Lecture 2 - C Fundamentals

Meng-Hsun Tsai
CSIE, NCKU

```
#include <stdio.h>

int main(void)
{
    printf("Hello NCKU!");
    return 0;
}
```

2.1 Writing a Simple Program

Program: Printing a Pun

```
/* pun.c: Printing a Pun */
#include <stdio.h>

int main(void)
{
  printf("To C, or not to C: that is the question.\n");
  return 0;
}
```

- This program might be stored in a file named `pun.c`.
- The file name doesn't matter, but the `.c` extension is often required.

Compiling and Linking

- Before a program can be executed, three steps are usually necessary:
  - ***Preprocessing.*** The ***preprocessor*** obeys commands that begin with # (known as ***directives***)
  - ***Compiling.*** A ***compiler*** then translates the program into machine instructions (***object code***).
  - ***Linking.*** A ***linker*** combines the object code produced by the compiler with any additional code needed to yield a complete executable program.

Compiling and Linking Using `gcc`

- To compile and link the `pun.c` program under UNIX, enter the following command in a terminal or command-line window:
  ```
  % gcc pun.c
  ```
  where the `%` character is the UNIX prompt.
- Linking is automatic when using `gcc`; no separate link command is necessary.
- After compiling and linking the program, `gcc` leaves the executable program in a file named `a.out` by default.
- The `-o` option lets us choose the name of the file containing the executable program.
  ```
  % gcc -o pun pun.c
  ```

Integrated Development Environments

- An ***integrated development environment (IDE)*** is a software package that makes it possible to edit, compile, link, execute, and debug a program without leaving the environment.

CLion

Qt Creator

Comments

- A ***comment*** begins with `/*` and end with `*/`.

```
/* This is a comment */
```

- Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text.
- Comments may extend over more than one line.

```
/* Name: pun.c
   Purpose: Prints a bad pun.
   Author: K. N. King */
```

Comments (cont.)

- *Warning:* Forgetting to terminate a comment may cause the compiler to ignore part of your program:

```
printf("My ");     /* forgot to close this
comment...
printf("cat ");
printf("has ");    /* so it ends here */
printf("fleas");
```

Comments (cont.)

- Comments can also be written in the following way:

```
// This is a comment
```

- This style of comment ends automatically at the end of a line.
- Advantages of `//` comments:
  - Safer: there's no chance that an unterminated comment will accidentally consume part of a program.
  - Multiline comments stand out better.

Layout of a C Program

- A C program is a series of ***tokens*.**
- Tokens include:
  - Identifiers
  - Keywords
  - Operators
  - Punctuation
  - Constants
  - String literals

Example: Tokens in a Statement

- The statement

  ```
  printf("Height: %d\n", height);
  ```

  consists of seven tokens:

`printf`	Identifier
`(`	Punctuation
`"Height: %d\n"`	String literal
`,`	Punctuation
`height`	Identifier
`)`	Punctuation
`;`	Punctuation

Space between Tokens

- The amount of space between tokens usually isn't critical.
- The whole program can't be put on one line, because each preprocessing directive requires a separate line.
- Compressing programs in this fashion isn't a good idea.

```
#include <stdio.h>
#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)
int main(void){float fahrenheit,celsius;printf(
"Enter Fahrenheit temperature: ");scanf("%f", &fahrenheit);
celsius=(fahrenheit-FREEZING_PT)*SCALE_FACTOR;
printf("Celsius equivalent: %.1f\n", celsius);return 0;}
```

Advantages of Adding Spaces between Tokens

- In fact, adding spaces and blank lines to a program can make it easier to read and understand.
- C allows any amount of space—blanks, tabs, and new-line characters—between tokens.
- Consequences for program layout:
  - *Statements can be divided* over any number of lines.
  - *Space between tokens* (such as before and after each operator, and after each comma) makes it easier for the eye to separate them.
  - *Indentation* can make nesting easier to spot.
  - *Blank lines* can divide a program into logical units.

Pitfalls When Adding Spaces within a Token

- Although extra spaces can be added between tokens, it's not possible to add space within a token without changing the meaning of the program or causing an error.
- Writing

```
fl oat fahrenheit, celsius;  /*** WRONG ***/
```

  produces an error when the program is compiled.

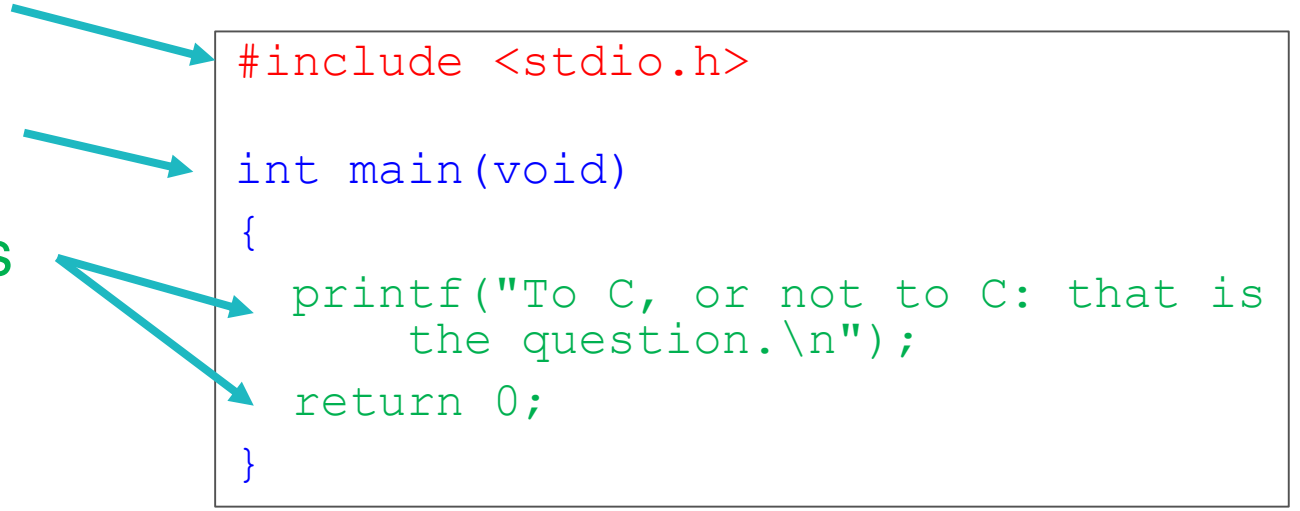
- Splitting a string over two lines is illegal:

```
printf("To C, or not to C:
that is the question.\n");
  /*** WRONG ***/
```

2.2 The General Form of a Simple Program

The General Form of a Simple Program

- Even the simplest C programs rely on three key language features:
  - Directives
  - Functions
  - Statements



```
#include <stdio.h>

int main(void)
{
  printf("To C, or not to C: that is
      the question.\n");
  return 0;
}
```

Directives

- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- Example:

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

- `<stdio.h>` is a ***header*** containing information about C's standard I/O library.
- Directives always begin with a `#` character.
- By default, directives are one line long; there's no semicolon or other special marker at the end.

A Simple Example using *#include*

header.h

```
1  printf("header.h\n");
```

main.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4  #include "header.h"
5        printf(“main.c\n”);
6        return 0;
7  }
```

```
> gcc -o main main.c
> ./main
header.h
main.c
```

Output of Preprocessor

```
$ gcc -E  main.c
…
# 797 "/usr/include/stdio.h" 3 4
}
# 2 "main.c" 2

# 2 "main.c"
int main(void)
{
# 1 "header.h" 1
printf("header.h\n");
# 5 "main.c" 2
 printf("main.c\n");
 return 0;
}
```

From gcc's man page:

-E Stop after the preprocessing stage; do not run the compiler.

Where is *stdio.h*?

In Cygwin

```
$ find /usr -name stdio.h
/usr/i686-w64-mingw32/sys-root/mingw/include/stdio.h
/usr/include/stdio.h
/usr/include/sys/stdio.h
/usr/lib/gcc/i686-w64-mingw32/6.4.0/include/c++/tr1/stdio.h
/usr/lib/gcc/i686-w64-mingw32/6.4.0/include/ssp/stdio.h
/usr/lib/gcc/x86_64-pc-cygwin/6.4.0/include/c++/tr1/stdio.h
/usr/lib/gcc/x86_64-pc-cygwin/6.4.0/include/ssp/stdio.h
```

What's Inside *stdio.h*?

```
$ cat /usr/include/stdio.h
...
int     _EXFUN(printf, (const char *__restrict, ...)
            _ATTRIBUTE ((__format__ (__printf__, 1, 2))));
...
$ gcc -E hello.c
...
int __attribute__((__cdecl__)) printf (const char *restrict, ...) __attribute__
    ((__format__ (__printf__, 1, 2)))
...
# 5 "hello.c"
int main(void)
{
 printf("Hello NCKU\n");
 return 0;
}
```

Include declaration of *printf()* manually

hello.c

```
1  /* #include <stdio.h> */
2  int __attribute__((__cdecl__)) printf (const char *restrict, ...)
   __attribute__ ((__format__ (__printf__, 1, 2)));
3
4  int main(void)
5  {
6          printf("Hello NCKU!\n");
7          return 0;
8  }
```

```
$ gcc -o hello hello.c
$ ./hello
Hello NCKU!
```

Functions

- A ***function*** is a series of statements that have been grouped together and given a name.
- ***Library functions*** are provided as part of the C implementation.
- A function that computes a value uses a `return` statement to specify what value it "returns":

```
return x + 1;
```

The `main` Function

- The `main` function is mandatory.
- `main` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value 0 indicates normal program termination.
- If there's no `return` statement at the end of the `main` function, many compilers will produce a warning message.

Getting Return Value in Unix

```
$ cat return_minus1.c
int main(void)
{
        return -1;
}
$ gcc -o return_minus1 return_minus1.c
$ echo $?
0
$ ./return_minus1
$ echo $?
255
$ echo $?
0
```

Statements

- A ***statement*** is a command to be executed when the program runs.
- `pun.c` uses only two kinds of statements. One is the `return` statement; the other is the ***function call.***
- Asking a function to perform its assigned task is known as ***calling*** the function.
- `pun.c` calls `printf` to display a string:

```
printf("To C, or not to C: that is the question.\n");
```

- C requires that each statement end with a semicolon.
  - There's one exception: the compound statement.

```
{ statement-1;
  statement-2;  }
```

Printing Strings

- When the `printf` function displays a ***string literal***—characters enclosed in double quotation marks—it doesn't show the quotation marks.
- `printf` doesn't automatically advance to the next output line when it finishes printing.
- To make `printf` advance one line, include `\n` (the ***new-line character***) in the string to be printed.
- One `printf()` call could be replaced by two `printf()` calls:

```
printf("To C, or not to C: ");
printf("that is the question.\n");
```

2.3 Variables and Assignment

Variables and Assignment

- Most programs need a way to store data temporarily during program execution.
- These storage locations are called ***variables*.**

Types

- Every variable must have a ***type***, which decides how the variable is stored and what operations can be performed.
- C has a wide variety of types, including `int` and `float`.
- A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or –2553.
- Also, a `float` (short for *floating-point*) variable can store numbers with digits after the decimal point, like 379.125.
- Drawbacks of `float` variables:
  - Slower arithmetic
  - Approximate nature of `float` values

```
float value = 0;
for (int i=0; i<100; i++)
    value += 0.03;
printf("%f\n", value);
```

```
$ ./float
2.999998
```

Declarations

- Variables must be ***declared*** before they are used.
- One or more variables can be declared at a time:

```
int height, length, width, volume;
float profit;
```

Assignment

- A variable can be given a value by means of ***assignment:***

```
height = 8;
```

  The number `8` is said to be a ***constant.***

- Before a variable can be assigned a value—or used in any other way—it must first be declared.
- A constant assigned to a `float` variable usually contains a decimal point:

```
profit = 2150.48;
```

- It's best to append the letter `f` to a floating-point constant if it is assigned to a `float` variable:

```
profit = 2150.48f;
```

Assignment

- An `int` variable is normally assigned a value of type `int`, and a `float` variable is normally assigned a value of type `float`.
- Mixing types (such as assigning a `float` value to an `int` variable) is possible but not always safe.
- Once a variable has been assigned a value, it can be used to help compute the value of another variable:

```
length = 12;
width = 10;
area = length * width;
```

- The right side of an assignment can be a formula (or ***expression,*** in C terminology) involving constants, variables, and operators.

Printing the Value of a Variable

- To print the message

  `Height:` *h*

  where *h* is the current value of the `height` variable, we'd use the following call of `printf`:

  ```
  printf("Height: %d\n", height);
  ```

- `%d` is a placeholder indicating where the value of `height` is to be filled in.
- `%d` works only for `int` variables; to print a `float` variable, use `%f` instead.

Printing the Value of a Variable (cont.)

- By default, `%f` displays a number with six digits after the decimal point.
- To force `%f` to display *p* digits after the decimal point, put `.`*p* between `%` and `f`.
- To print the line

  ```
  Profit: $2150.48
  ```

  use the following call of `printf`:

  ```
  printf("Profit: $%.2f\n", profit);
  ```

- There's no limit to the number of variables that can be printed:

  ```
  printf("Height: %d  Length: %d\n", height, length);
  ```

Program: Computing the Dimensional Weight of a Box

- Shipping companies often charge extra for boxes that are large but very light, basing the fee on volume instead of weight.
- The usual method to compute the "dimensional weight" is to divide the volume by 166 (the allowable number of cubic inches per pound).
- Division is represented by `/` in C, so the obvious way to compute the dimensional weight would be

```
weight = volume / 166;
```

Program: Computing the Dimensional Weight of a Box (cont.)

- In C, however, when one integer is divided by another, the answer is "truncated" (rounded down): all digits after the decimal point are lost.
  - The volume of a 12" × 10" × 8" box will be 960 cubic inches.
  - Dividing by 166 gives 5 instead of 5.783.
- However, the shipping company expects to round up. One solution is to add 165 to the volume before dividing by 166:

```
weight = (volume + 165) / 166;
```

- A volume of 166 would give a weight of 331/166, or 1, while a volume of 167 would yield 332/166, or 2.

Program: Computing the Dimensional Weight of a Box (cont.)

dweight.c

```
#include <stdio.h>
int main(void)
{
  int height, length, width, volume, weight;

  height = 8;
  length = 12;
  width = 10;
  volume = height * length * width;
  weight = (volume + 165) / 166;

  printf("Dimensions: %dx%dx%d\n", length, width, height);
  printf("Volume (cubic inches): %d\n", volume);
  printf("Dimensional weight (pounds): %d\n", weight);

  return 0;
}
```

```
Dimensions: 12x10x8
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

Initialization

- Some variables are automatically set to zero when a program begins to execute, but most are not.
- A variable that doesn't have a default value and hasn't yet been assigned a value by the program is said to be ***uninitialized***.
- Attempting to access the value of an uninitialized variable may yield an unpredictable result.
- The initial value of a variable may be included in its declaration:

```
int height = 8;
```

  The value `8` is said to be an ***initializer***.

- Any number of variables can be initialized in the same declaration:

```
int height = 8, length = 12, width = 10;
```

Printing Expressions

- `printf` can display the value of any numeric expression.
- The statements

```
volume = height * length * width;
printf("%d\n", volume);
```

  could be replaced by

```
printf("%d\n", height * length * width);
```

2.4 Assigning from Input or Constants

Reading Input

- `scanf` requires a ***format string*** to specify the appearance of the input data.
- Using `%d` to read an `int` value and store into variable `i`:

```
scanf("%d", &i);
```

- Using `%f` to read a `float` value and store into variable `x`:

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

- The `&` symbol obtains the address of a variable in memory for `scanf` to store the input value.

```
int x = 1, y = 2;
printf("%d %d\n", x,y);
printf("%u %u\n", &x,&y);
```

```
1 2
4294953980 4294953976
```

Program: Computing the Dimensional Weight of a Box (Revisited)

dweight2.c

```
 1 #include <stdio.h>
 2 int main(void)
 3 {
 4   int height, length, width,
       volume, weight;
 5
 6   printf("Enter box height: ");
 7   scanf("%d", &height);
 8   printf("Enter box length: ");
 9   scanf("%d", &length);
10   printf("Enter box width: ");
11   scanf("%d", &width);
12   volume = height * length
       * width;
13   weight = (volume + 165) / 166;
14
15   printf("Volume: %d\n", volume);
16   printf("Dimensional weight:
       %d\n", weight);
17
18   return 0;
19}
```

```
Enter box height: 8
Enter box length: 12
Enter box width: 10
Volume: 960
Dimensional weight: 6
```

Program: Computing the Dimensional Weight of a Box (Revisited) (cont.)

- `dweight2.c` is an improved version of the dimensional weight program in which the user enters the dimensions.
- Each call of `scanf` is immediately preceded by a call of `printf` that displays a ***prompt.***
- Note that a prompt shouldn't end with a new-line character.

Defining Names for Constants

- `dweight.c` and `dweight2.c` rely on the constant 166, whose meaning may not be clear to someone reading the program.
- Using a feature known as ***macro definition,*** we can name this constant:

```
#define INCHES_PER_POUND 166
```

- When a program is compiled, the preprocessor replaces each macro by the value that it represents.
- During preprocessing, the statement

```
weight = (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND;
```

will become

```
weight = (volume + 166 - 1) / 166;
```

Defining Names for Constants (cont.)

- The value of a macro can be an expression:

```
#define RECIPROCAL_OF_PI (1.0f / 3.14159f)
```

- If it contains operators, the expression should be enclosed in parentheses.
- Using only upper-case letters in macro names is a common convention.

Program: Converting from Fahrenheit to Celsius

celsius.c

```
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
    float fahrenheit, celsius;

    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);

    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

    printf("Celsius equivalent: %.1f\n", celsius);

    return 0;
}
```

```
Enter Fahrenheit temperature: 100
Celsius equivalent: 37.8
```

Program: Converting from Fahrenheit to Celsius (cont.)

- The `celsius.c` program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.
- The program will allow temperatures that aren't integers.
- Defining `SCALE_FACTOR` to be `(5.0f / 9.0f)` instead of `(5 / 9)` is important.
- Note the use of `%.1f` to display `celsius` with just one digit (rounded) after the decimal point.

```
printf("%f\n", 5/9);
printf("%f\n", 5.0/9.0);
```

```
0.000000
0.555556
```

Identifiers

- Names for variables, functions, macros, and other entities are called ***identifiers.***
- An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

```
times10   get_next_char   _done
```

  It's usually best to avoid identifiers that begin with an underscore.

- Examples of illegal identifiers:

```
10times   get-next-char
```

Identifiers (cont.)

- C is ***case-sensitive:*** it distinguishes between upper-case and lower-case letters in identifiers.
- For example, the following identifiers are all different:

```
job   joB   jOb   jOB   Job   JoB   JOb   JOB
```

- Many programmers use only lower-case letters in identifiers (other than macros), with underscores inserted for legibility:

```
symbol_table   current_page   name_and_address
```

- Other programmers use an upper-case letter to begin each word within an identifier:

```
symbolTable   currentPage   nameAndAddress
```

- C places no limit on the maximum length of an identifier.

Keywords

- The following 37 ***keywords*** can't be used as identifiers:

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

*C99 only

2.5 Arithmetic Operators

Operators

- C emphasizes expressions rather than statements.
- Expressions are built from variables, constants, and operators.
- C has a rich collection of operators, including
  - arithmetic operators
  - relational operators
  - logical operators
  - assignment operators
  - increment and decrement operators

  and many others

Arithmetic Operators

- C provides five binary ***arithmetic operators:***
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `%` remainder
- An operator is ***binary*** if it has two operands.
- There are also two ***unary*** arithmetic operators:
  - `+` unary plus
  - `-` unary minus

Unary Arithmetic Operators

- The unary operators require one operand:

```
i = +1;
j = -i;
```

- The unary + operator does nothing. It's used primarily to emphasize that a numeric constant is positive.

Binary Arithmetic Operators

- The value of `i % j` is the remainder when `i` is divided by `j`.

  `10 % 3` has the value 1, and `12 % 4` has the value 0.

- Binary arithmetic operators—with the exception of `%`—allow either integer or floating-point operands, with mixing allowed.

- When `int` and `float` operands are mixed, the result has type `float`.

  `9 + 2.5f` has the value 11.5, and `6.7f / 2` has the value 3.35.

The / and % Operators

- The / and % operators require special care:
  - When both operands are integers, / "truncates" the result. The value of 1 / 2 is 0, not 0.5.
  - The % operator requires integer operands; if either operand is not an integer, the program won't compile.

```
x = 10 % 3.0;
```

```
error : invalid operands to binary % (have 「int」 and 「double」)
  x = 10 % 3.0;
         ^
```

The / and % Operators (cont.)

- Using zero as the right operand of either / or % causes undefined behavior.
  - Division by zero using integer arithmetic typically causes a program to terminate prematurely.
  - In floating-point arithmetic, some implementations allow division by zero, in which case positive or negative infinity is displayed as INF or -INF, respectively.

```
printf("2.0 / 0 = %f\n", 2.0/0);
printf("2 / 0 = %d\n", 2/0);
```

```
$ ./division_and_remainder
2.0 / 0 = inf
Floating exception (core dumped)
```

The / and % Operators (cont.)

- The behavior when / and % are used with negative operands is
  - ***implementation-defined*** in C89.
  - always truncated toward zero and the value of i % j has the same sign as i in C99.

```
printf("%d\n", 7 % 3);
printf("%d\n", 7 % -3);
printf("%d\n", -7 % 3);
printf("%d\n", -7 % -3);
```

```
1
1
-1
-1
```

Implementation-Defined Behavior

- The C standard deliberately leaves parts of the language unspecified.
- Leaving parts of the language unspecified reflects C's emphasis on efficiency, which often means matching the way that hardware behaves.
- It's best to avoid writing programs that depend on implementation-defined behavior.

Operator Precedence

- Does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`" or "multiply `j` and `k`, then add `i`"?
- One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`.
- If the parentheses are omitted, C uses ***operator precedence*** rules to determine the meaning of the expression.

Operator Precedence (cont.)

- The arithmetic operators have the following relative precedence:

  Highest: `+ -` (unary)
  `* / %`
  Lowest: `+ -` (binary)

- Examples:

  `i + j * k` is equivalent to `i + (j * k)`

  `-i * -j` is equivalent to `(-i) * (-j)`

  `+i + j / k` is equivalent to `(+i) + (j / k)`

Operator Associativity

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be ***left associative*** if it groups from left to right.
- The binary arithmetic operators (`*`, `/`, `%`, `+`, and `-`) are all left associative, so

  `i - j - k` is equivalent to `(i - j) - k`
  `i * j / k` is equivalent to `(i * j) / k`

- An operator is ***right associative*** if it groups from right to left.
- The unary arithmetic operators (`+` and `-`) are both right associative, so

  `- + i` is equivalent to `-(+i)`

Program: Computing a UPC Check Digit

- Most goods sold in U.S. and Canadian stores are marked with a Universal Product Code (UPC):
- Meaning of the digits underneath the bar code:
  - First digit: Type of item (0)
  - First group of five digits: Manufacturer (13800)
  - Second group of five digits: Product (including package size) (15173)
  - Final digit: Check digit (5), used to help identify an error in the preceding digits

Program: Computing a UPC Check Digit (cont.)

- How to compute the check digit (e.g. 0 13800 15173 5):
  1. Add the first, third, fifth, seventh, ninth, and eleventh digits.
     0 + 3 + 0 + 1 + 1 + 3 = 8
  2. Add the second, fourth, sixth, eighth, and tenth digits.
     1 + 8 + 0 + 5 + 7 = 21
  3. Multiply the first sum by 3 and add it to the second sum.
     3 * 8 + 21 = 45
  4. Subtract 1 from the total.
     45 – 1 = 44
  5. Compute the remainder when the adjusted total is divided by 10. Subtract the remainder from 9.
     9 – 44 %10 = **5**

Program: Computing a UPC Check Digit (cont.)

upc.c

```
#include <stdio.h>
int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");
  scanf("%1d", &d);
  printf("Enter first group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Enter second group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);
  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Check digit: %d\n", 9 - ((total - 1) % 10));
  return 0;
}
```

Program: Computing a UPC Check Digit (cont.)

- The `upc.c` program asks the user to enter the first 11 digits of a UPC, then displays the corresponding check digit:

```
Enter the first (single) digit: 0
Enter first group of five digits: 13800
Enter second group of five digits: 15173
Check digit: 5
```

- The program reads each digit group as five one-digit numbers.
- To read single digits, we use `scanf` with the `%1d` conversion specification.

2.6 Assignment Operators

Assignment Operators

- ***Simple assignment:*** used for storing a value into a variable
- ***Compound assignment:*** used for updating a value already stored in a variable

Simple Assignment

- The effect of the assignment *v* = *e* is to evaluate the expression *e* and copy its value into *v*.
- *e* can be a constant, a variable, or a more complicated expression:

```
i = 5;                  /* i is now 5  */
j = i;                  /* j is now 5  */
k = 10 * i + j;         /* k is now 55 */
```

Simple Assignment (cont.)

- If *v* and *e* don't have the same type, then the value of *e* is converted to the type of *v* as the assignment takes place:

```
int i;
float f;
i = 72.99f;     /* i is now 72 */
f = 136;        /* f is now 136.0 */
```

- In many programming languages, assignment is a statement; in C, however, assignment is an operator, just like `+`.
- The value of an assignment *v* = *e* is the value of *v* after the assignment.
  - The value of `i = 72.99f` is 72 (not 72.99).

Side Effects

- An operator that modifies one of its operands is said to have a ***side effect***.
- The simple assignment operator has a side effect: it modifies its left operand.
- Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.
- Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

- The `=` operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```

Side Effects (cont.)

- Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;
f = i = 33.3f;
```

`i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3).

- "Embedded assignments" can make programs hard to read:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k); /* prints "1 1 2" */
```

- They can also be a source of subtle bugs.

Lvalues

- The assignment operator requires an ***lvalue*** as its left operand.
- An lvalue represents an object stored in computer memory, not a constant or the result of a computation.
- Variables are lvalues; expressions such as `10` or `2 * i` are not.
- It's illegal to put any other kind of expression on the left side of an assignment expression:

```
12 = i;          /*** WRONG ***/
i + j = 0;       /*** WRONG ***/
-i = j;          /*** WRONG ***/
```

- The compiler will produce an error message such as *"invalid lvalue in assignment."*

Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common. Example:

```
i = i + 2;
```

- Using the += compound assignment operator, we simply write:

```
i += 2;    /* same as i = i + 2; */
```

Compound Assignment (cont.)

- There are nine other compound assignment operators, including the following:

  `-=` `*=` `/=` `%=`

- All compound assignment operators work in much the same way:

  v `+=` *e* adds *v* to *e*, storing the result in *v*

  v `-=` *e* subtracts *e* from *v*, storing the result in *v*

  v `*=` *e* multiplies *v* by *e*, storing the result in *v*

  v `/=` *e* divides *v* by *e*, storing the result in *v*

  v `%=` *e* computes the remainder when *v* is divided by *e*, storing the result in *v*

Compound Assignment (cont.)

- *v* += *e* isn't "equivalent" to *v* = *v* + *e*.
- One problem is operator precedence: `i *= j + k` isn't the same as `i = i * j + k`.
- There are also rare cases in which *v* += *e* differs from *v* = *v* + *e* because *v* itself has a side effect.

```
a[i++]+= 2;
```

- Similar remarks apply to the other compound assignment operators.
- When using the compound assignment operators, be careful not to switch the two characters that make up the operator.

```
i =+ j
```

- Although `i =+ j` will compile, it is equivalent to `i = (+j)`, which merely copies the value of `j` into `i`.

Increment and Decrement Operators

- Two of the most common operations on a variable are "incrementing" (adding 1) and "decrementing" (subtracting 1):

```
i = i + 1;
j = j - 1;
```

- Incrementing and decrementing can be done using the compound assignment operators:

```
i += 1;
j -= 1;
```

- C provides special `++` (***increment***) and `--` (***decrement***) operators.
- They can be used as ***prefix*** operators (`++i` and `--i`) or ***postfix*** operators (`i++` and `i--`).

Increment and Decrement Operators (cont.)

- Evaluating the expression `++i` (a "pre-increment") yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

- Evaluating the expression `i++` (a "post-increment") produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

Increment and Decrement Operators (cont.)

- `++i` means "increment `i` immediately," while `i++` means "use the old value of `i` for now, but increment `i` later."
- How much later? The C standard doesn't specify a precise time, but it's safe to assume that `i` will be incremented before the next statement is executed.

Increment and Decrement Operators (cont.)

- The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);    /* prints "i is 0" */
printf("i is %d\n", i);      /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 0" */
```

Increment and Decrement Operators (cont.)

- When `++` or `--` is used more than once in the same expression, the result can often be hard to understand.
- Example:

```
i = 1;
j = 2;
k = ++i + j++;
```

i = 2
j = 3
k = 4

The last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

Increment and Decrement Operators (cont.)

- In contrast, executing the statements

```
i = 1;
j = 2;
k = i++ + j++;
```

i=2
j=3
k=3

will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

2.7 Expression Evaluation

Expression Evaluation

Table of operators discussed so far:

Precedence	*Name*	*Symbol(s)*	*Associativity*
1	increment (postfix) decrement (postfix)	++ −−	left
2	increment (prefix) decrement (prefix) unary plus unary minus	++ −− + −	right
3	multiplicative	* / %	left
4	additive	+ −	left
5	assignment	= *= /= %= += −=	right

Expression Evaluation (cont.)

- The table can be used to add parentheses to an expression that lacks them.
- Starting with the operator with highest precedence, put parentheses around the operator and its operands.
- Example:

```
a = b += c++ - d + --e / -f
```

	Precedence level
`a = b += (c++) - d + --e / -f`	1
`a = b += (c++) - d + (--e) / (-f)`	2
`a = b += (c++) - d + ((--e) / (-f))`	3
`a = b += (((c++) - d) + ((--e) / (-f)))`	4
`(a = (b += (((c++) - d) + ((--e) / (-f)))))`	5

Order of Subexpression Evaluation

- The value of an expression may depend on the order in which its subexpressions are evaluated.
- C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical and, logical or, conditional, and comma operators).
- In the expression `(a + b) * (c - d)` we don't know whether `(a + b)` will be evaluated before `(c - d)`.

Order of Subexpression Evaluation (cont.)

- Most expressions have the same value regardless of the order in which their subexpressions are evaluated.
- However, this may not be true when a subexpression modifies one of its operands:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

- The effect of executing the second statement is undefined.
- Avoid writing expressions that access the value of a variable and also modify the variable elsewhere in the expression.
- Some compilers may produce a warning message such as *"operation on 'a' may be undefined"* when they encounter such an expression.

Order of Subexpression Evaluation (cont.)

- To prevent problems, it's a good idea to avoid using the assignment operators in subexpressions.
- Instead, use a series of separate assignments:

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of `c` will always be 6.

Order of Subexpression Evaluation (cont.)

- Besides the assignment operators, the only operators that modify their operands are increment and decrement.
- When using these operators, be careful that an expression doesn't depend on a particular order of evaluation. Example:

```
i = 2;
j = i * i++;
```

- It's natural to assume that `j` is assigned 4. However, `j` could just as well be assigned 6 instead:

1. The second operand (the original value of i) is fetched, then `i` is incremented.
2. The first operand (the new value of `i`) is fetched.
3. The new and old values of `i` are multiplied, yielding 6.

Undefined Behavior

- Statements such as `c = (b = a + 2) - (a = 1);` and `j = i * i++;` cause ***undefined behavior.***
- Possible effects of undefined behavior:
  - The program may behave differently when compiled with different compilers.
  - The program may not compile in the first place.
  - If it compiles it may not run.
  - If it does run, the program may crash, behave erratically, or produce meaningless results.
- Undefined behavior should be avoided.

Expression Statements

- C has the unusual rule that any expression can be used as a statement. Example:

```
++i;
```

  `i` is first incremented, then the new value of `i` is fetched but then discarded.

- Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect:

```
i = 1;          /* useful */
i--;            /* useful */
i * j - 1;      /* not useful */
```

Expression Statements (cont.)

- A slip of the finger can easily create a "do-nothing" expression statement.
- For example, instead of entering

```
i = j;
```

  we might accidentally type

```
i + j;
```

- Some compilers can detect meaningless expression statements; you'll get a warning such as *"statement with no effect."*

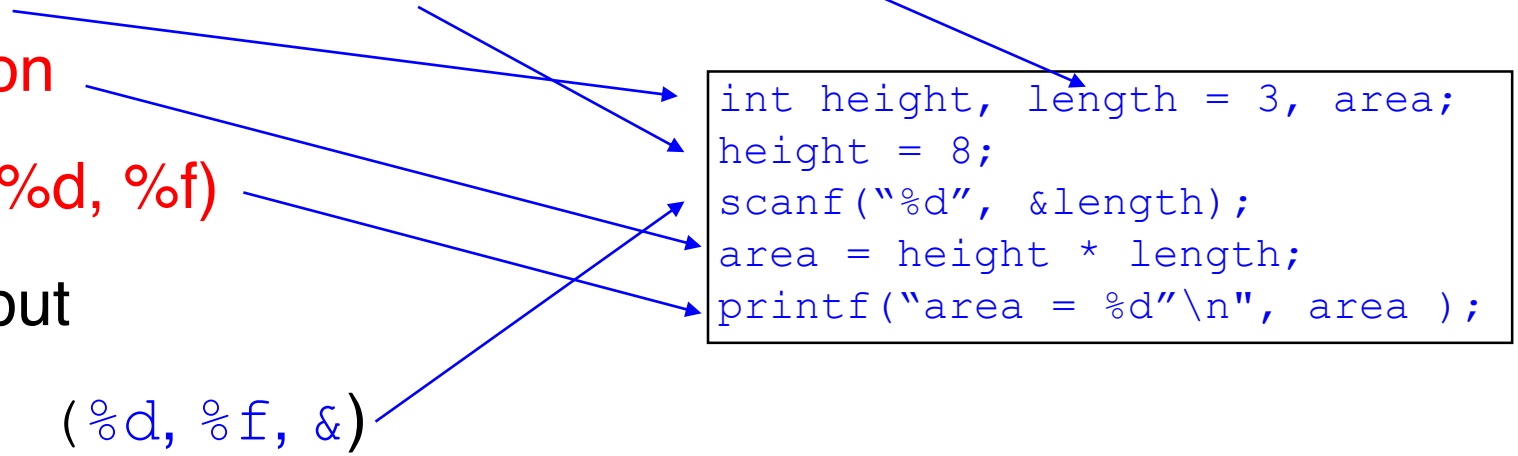
A Quick Review to This Lecture

- Three key features in a C program
  - Directive / Function / Statement
- Three stages of gcc
  - Preprocessing / Compiling / Linking
- Statements
  - Function calls (`printf()`, `scanf()`)
  - `return`
- Comments (`/*  */`, `//`)

```
/* This is a comment */
#include <stdio.h>
int main(void)
{
   printf("Hello NCKU!");
   return 0;     // main ends
}
```

A Quick Review to This Lecture (cont.)

- Variables and Assignments
  - Types (`int`, `float`)
  - Declarations / Assignments / Initialization
  - Expression
  - Printing (%d, %f)
- Reading Input
  - `scanf()` (`%d`, `%f`, `&`)



A Quick Review to This Lecture (cont.)

- Defining Names for Constants
  - `#define` macro
- Identifiers
  - Letter, underscore, digit
  - 37 keywords (`int`, `float`, `return`, `void`, ~~`main`~~)
- Layout of a C Program
  - Tokens
  - Space between Tokens / ~~within a Token~~

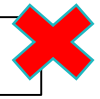
```
#define PI 3.14159f

area_123 = _r * _r * PI;
```

A Quick Review to This Lecture (cont.)

- Arithmetic Operators
  Unary: `+` `-` Binary: `+` `-` `*` `/` `%`
- When both operands are integers, `/` "truncates" the result. The value of `1 / 2` is 0, not 0.
- If either operand of the % operator is not an integer, the program won't compile.

```
x = 10 % 3.0;
```



- Using zero as the right operand of either `/` or `%` causes undefined behavior. (e.g., integer -> terminate, float -> return INF)
- `/` and `%` are used with negative operands
  - always truncated toward zero and the value of `i % j` has the same sign as `i`.

```
x = 2 / 0;
```

terminate

```
x = 2.0 / 0;
```

INF

```
 7 % 3 = 1     7 % -3 = 1
-7 % 3 = -1   -7 % -3 = -1
```

A Quick Review to This Lecture (cont.)

- Assignment Operators

  Simple: `=` Compound: `+=` `-=` `*=` `/=` `%=`

- *v* = *e* evaluates the expression *e* and copies (or converts) its value into *v*.

- Operators that modifies one of its operands is said to have a ***side effect:***

  Assignment Operators / Increment and Decrement Operators

- Unexpected results in chained assignments with type conversion

```
f = i = 33.3f;    // (float -> int -> float)
```

- "Embedded assignments" makes programs hard to read:

```
k = 1 + (j = i);
```

A Quick Review to This Lecture (cont.)

- Lvalue: something (e.g., variables) that can be left operand of an assignment operator.
- Increment and Decrement Operators

  Prefix: `++i` `--i`

  Increment/decrement immediately

  Postfix: `i++` `i--`

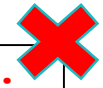
  Increment/decrement before the next statement is executed.
- Using `++` or `--` more than once makes the program hard to understand.

```
k = ++i + j++;
```

A Quick Review to This Lecture (cont.)

- Expression Evaluation (see p.32 for Precedence and Associativity)
  - C doesn't define the order in which subexpressions are evaluated (exception: logical and, logical or, conditional, and comma operators).
  - Undefined Behavior

```
c = (b = a + 2) - (a = 1);
```



    - accessing a variable and also modifying the variable
    - accessing an incremented/decremented variable twice

```
j = i * i++;
```

- Expression Statement
  - C allows any expression to be used as a statement.

```
++i;
```

  - Use it only when the expression has a side effect