Introduction to Scientific and Engineering Computation (BIL 104E)

Lecture 7

Functions and Structural Programming

Functions

A function definition is always given first, before the function is called from a **main()** function.

You can put a function definition anywhere you want, as long as you put the function declaration before the first place where the function is called

- **Function declarations**
- **Prototyping**
- **Values returned from functions**
- **Arguments to functions**
- **Structured programming**

Declaring Functions

You have to declare or define not only a variable but also a function before you can use them.

The **definition** requires the C compiler to reserve storage for a **variable** or **function** named by an identifier.

A **variable** declaration is a definition, but a **function** declaration is not because of these reasons as follows:

❖ **A function declaration** indicates to a function that is defined and specifies what kind of value is returned by the function.

❖ **A function definition** defines what the function does, as well as gives the number and type of arguments passed to the function.

Declaring Functions

If a function definition is placed in your source file before the function is first called, you don't need to make the function declaration.

If a function decleration is placed before the function is first called, you also need function definition but this definition is placed somewhere else.

For example,
Whenever you need the **printf()** function, each time, you have to include a header file, **stdio.h**, because the header file contains the declaration of **printf()**.

This **decleration** indicates to the compiler the return type and prototype of the function.
The **definition** of the printf() function is placed somewhere else

Functions : Specifying Return Types

A function can be declared to return any data type, except an array or function.

The **return** statement used in a function definition returns a single value.

The type of **return** value should match the type declared in the function declaration.

If no explicit data type is specified for the function , the **return** type of a function is **int**, by default.

A data type specifier is placed prior to the name of a function:

data_type_specifier function_name();

Functions : Using Prototypes

The number and types of an argument are called the **function prototype**.

The general form of a function declaration, including its prototype, is as follows:

```
data_type_specifier    function_name (
        data_type_specifier    argument_name1,
        data_type_specifier    argument_name2,
        data_type_specifier    argument_name3,
          .
          .
        data_type_specifier    argument_nameN,
);
```

The purpose of using a function prototype is to help the compiler check whether the data types of arguments passed to a function match what the function expects.

Functions : Making Function Calls

When a function call is made

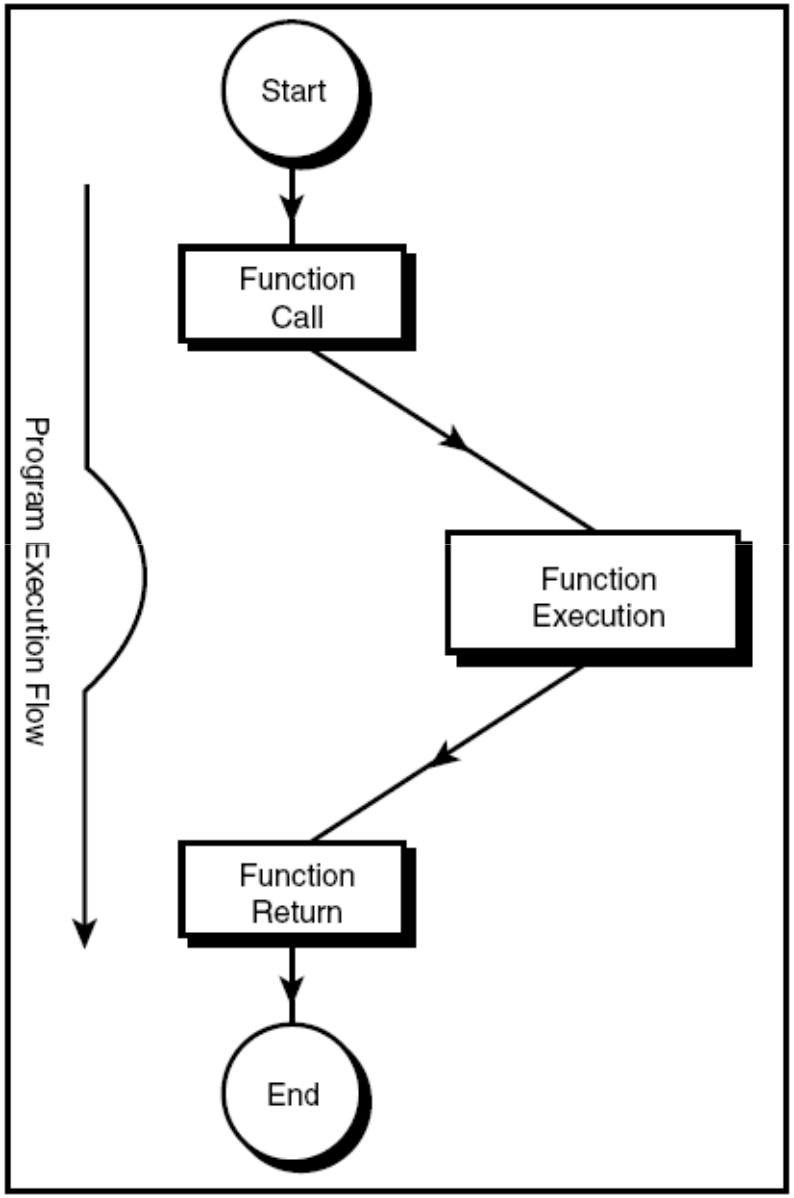
Step1) the program execution jumps to the function

Step2) The task assigned to the function is finished.

Step3) The program execution resumes after the called function returns.

A **function call** is an expression that can be used as a single statement or within other statements.

Functions : Making Function Calls



Functions : Making Function Calls

Calling Functions After They Are Declared and Defined

```
1:  /* 15L01.c: Making function calls */
2:  #include <stdio.h>
3:
4:  int function_1(int x, int y);
5:  double function_2(double x, double y)
6:  {
7:     printf("Within function_2.\n");
8:     return (x - y);
9:  }
10:
11: main()
12: {
13:    int x1 = 80;
14:    int y1 = 10;
15:    double x2 = 100.123456;
16:    double y2 = 10.123456;
17:
18:    printf("Pass function_1  %d and %d.\n", x1, y1);
19:    printf("function_1 returns %d.\n", function_1(x1, y1));
20:    printf("Pass function_2  %f and %f.\n", x2, y2);
21:    printf("function_2 returns %f.\n", function_2(x2, y2));
22:    return 0;
23: }
24: /* function_1() definition */
25: int function_1(int x, int y)
26: {
27:    printf("Within function_1.\n");
28:    return (x + y);
29: }
```

function_1 declaration

function_2 definition

If you make decleration first before making function call, you can make function definition somewhere else.

function_1 definition

Functions : Making Function Calls

Computer Screen

```
Pass function_1  80 and 10.
Within function_1.
function_1 returns 90.
Pass function_2  100.123456. and 10.123456.
Within function_2.
function_2 returns 90.000000.
```

Functions : Prototyping Functions

Three types of way are possible for arguments passed to functions.

- **functions take no argument**
- **functions take a fixed number of arguments**
- **functions take a variable number of arguments**

Functions with No Arguments

For instance, the C library function **getchar()** does not need any arguments. It can be used in a program like this:

```
int c;
c = getchar();
```

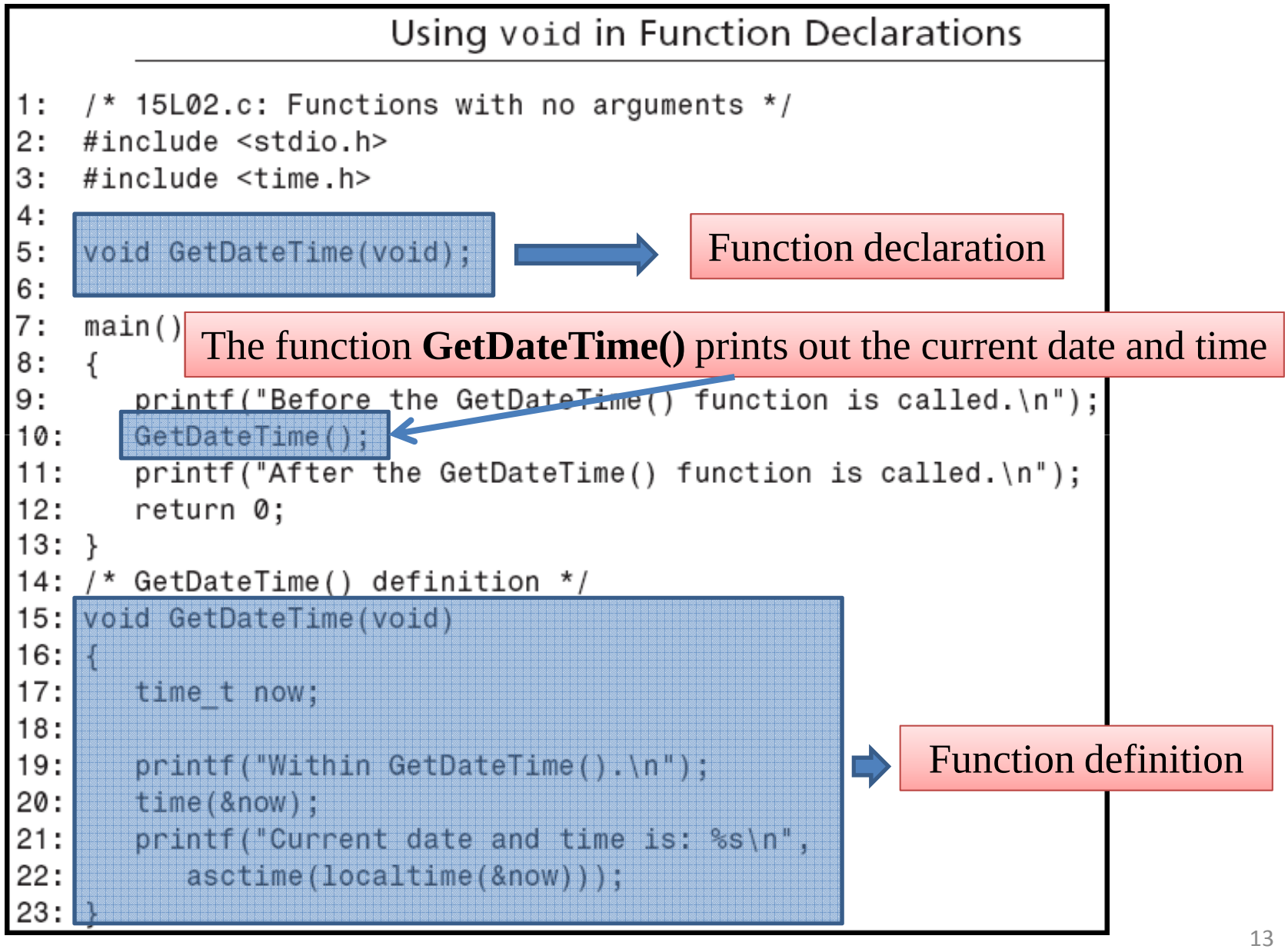
In C, the declaration of the **getchar()** function can be something like this:

```
int  getchar( void );
```

The keyword **void** is used in the declaration to indicate that no argument is needed by this function.

When **getchar()** is called if there is any argument passed to getchar(), the compiler will generate an error message.

Functions with No Arguments



Functions with No Arguments

Computer Screen

```
Before the GetDateTime() function is called.
Within GetDateTime().
Current date and time is: Sat Apr 05 11:50:10 1997
```

Functions: time(), localtime(), and asctime()

The **declarations** of all **date and time functions** are included in the header file **time.h**.

These functions can give three types of date and time:

❖**Calendar time:** gives the current date and time based on the Gregorian calendar (solar calendar).

❖**Local time:** represents the calendar time in a specific time zone

❖ **Daylight savings time:** is the local time under the daylight savings rule

Functions : time()

The **time()** function returns the calendar time.

The syntax for the **time()** function is

```
#include <time.h>
time_t    time(time_t  *timer);
```

time_t is the arithmetic type that is used to represent time.

timer is a pointer variable pointing to a memory storage that can hold the calendar time returned by this function.

The **time()** function returns **-1** if the calendar time is not available on the computer.

Functions : localtime()

The **localtime()** function returns the local time converted from the calendar time.

The syntax for the localtime() function is

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

tm is a structure that contains the components of the calendar time. **struct** is the keyword for structure, which is another data type in C.

timer is a pointer variable pointing to a memory storage that holds the calendar time returned by the **time()** function.

Functions : asctime()

To convert the date and time represented by the structure **tm**, **asctime()** function can be called.

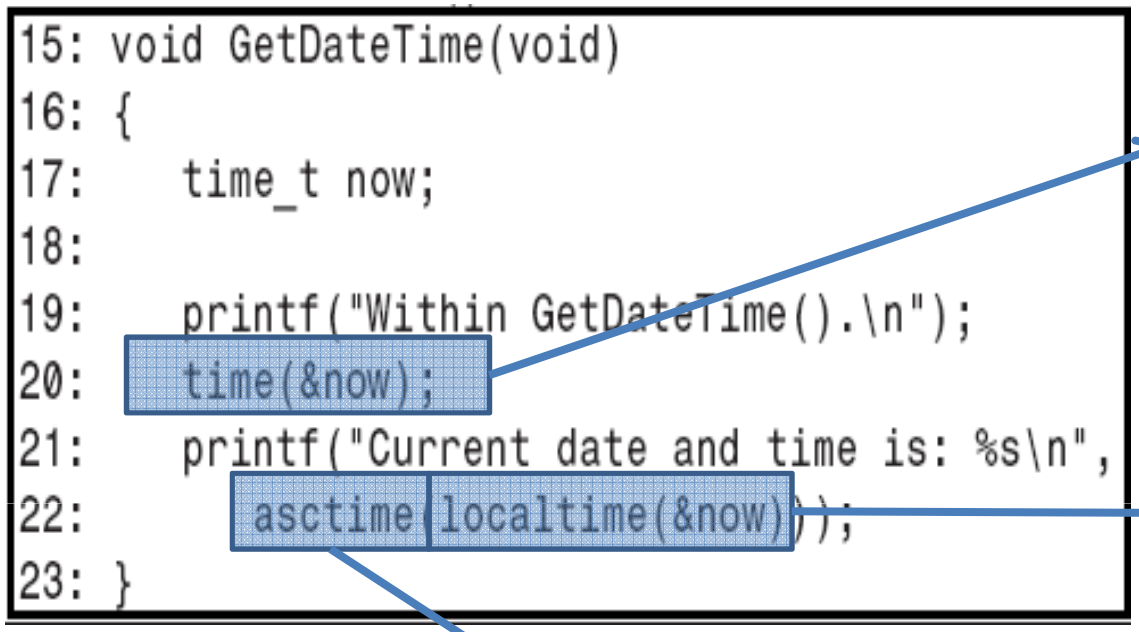
The syntax for the asctime() function is

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

timeptr is a pointer referencing the structure **tm** returned by date and time functions like **localtime()**.

The **asctime()** function converts the date and time represented by **tm** into a character string.

Functions : GetDateTime()



Line 20 stores the calendar time in the memory location referenced by the **now** variable.

The local time expression of the calendar time is obtained by **localtime()**

The character string representing the date and time is obtained with help from **asctime()**

Functions with a Fixed Number of Arguments

To declare a function with a fixed number of arguments, you need to specify the **data type** and **name** of each argument.

Because the compiler can check whether the function declaration match with the implementation in the function definition.

For instance:

int function_1(int x, int y);

This **declaration** contains the prototype of two arguments, x and y.

Prototyping a Variable Number of Arguments

The following is a general form to declare a function with a variable number of arguments:

```
data_type_specifier   function_name(
        data_type_specifier   argument_name1, ...
);
```

The first argument name is followed by the **… (three dot)** that represents the rest of unspecified arguments.

For instance, the syntax of the printf() function is,

```
int   printf(const  char  *format[  , argument, ...]);
```

The brackets ([and]) indicate that the unspecified arguments are optional.

Processing Variable Arguments

There are three routines declared in the header file **stdarg.h** that enable you to write functions that take a variable number of arguments.

va_start(), va_arg(), and va_end().

Also **stdarg.h** includes data type, **va_list**, which defines an array type suitable for containing data items

The syntax for the **va_start()** macro is

```
#include <stdarg.h>
void   va_start(va_list ap, lastfix);
```

ap is the name of the array that is about to be initialized by the **va_start()** macro routine.

lastfix should be the argument before the (...) in the function declaration.

Processing Variable Arguments

the **va_arg()** macro can be used to get the next argument passed to the function.

The syntax for the **va_arg()** macro is

```
#include <stdarg.h>
type   va_arg(va_list ap,    data_type);
```

data_type is the data type of the argument passed to function.

To facilitate a normal return from your function, you have to use the **va_end()** function in your program after all arguments have been processed.

The syntax for the va_end() function is

```
#include <stdarg.h>
void va_end(va_list ap);
```

Processing Variable Arguments

Processing Variable Arguments

```
1:  /* 15L03.c: Processing variable arguments */
2:  #include <stdio.h>
3:  #include <stdarg.h>
4:
5:  double AddDouble(int x, ...);
6:
7:  main ()
8:  {
9:     double d1 = 1.5;
10:    double d2 = 2.5;
11:    double d3 = 3.5;
12:    double d4 = 4.5;
13:
14:    printf("Given an argument: %2.1f\n", d1);
15:    printf("The result returned by AddDouble() is: %2.1f\n\n",
16:       AddDouble(1, d1));
17:    printf("Given arguments: %2.1f and %2.1f\n", d1, d2);
18:    printf("The result returned by AddDouble() is: %2.1f\n\n",
19:       AddDouble(2, d1, d2));
20:    printf("Given arguments: %2.1f, %2.1f and %2.1f\n", d1, d2, d3);
21:    printf("The result returned by AddDouble() is: %2.1f\n\n",
22:       AddDouble(3, d1, d2, d3));
23:    printf("Given arguments: %2.1f, %2.1f, %2.1f, and %2.1f\n",
24:           d1, d2, d3, d4);
25:    printf("The result returned by AddDouble() is: %2.1f\n",
```

Processing Variable Arguments

```
26:        AddDouble(4, d1, d2, d3, d4));
27:    return 0;
28: }
29: /* definition of AddDouble() */
30: double AddDouble(int x, ...)
31: {
32:    va_list   arglist;
33:    int i;
34:    double result = 0.0;
35:
36:    printf("The number of arguments is: %d\n", x);
37:    va_start (arglist, x);
38:    for (i=0; i<x; i++)
39:       result += va_arg(arglist, double);
40:    va_end (arglist);
41:    return result;
42: }
```

Processing Variable Arguments

Computer Screen

```
Given an argument: 1.5
The number of arguments is: 1
The result returned by AddDouble() is: 1.5

Given arguments: 1.5 and 2.5
The number of arguments is: 2
The result returned by AddDouble() is: 4.0

Given arguments: 1.5, 2.5, and 3.5
The number of arguments is: 3
The result returned by AddDouble() is: 7.5

Given arguments: 1.5, 2.5, 3.5, and 4.5
The number of arguments is: 4
The result returned by AddDouble() is: 12.0
```

Structured Programming

Structured programming is one of the best programming methodologies. Basically, there are two types of structured programming:

- ❖**top-down programming**
- ❖**bottom-up programming.**

top-down programming :

To solve a problem, you can first work out an outline and start your programming at a higher level.

For instance, you can work on the main() function at the beginning, and then move to the next lower level until the lowest-level functions are written.

Structured Programming

bottom-up programming :
To solve a problem, you can work on the smallest pieces of the problem. First, you define and write functions for each piece. After each function is written and tested, you begin to put them together to build a program that can solve the problem.

it’s useful to combine these two types of structured programming and use them alternately in order to solve real problems.