Lecture – Getting start

A single-input C program

- ```
  // This program computes 1+2+...+n.   ①
  #include <stdio.h>                    ②
  int main(void)
  {
     int n,sum,i;
     printf("Enter an integer >= 0: ");
     scanf("%d",&n);
     sum=0;
     for (i=1;i<=n;i++) sum+=i;
     printf("1+2+...+%d=%d\n",n,sum);
  }
  ```

  ① Two ways to comment on the program:

// *comment*	// single line (C99,C++)
/* *comment* ***/**	// multiple lines

  A **//** comment is removed by the compiler entirely.
  A **/**/** comment is replaced by a space. Thus,
  su/*initialize*/m=0;
  is illegal, as it is interpreted as
  su m=0;

  ② Two ways to include header files:

#include <*system.h*>	// search the default path
#include "*user.h*"	// search the working directory // and then the default path

  The preprocessor syntax differs from that of C/C++.
  The preprocessor syntax is fixed format – a directive must occupy one line unless explicitly continued.

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

- In C89, declarations must be placed before statements.
  Disadvantage:

```
int main(void)
{
   int n,sum,i;
   printf("Enter an integer >= 0: ");
   scanf("%d",&sum);                    // mistake
   sum=0;
   for (i=1;i<=n;i++) sum+=i;
   printf("1+2+...+%d=%d\n",i,sum);  // mistake
}
```

  The scope of variables is too large – the two mistakes cannot be detected by the compiler.

- C99 and C++ support limited scopes.
  Declarations may interlace with statements so that they may be declared as late as possible.
  The control variable of **for** loop may be declared in the initialization part of the loop so as to make it visible only within the loop.

```
int main(void)
{
   printf("Enter an integer >= 0: ");
   int n;
   scanf("%d",&n);            // sum is invisible here
   int sum=0;
   for (int i=1;i<=n;i++) sum+=i;
   printf("1+2+...+%d=%d\n",n,sum);
}                             // i is invisible here
```

A multiple-input C program

- Multiple input data

stdout

```
Enter an integer >= 0: 10
1+2+...+10=55
Enter an integer >= 0: 15
1+2+...+15=120
Enter an integer >= 0: ^Z
```

stdin

This program handles input data one by one until it encounters an end-of-file mark (Ctrl-Z in Windows and Ctrl-D in Unix). N.B. Ctrl-Z or Ctrl-D must be the 1st character of an input line.

- On **`printf`** and **`scanf`**

Calls to **`printf`** and **`scanf`** are expressions with side effects.

`scanf`	Action	the user keys in input
	Value	the number of items that were read successfully or, -1 if an end-of-file mark was encountered

```
int m,n;
scanf("%d%d",&m,&n)
```

Input	value	m	n
5 3↵	2	5	3
5 snoopy↵	1	5	unchanged
snoopy 3↵	0	unchanged	unchanged
^Z↵	-1	unchanged	unchanged

The red-colored character is the next character to read.

`printf`	Action	display a string of characters on the screen
	Value	the number of characters displayed

- On **printf** and **scanf** (Cont'd)

  ```
  printf("%d\n",printf("Snoopy"));  // Snoopy6
  ```

  The value of this call to **printf** is 2, which is discarded.

- ```
  #include <stdio.h>
  int sum(int);
  int main(void)
  {
     printf("Enter an integer >= 0: ");
     int n;
     while (scanf("%d",&n)!=EOF) {       ②
        printf("1+2+...+%d=%d\n",n,sum(n));
        printf("Enter an integer >= 0: ");
     }
  }
  int sum(int n)            ①
  {
     int sum=0;             ①
     for (int i=1;i<=n;i++) sum+=i;
     return sum;
  }
  ```

  ① Within a scope, a name can't be declared more than once, say, as a function and as a variable. The two **sum**'s here have no problem, for they are in distinct scopes.
  The function **sum** is hidden in the inner block – there is a *hole* in the *potential scope* of the function **sum**.

  ② The symbolic constant **EOF** is defined in **<stdio.h>** by the preprocessor command

  ```
  #define EOF (-1)
  ```

  Having seen this definition, the predecessor will replace every occurrence of the token **EOF** that appears later in the file by **(-1)**.

  ```
  scanf("%d",&n)!=EOF  ⇒ scanf("%d",&n)!=(-1)
  int EOF_mark;        ⇒ int EOF_mark;
  ```

  In technical term, **EOF** is a *macro* that expands to (-1).
  ```

- On **`while`** and **`for`** loops

  `while` loop – better for unknown iteration number
  `for` loop – better for known iteration number

  However, they are interchangeable in this example, e.g

```
while (scanf("%d",&n)!=EOF) { … }
≡
for (;scanf("%d",&n)!=EOF;) { … }
```

- Input buffer

```
Enter an integer >= 0: ↵
   8  ↵
1+2+...+8=36
Enter an integer >= 0: 9  10↵
1+2+...+9=45
Enter an integer >= 0: 1+2+...+10=55
Enter an integer >= 0: ^Z
```

`\n   8  \n9  10\n^Z`	Input buffer

  In search of a number, **`scanf`** ignores white-space characters (the space, tab, new line characters).

```
Enter an integer >= 0: 12.34↵
1+2+...+12=78
Enter an integer >= 0: 1+2+...+12=78
Enter an integer >= 0: 1+2+...+12=78
```

  ⋮ infinite loop

`12.34\n`	Input buffer

  To make the program robust, change the test to

```
scanf("%d",&n)==1
```

- On ordinary characters in format strings

  For **`printf`**, they are characters to be outputted exactly.

  For **`scanf`**, they are pattern-matching characters:

  1. White-space characters
     A white-space character in a format string matches a possibly empty sequence of white-space characters in the input
  2. Other characters
     A non-white-space character in a format string matches the same character in the input.

  Example

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

All are distinct.

```
scanf("%d",&n);
scanf(" %d",&n);
scanf("\n  %d",&n);
```

All are the same.

`    \n    12\n`	input buffer

Skipped by **`%d`** in the format string **`"%d"`**
Matched with the space in the format string **`" %d"`**
Matched with **`\n`** in the format string **`"\n  %d"`** (the remaining spaces match nothing)

Example

The following two are different.

```
scanf("NT$%d",&n);        // only A
scanf(" NT$%d",&n);       // both A and B
```

`NT$12\n`	A
`   NT$12\n`	B
`US$12\n`	C

More examples

Example 1 – Fibonacci numbers

$$\begin{aligned} \mathrm{fib}(n) &= n, & n \le 1 \\ &= \mathrm{fib}(n-1) + \mathrm{fib}(n-2), & n > 1 \end{aligned}$$

n	0	1	2	3	4	5	6	7	8	9	10	…
fib(*n*)	0	1	1	2	3	5	8	13	21	34	55	…

↑ ↑ ↑
a *b* *a+b*
↑ ↑
a *b*

Algorithm A – Let the variable **a** hold the answer

```
int fib(int n)
{
    int a=0,b=1;
    for (int i=1;i<=n;i++)
    {
        int c=a; a=b; b=c+b;      // ✗ a=b; b=a+b;    ①
    }
    return a;
}
```

① **{ … }** is called a block statement (or compound statement). Variables declared in a block statement are local to the block.

Alternatively, this line can be written as

```
int c=b; b=a+b; a=c;      // ✗ b=a+b; a=b;
```

or,

```
b=a+b; a=b-a;
```

Similar problem: Swap the values of two variables **x** and **y**.

```
x=y; y=x;                 // ✗
int z=x; x=y; y=z;        // for any type
x=x+y; y=x-y; x=x-y;      // only for arithmetic type
```

Example 1 (Cont'd)

Algorithm B – Let the variable **b** hold the answer when $n \geq 1$

```
int fib(int n)
{
   int a=0,b=1;
   for (int i=1;i<n;i++) {
      int c=b; b=a+b; a=c;
   }
   return n==0? a: b;   ①
}
```

① Conditional expression: *exp*1**?** *exp*2**:** *exp*3
Conditional statement : **if (***exp***)** *stmt*1**;** **else** *stmt*2**;**
if (n==0) return a; else return b;

Example 2 – Integer length (i.e. # of digits)

n	23456	2345	234	23	2	0
r	0	1	2	3	4	5

Bad algorithm

```
int len(int n)
{
   int r=0;
   while (n>0) { n/=10; r++; }        ①
   return r;
}
```

① Bug: If $n = 0$, the **while** loop isn't executed.

Algorithm A

```
int len(int n)
{
   int r=0;
   do { n/=10; r++; } while (n>0);    ①
   return r;
}
```

① No matter what the value of *n* is, **do**...**while** loop is executed at least once.

Example 2 (Cont'd)

Algorithm B

```
int len(int n)
{
   int r=1;
   while (n>=10) {          ①
      n/=10; r++;
   }
   return r;
}
```

① Execute the **while** loop only for multiple-digit numbers

Example 3 – Digit sum

Algorithm A – From LSD to MSD LSD (Least Significant Digit)
MSD (Most Significant Digit)

n	23456	2345	234	23	2	0
sum	0	6	11	15	18	20

```
int sum(int n)
{
   int sum=0;
   while (n>0) {          ①
      sum+=n%10; n/=10;
   }
   return sum;
}
```

① **do**...**while** also works, but is less efficient when $n = 0$.

```
   int sum=0;
   do {
      sum+=n%10; n/=10;
   } while (n>0);
```

Example 3 (Cont'd)

Algorithm B – From MSD to LSD

n	23456				
r	1	10	100	1000	10000
n/r	23456	2345	234	23	2

n	23456	3456	456	56	6	0
r	10000	1000	100	10	1	0
sum	0	2	5	9	14	20

```
int sum(int n)
{
   int r=1;
   while (n/r>=10) r*=10;
   int sum=0;
   while (n>0) {      ①
      sum+=n/r; n%=r; r/=10;
   }
   return sum;
}
```

① Using **r > 0** as the loop termination condition is less efficient when the value of *n* has trailing zeros.

n	23000	3000	0	0	0	0
r	10000	1000	100	10	1	0
sum	0	2	5	5	5	5

Digit sum with equation

Display an equation of summing digits, e.g. 2+3+4+5+6=20

```
int main(void)
{
   printf("%d\n",sum(23456));    ①
}
```

① This call to **sum** outputs 2+3+4+5+6= as a side effect and yields 20 as a value.

Example 3 (Cont'd)

Algorithm A

```
int sum(int n)
{
   int r=1; while (n/r>=10) r*=10;
   int sum=0;
   while (r>0) {      ①
      int d=n/r;      ②
      printf("%d",d);
      if (r>=10) printf("+");  // or, else printf("=");
      sum+=d; n%=r; r/=10;
   }
   printf("=");
   return sum;
}
```

① Using **n > 0** as the loop termination condition is incorrect when the value of *n* has trailing zeros.

② Use a local variable to avoid recomputation.

Algorithm B

```
int sum(int n)
{
   int r=1; while (n/r>=10) r*=10;
   int sum=0;
   while (r>=10) {
      int d=n/r;
      printf("%d+",d);
      sum+=d; n%=r; r/=10;
   }
   sum+=n;            ①
   printf("%d=",n);
   return sum;
}
```

① Handling the LSD here saves the test **r >= 10** within the loop.

Example 3 (Cont'd)

Alternative design – Let the function **sum** do all the jobs

```
void sum(int n)                        ①
{
   // replace the last 2 statements of Algorithm B by the following:
   printf("%d=%d",n,sum);
   return;                             ②
}
int main(void) { sum(23456); }  ③
```

① **void** is a type denoting the empty set.
The two occurrences of **void** have distinct meanings.

```
void sum(int);          // type
int main(void);         // non-type; parameterless
```

② **return;** is redundant here – the function **sum** will automatically return when it falls off the function's body.

It is needed if a function wants to return before falling off body:

```
// speed up for single-digit numbers
void sum(int n)
{
   if (n<10) {
      printf("%d=%d",n,n); return;
   }
   // same code as above
}
```

Remarks

A function whose return type is **void** may or may not contain a **return;** statement.

A function whose return type isn't **void** must contain a **return** *exp*; statement, for some *exp*.

③ A call to **sum** can't be used in a place where a value is required, e.g. **printf("%d\n",sum(23456));**

Example 4 – 9x9 multiplication table; nested loops

```
   |    1    2    3    4    5    6    7    8    9
---|---------------------------------------------
 1 |    1    2    3    4    5    6    7    8    9
 2 |    2    4    6    8   10   12   14   16   18
 3 |    3    6    9   12   15   18   21   24   27
 4 |    4    8   12   16   20   24   28   32   36
 5 |    5   10   15   20   25   30   35   40   45
 6 |    6   12   18   24   30   36   42   48   54
 7 |    7   14   21   28   35   42   49   56   63
 8 |    8   16   24   32   40   48   56   64   72
 9 |    9   18   27   36   45   54   63   72   81
```

```
#include <stdio.h>
void _9x9(void)
{
   printf("   |");
   for (int j=1;j<=9;j++) printf("%5d",j);   // line 1
   printf("\n---|");
   for (int j=1;j<=9;j++) printf("-----");
   printf("\n");                              // line 2
   for (int i=1;i<=9;i++) {
      printf(" %d |",i);
      for (int j=1;j<=9;j++) printf("%5d",i*j);
      printf("\n");
   }
}
int main(void) { _9x9(); }
```

On formatting integers

`%d`	minimum spaces
`%5d`	minimum 5 spaces, right-justified
`%5.2d`	minimum 5 spaces, minimum 2 characters (zero-filled), right-justified
`%05d`	minimum 5 spaces, right-justified, zero-filled
`%-5d`	minimum 5 spaces, left-justified
`%-5.2d`	minimum 5 spaces, minimum 2 characters (zero-filled), left-justified

Example 4 (Continued)

```
printf("|%d|",7);          |7|
printf("|%5d|",7);         |    7|
printf("|%5.2d|",7);       |   07|
printf("|%05d|",7);        |00007|
printf("|%-5d|",7);        |7    |
printf("|%-5.2d|",7);      |07   |
```

On parameterless functions

```
void _9x9();
void _9x9(void);
```

In C++, these two declarations are identical – they both say that the function **_9x9** has no parameters and can only be invoked by **_9x9();**

However, they are different in C.

```
void _9x9();          // unknown parameter list
void _9x9(void);      // parameterless
```

The former can be invoked with any number of arguments of any type, even though the arguments are useless,e.g.

```
_9x9(777,"Bingo");    // OK for the former
_9x9();               // OK for both
```

Example 5 – Factorial; Nested loops

Algorithm A – By multiplication

```
int fact(int n)
{
   int r=1;
   for (int i=2;i<=n;i++) r=r*i;
   return r;
}                              r = r+r+...+r   (i times)
```

Q: How many times has **r=r*i** been executed?

A: $\sum_{i=2}^{n} 1 = n-1$ times

Example 5 (Cont'd)

Algorithm B – By addition

```
int fact(int n)
{
   int r=1;
   for (int i=2;i<=n;i++) {
      int s=0;
      for (int j=1;j<=i;j++) s=s+r;
      r=s;
   }
   return r;                        s = s+1+…+1   (r times)
}
```

Q: How many times has **s=s+r** been executed?

A: $\sum_{i=2}^{n}\sum_{j=1}^{i} 1 = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$ times

Algorithm C – By incrementation

```
int fact(int n)
{
   int r=1;
   for (int i=2;i<=n;i++) {
      int s=0;
      for (int j=1;j<=i;j++)
         for (int k=1;k<=r;k++) s++;
      r=s;
   }
   return r;
}
```

Q: How many times has **s++** been executed?

A: At the beginning of the outermost loop, $r = (i-1)!$. Thus, it is executed $\sum_{i=2}^{n}\sum_{j=1}^{i}\sum_{k=1}^{(i-1)!} 1 = \sum_{i=2}^{n} i!$ times.

Starting with **s=r** and executing the middle loop $i-1$ times give us

$\sum_{i=2}^{n}\sum_{j=1}^{i-1}\sum_{k=1}^{(i-1)!} 1 = \sum_{i=2}^{n}(i-1)(i-1)! = n! - 1$ times.

Appendix A: A sample batch file

```
#include <stdio.h>
int main(void)
{
   int n;
   printf("Enter an integer >= 0: ");
   scanf("%d",&n);
   return n;
}
```

Let **`demo.exe`** be the executable of the above program.
The following DOS batch file invokes **`demo.exe`** and acts on the value returned by **`main`**.

```
echo off
demo
rem The value returned is stored in errorlevel.
rem if (errorlevel>=6) goto err6
if errorlevel 6 goto err6
if errorlevel 3 goto err3
echo 0,1,2
goto exit
:err3
echo 3,4,5
goto exit
:err6
echo 6,7,8,...
:exit
```

Appendix B: GNU Compiler Collection (gcc)

gcc for C programs
g++ for C++ and C programs

Let **`1st.c`** be the file containing our first C program (p.1)

```
% gcc 1st.c
% ./a.out
Enter an integer >= 0: 10
1+2+...+10=55
```

```
% gcc -o 1st 1st.c
% ./1st
Enter an integer >= 0: 12
1+2+...+12=78
```

Let **`2nd.c`** be the file containing our second C program (p.4)

```
% gcc -std=c89 2nd.c
2nd.c: In function 'sum':
2nd.c:15: error: 'for' loop initial declaration used
outside C99 mode
```

```
% gcc -std=c99 2nd.c
Enter an integer >= 0: 10
1+2+...+10=55
Enter an integer >= 0: 12
1+2+...+12=78
Enter an integer >= 0: ^D
```

Both can be compiled by g++.

```
% g++ 1st.c
% g++ 2nd.c
```