multiple use of such symbols as * and =
    - **, ==

- Nevertheless, C is an elegant language

Example: Hello world (1/3)

1. Create a C source file
   - use a text editor
     - Vi, Microsoft Visual C++ Editor, …



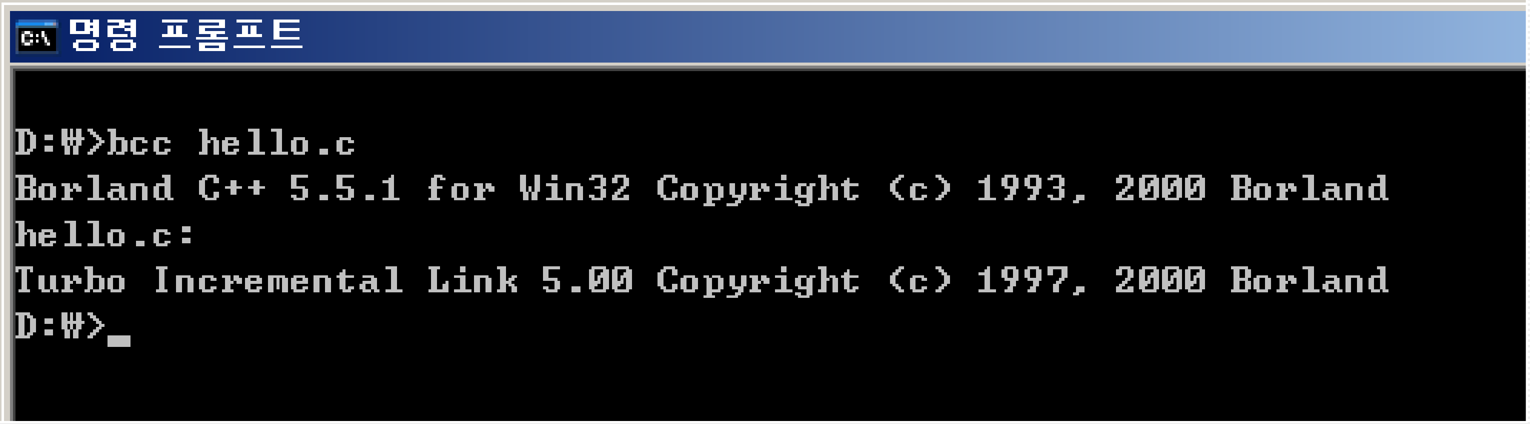
```
#include <stdio.h>

int main(void)
{
        printf( “Hello world!₩n” );
        return 0;
}
```

Hello world (2/3)

2. Compile

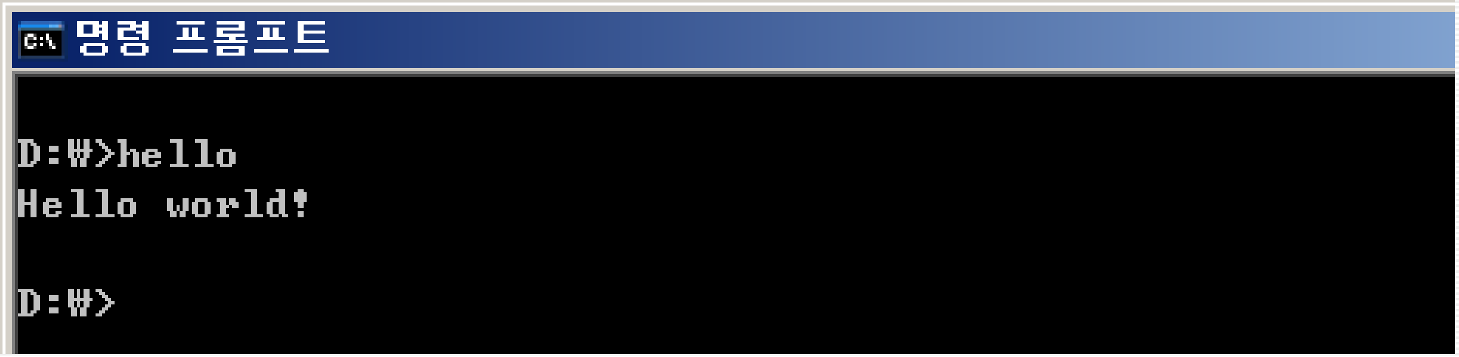
- Convert source codes to object codes
- Compiler does the job



```
D:₩>bcc hello.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
hello.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
D:₩>_
```

Hello world (3/3)

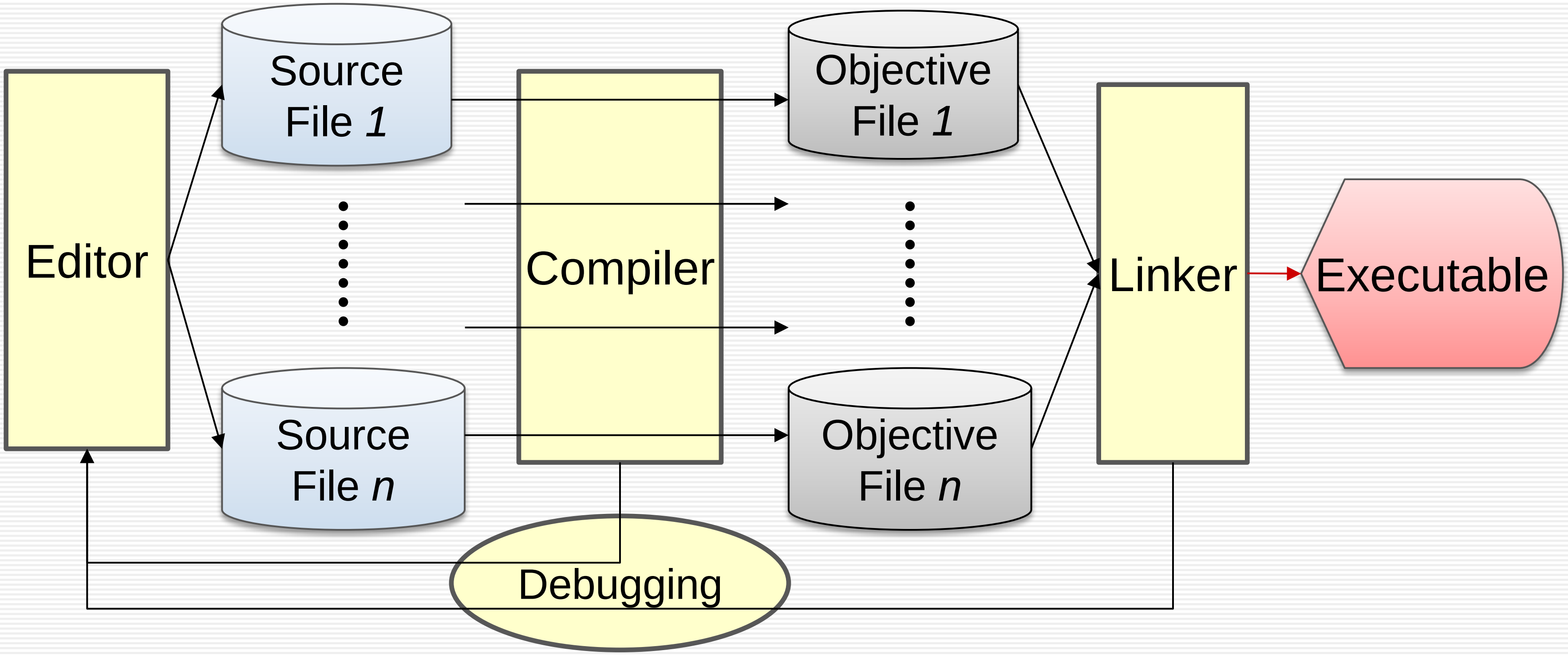
3. Linking
   - Convert object codes to executable file
   - Linker does the job
4. Debugging
   - Fix the bugs in the source codes
   - Debugger does the job
5. Run or Excute



```
D:₩>hello
Hello world!

D:₩>
```

From Source to Executable



Program Output (1/6)

Source file: sea.c

```
#include <stdio.h>


int main(void)
{
    printf(“from sea to shining
        C\n”);
    return 0;
}
```

```
from sea to shining C
```

Program Output (2/6)

#include <stdio.h>

- Preprocessor
  - built into the C compiler
  - Lines beginning with #: communicate with the preprocessor

- #include
  - Preprocessor includes a copy of the header file ***stdio.h***
  - **stdio.h**
    - provided by the C system
    - Declaration of standard input/output functions, e.g. **printf()**

Program Output (3/6)

```
int main(void)
{
          ⋮
}
```

- The 1st line of the function definition for main()
- int, void
  - Keywords
  - Special meaning to the compiler

- Every program has a function named **main()**
- **void,** no argument / return an **int** value
- **{** … **}**, the body of a function definition

Program Output (4/6)

printf()

- A function that prints on the screen
- information in the header file ***stdio.h***

“from sea to shinning C\n”

- “… “ : string constant in C
- **\n** : a single character called *newline*

printf(“from sea to shinning C\n”);

- statement : end with a semicolon

Program Output (5/6)

return 0;

- A **return** statement
- causes the value *zero* to be returned to the operating system

}

- The right brace matches the left brace
- ending the function definition for **main()**

Program Output (6/6)

```
#include <stdio.h>

int main(void)
{
      printf(“from sea to “);
      printf(“shining C“);
      printf(“\n“);
      return 0;
}
```

```
from sea to shining C
```

```
#include <stdio.h>

int main(void)
{
     printf(“from sea\n“);
     printf(“to shining\nC\n“);
     return 0;
}
```

```
from sea
to shining
C
```

Compiling

- Convert source file to objective file
  - sea.c to sea.o (or sea.obj)
- Object file
  - a file with expressions that computers can understand
- When compiling fails?
  - something wrong with source file ...
    - expressions with wrong C grammar

Errors in Source File (example)

```
#include <stdio.h>

int main(void)
{
        printf("from sea to shining
                C\n");
        returm 0;
}
```

- returm 0;

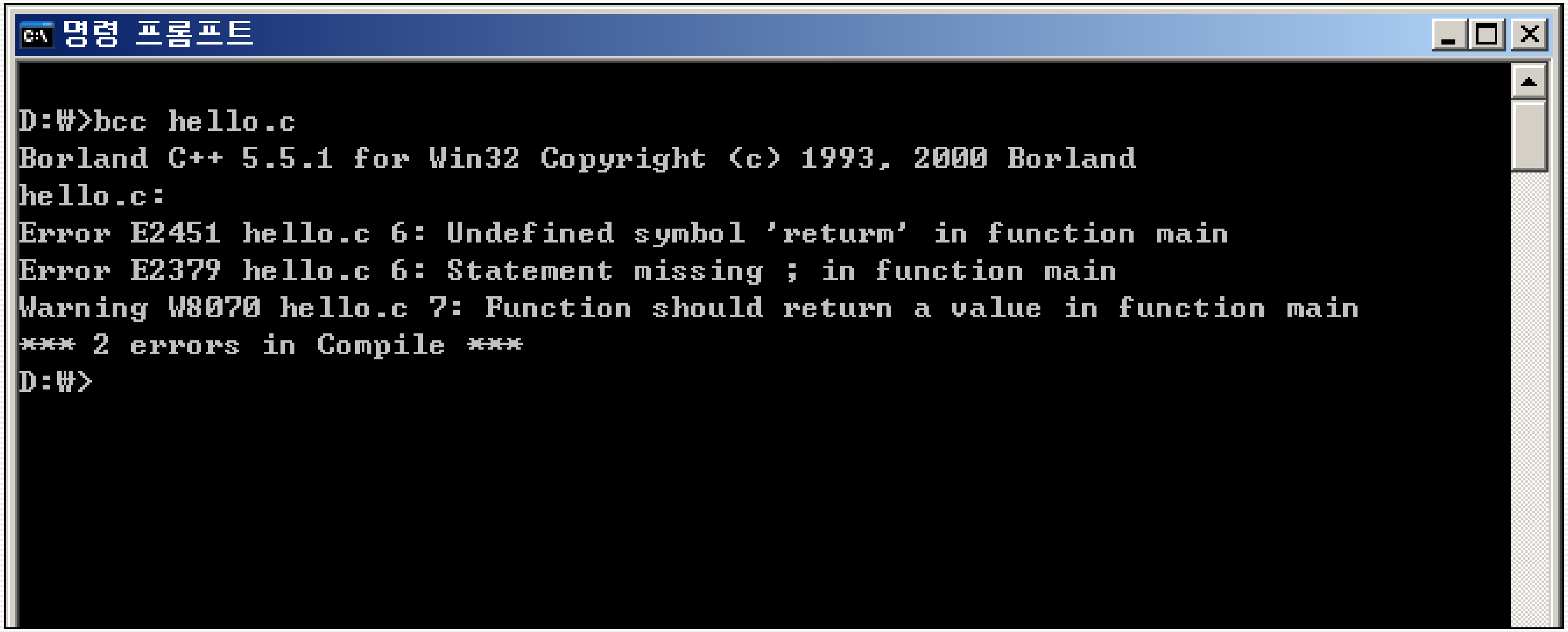
 incorrect C language grammar

- compiler fails to make an obj file and returns an error.

- debugging:

change "returm 0;" to "return 0;"

Errors in Source File (example)



```
D:₩>bcc hello.c
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
hello.c:
Error E2451 hello.c 6: Undefined symbol 'returm' in function main
Error E2379 hello.c 6: Statement missing ; in function main
Warning W8070 hello.c 7: Function should return a value in function main
*** 2 errors in Compile ***
D:₩>
```

Linking and Running a Program

- Linking
  - The process to make an executable program out of objective file(s)
    - sea.o (or sea.obj) → a.out (sea.exe)

- Run a program
  - type “a.out” or “sea”
    - computer prints “from see to shining C”

Simple Examples

1. Assignments
2. Control Flow

[Ex.1] Distance of a marathon in kilometers

- Marathon: 26 miles 385 yards
- 1 yard → 1/1760 mile
- 1 mile→ 1.609 km

$\Downarrow$ Marathon

- (26 + 385/1760) miles
- (26 + 385/1760) x 1.609 km

Variable, Expressions, Assignment (1/7)

```
/* the distance of a marathon in kilometers */
#include <stdio.h>
int main(void)
{
      int    miles, yards;
      float  kilometers;

      miles = 26;
      yards = 385;
      kilometers = 1.609 * (miles + yards / 1760.0);
      printf(“\nA marathon is %f kilometers.\n\n”,
            kilometers);
      return 0;
}
```

Variable, Expressions, Assignment (2/7)

/* the distance of a marathon in kilometers */

- **/*** … ***/**
  - comment
  - ignored by the compiler

Variable, Expressions, Assignment (3/7)

int miles, yards;

- **Int**
  - A keyword, integer value
- declaration of the variables **miles** and **yards** of type **int**
- declarations and statements end with a semicolon
- variable: memory space to hold a value

float kilometers;

- **float**
  - a keyword, real value
- declaration of the variable **kilometers** of type **float**

Variable, Expressions, Assignment (4/7)

```
miles = 26;
yards = 385;
kilometers = 1.609 * (miles + yards / 1760.0);
```

- **Assignment statement**
  - **variable = expression;**
  - Equal sign (=) : assignment operator
  - The value of the expression on the right side of the equal sign is assigned to the variable
- **Expression**
  - On the right side of assignment operators
  - constants , variables, or combinations of operators with variables and constants

Variable, Expressions, Assignment (5/7)

26, 385

- An integer constant
- integer types: **short, int, long, …**

1.609, 1760.0

- A floating-point constant
- Three floating types : **float**, **double**, **long double**
- floating-point constants are automatically of type **double**

Variable, Expressions, Assignment (6/7)

Evaluation of Expression

- Conversion rule
  - Division of two integers results in an integer values. 7/2 is 3
  - A double divided by an integer
    - Integer is automatically converted to double
    - 7.0/2 is 3.5

kilometers = 1.609 * (miles + yards / 1760); bug!!!

printf("\nA marathon is %f kilometers.\n\n", kilometers);

- Control string
- **%f** : format, conversion specification
  - Matched with the remaining argument, the variable **kilometers**

Variable, Expressions, Assignment (7/7)

```
/*the distance of a marathon in kilometers*/
#include <stdio.h>
int main(void)
{
      int  miles, yards;
      float                      kilometers;

      miles = 26;
      yards = 385;
      kilometers = 1.609 * (miles + yards / 1760.0);
      printf(“\nA marathon is %f kilometers.\n\n”, kilometers);
      return 0;
}
```

```
A marathon is 42.195970 kilometers.
```

[Ex.2] Average Score (1/2)

```
#include <stdio.h>
int main(void)
{
    int    score1, score2, score3, avg_score;
    int    num_score;

    score1 = 87; score2 = 93; score3 = 100;
    num_score = 3;
    avg_score = (score1 + score2 + score3) / num_score;
    printf(“Average score: %d\n”, avg_score);
    return 0;
}
```

```
Average score: 93
```

[Ex.2] Average score (2/2)

```
#include <stdio.h>
int main(void)
{
    float    fscore1, fscore2, fscore3;
    float    avg_fscore;
    int      num_score;

    fscore1 = 87.0; fscore2 = 93.0; fscore3 = 100.0;
    num_score = 3;
    avg_fscore = (fscore1 + fscore2 + fscore3) / num_score;
    printf(“Average score: %f\n”, avg_fscore);
    return 0;
}
```

```
Average score: 93.333333
```

Flow of Control: *Alternative actions* (1/5)

if statement

```
#include <stdio.h>
int main(void)
{
    int  a, b;
    ……
    a = 1;
    if ( b == 3 )
        a = 5;
    printf(“%d”, a);
    return 0;
}
```

Flow of Control: Alternative actions (2/5)

```
if (expr)
    statement
```

- If **expr** is nonzero(true), then **statement** is executed;
- otherwise, it is skipped

```
if (b==3)
    a = 5;
```

- **==** : *equal to* operator
- **b==3**
  - logical expression : either the integer value 1 (true) or 0 (false)

Flow of Control: Alternative actions (3/5)

```
#include <stdio.h>
int main(void)
{
        int  a, b;
        b = 3;
        a = 1;
        if ( b == 3 )
                a = 5;
        printf("%d", a);
        return 0;
}
```

5

```
#include <stdio.h>
int main(void)
{
      int    a, b;
      b = 2;
      a = 1;
      if ( b == 3 )
          a = 5;
      printf("%d", a);
      return 0;
}
```

1

Flow of Control: *Alternative actions* (4/5)

```
if (a == 3)
{
      b = 5;
      c = 7;
}
```

Compound statement

- A group of statement surrounded by braces
- a statement, itself

Flow of Control: *Alternative actions* (5/5)

```
if (expr)
      statement1
else
      statement2
```

```
if (cnt == 0)
{
      a = 2;
      b = 3;
      c = 5;
}
else
{
      a = -2;
      b = -3;
      c = -5;
}
```

Flow of Control: Looping (1/4)

while statement

```
#include <stdio.h>
int main(void)
{
        int  i = 1, sum = 0;

        while ( i <= 5 )
        {
                sum = sum + i;
                ++i;
        }
        printf("sum = %d\n", sum);
        return 0;
}
```

Flow of Control: Looping (2/4)

```
while (i <= 5)
{
        sum = sum + i;
        ++i;
}
```

while (expr)
statement

- If **expr** is true, the **compound statement** is executed,
- and control is passed back to the beginning of the **while** loop for the process to start over again
- The **while** loop is repeatedly executed until the test fails

++i;

- **++** : increment operator
- **i = i + 1;**

Flow of Control: Looping (3/4)

```
#include <stdio.h>
int main(void)
{
        int  i = 1, sum = 0;

        while ( i <= 5 )
        {
                sum = sum + i;
                ++i;
        }
        printf("sum = %d\n", sum);
        return 0;
}
```

1+2+3+4+5

```
sum = 15
```

Flow of Control: Looping (4/4)

for statement

```
#include <stdio.h>
int main(void)
{
        int  i, sum = 0;

        for (i=1; i <= 5; ++i )
        {
                sum = sum + i;
        }
        printf("sum = %d\n",
                sum);
        return 0;
}
```

```
for (expr1; expr2; expr3)
      statement
```

is semantically equivalent to

```
expr1;
while (expr2) {
      statement
      expr3;
}
```

C Program is …

- **A sequence of FUNCTIONS**
  - main() function is executed first

- A **FUNCTION** consists of:
  - Declarations
  - Statements

- **Declaration**: variable names and their types
  - **int   miles;**

- **Statement**: data processing or control
  - **miles = 26;**
  - **if (b == 3) { …};**
  - **printf(…);**