Lecture 2 - C Fundamentals

Meng-Hsun Tsai
CSIE, NCKU

```
#include <stdio.h>

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

Program: 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.
- The preprocessor is usually integrated with the compiler.

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.

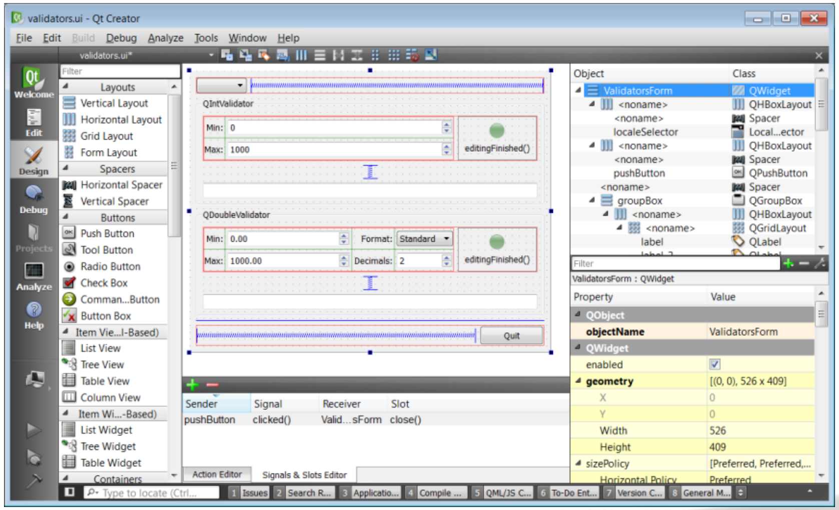
  ```
  % gcc -o pun pun.c
  ```

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



Object	Class
ValidatorsForm	QWidget
<noname>	QHBoxLayout
<noname>	Spacer
localeSelector	Local...ector
<noname>	QHBoxLayout
<noname>	Spacer
pushButton	QPushButton
<noname>	Spacer
groupBox	QGroupBox
<noname>	QHBoxLayout
<noname>	QGridLayout
label	QLabel

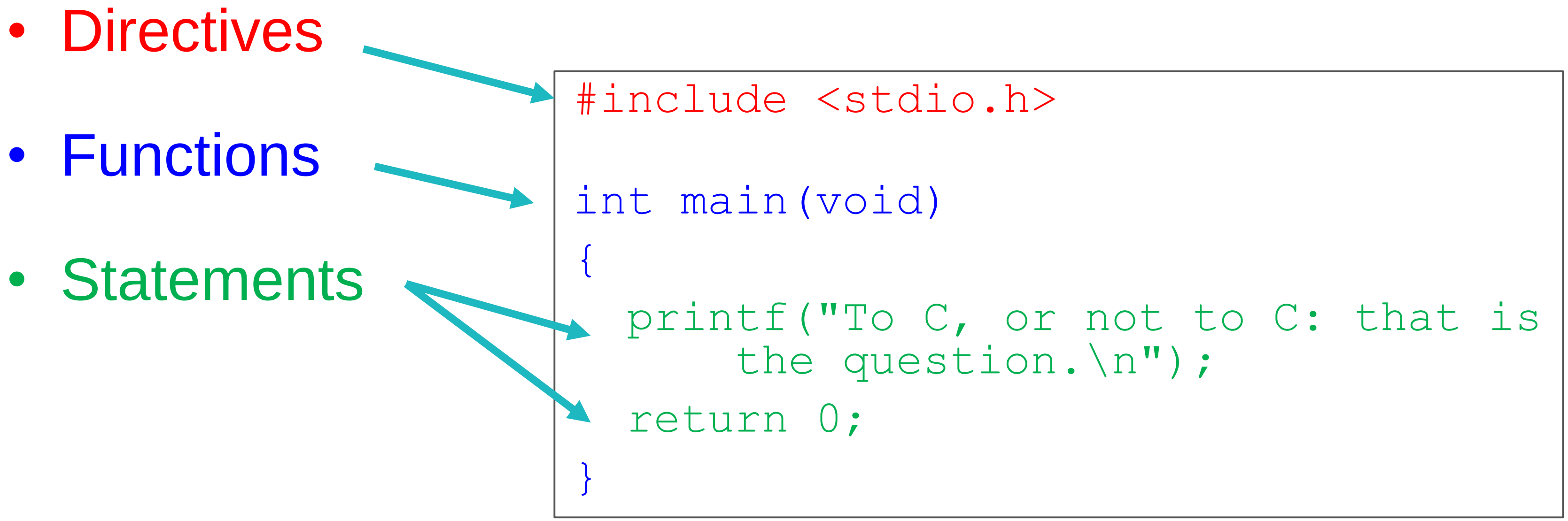
Sender	Signal	Receiver	Slot
pushButton	clicked()	Valid...sForm	close()

ValidatorsForm : QWidget

Property	Value
QObject	
objectName	ValidatorsForm
QWidget	
enabled	☑
geometry	[(0, 0), 526 x 409]
X	0
Y	0
Width	526
Height	409
sizePolicy	[Preferred, Preferred,...
Horizontal Policy	Preferred

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

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

main.c

```
#include <stdio.h>
int main(void)
{
#include "header.h"
    printf(“main.c\n”);
    return 0;
}
```

```
> 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

```
/* #include <stdio.h> */
int __attribute__((__cdecl__)) printf (const char *restrict, ...) __attribute__ ((__format__ (__printf__, 1, 2)));

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

```
$ 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");
```

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 in C99

- In C99, 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.

Variables and Assignment

- Most programs need to 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;
```

- Before C99, declarations must precede statements:

```
int main(void)
{
   declarations
   statements
}
```

- In C99, declarations don’t have to come before statements.

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);
```

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

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

  printf("Enter box height: ");
  scanf("%d", &height);
  printf("Enter box length: ");
  scanf("%d", &length);
  printf("Enter box width: ");
  scanf("%d", &width);
  volume = height * length
      * width;
  weight = (volume + 165) / 166;

  printf("Volume: %d\n", volume);
  printf("Dimensional weight:
      %d\n", weight);

  return 0;
}
```

```
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

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

*C99 only

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 ***/
```

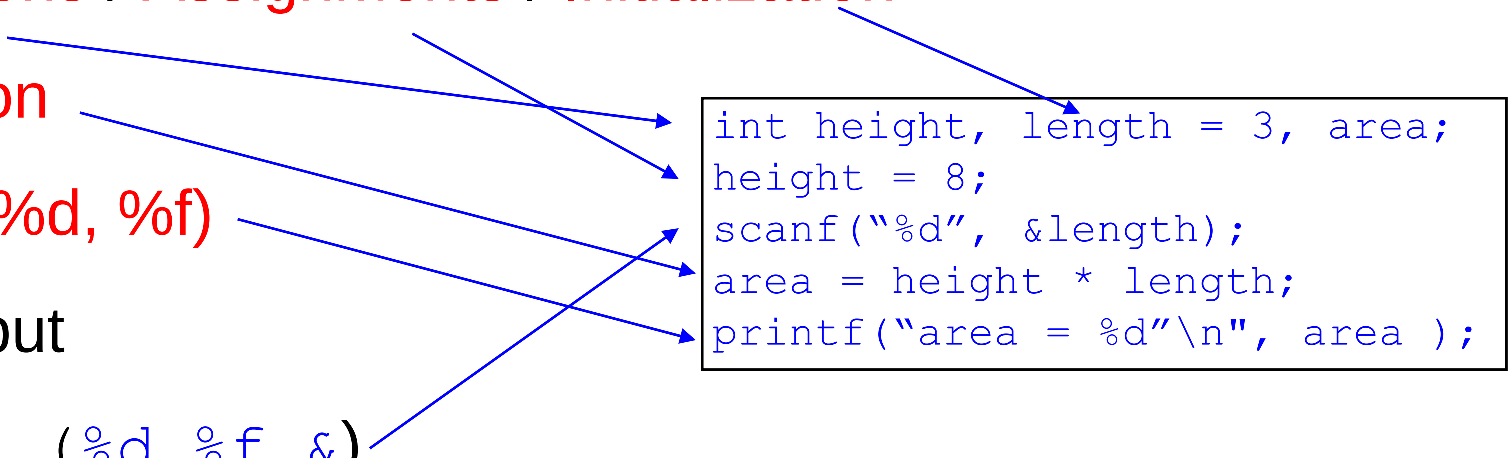
A Quick Review to This Lecture

- 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, &)



```
int height, length = 3, area;
height = 8;
scanf("%d", &length);
area = height * length;
printf("area = %d\n", area );
```

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;
```