FUNCTIONS

Lecture # 9

VOID FUNCTIONS

- Void functions and value-returning functions have similar structures
  - Both have a heading part and a statement part
- User-defined void functions can be placed either before or after the function `main`
- If user-defined void functions are placed after the function `main`
  - The function prototype must be placed before the function `main`

VOID FUNCTIONS (CONTINUED)

- A void function does not have a return type
  - `return` statement without any value is typically used to exit the function early
- Formal parameters are optional
- A call to a void function is a stand-alone statement

VOID FUNCTIONS WITHOUT PARAMETERS

- Function definition syntax:

```
void functionName()
{
     statements
}
```

- `void` is a reserved word
- Function call syntax:

```
functionName();
```

VOID FUNCTIONS WITH PARAMETERS

- Function definition syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

- Formal parameter list syntax:

```
dataType& variable, dataType& variable, ...
```

- Function call syntax:

```
functionName(actual parameter list);
```

- Actual parameter list syntax:

```
expression or variable,expression or variable, ...
```

VOID FUNCTIONS WITH PARAMETERS (CONTINUED)

- Value parameter: a formal parameter that receives a copy of the content of corresponding actual parameter
- Reference parameter: a formal parameter that receives the location (memory address) of the corresponding actual parameter

VALUE PARAMETERS

- If a formal parameter is a value parameter
  - The value of the corresponding actual parameter is copied into it
- The value parameter has its own copy of the data
- During program execution
  - The value parameter manipulates the data stored in its own memory space

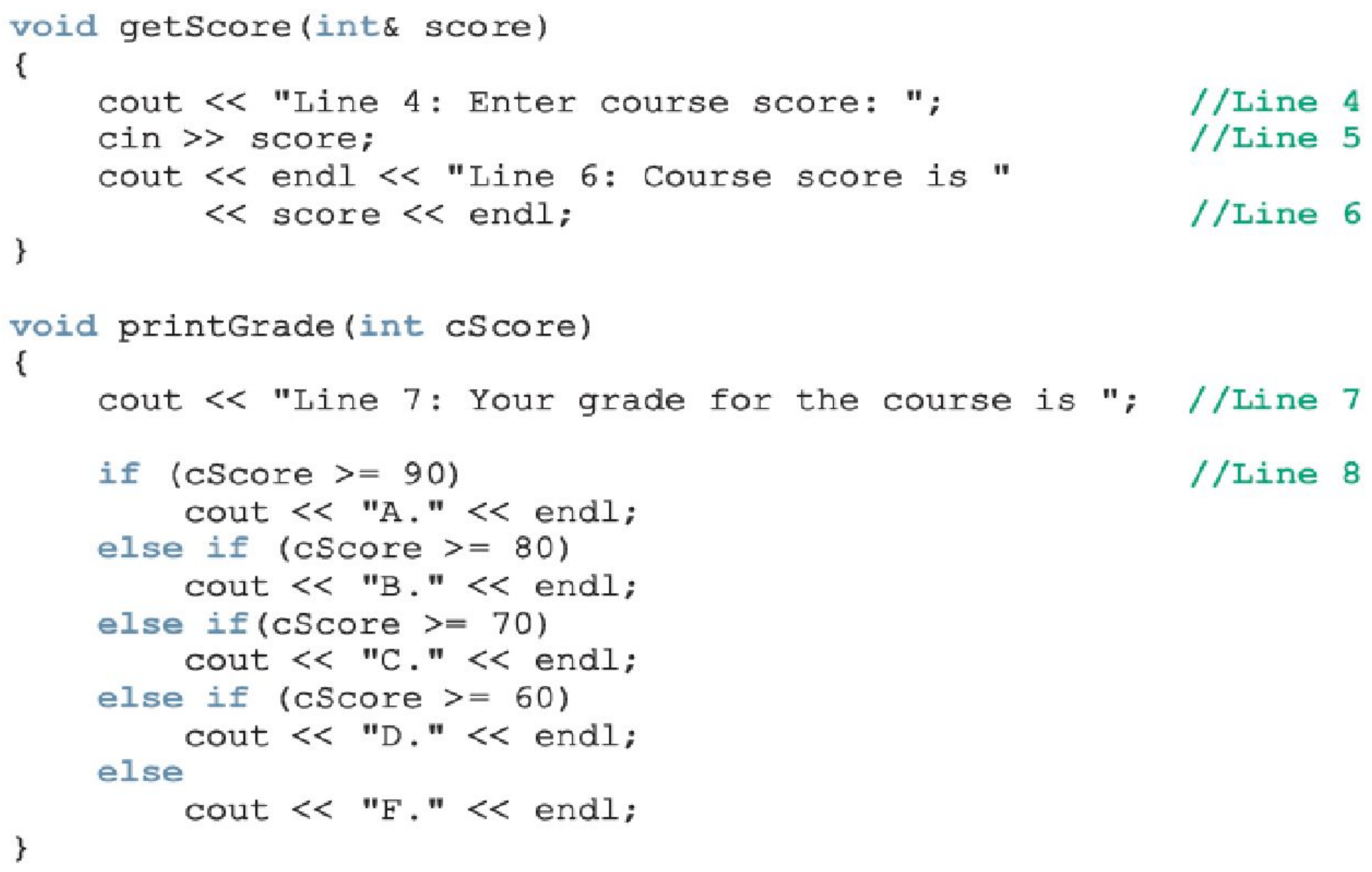
REFERENCE VARIABLES AS PARAMETERS

- If a formal parameter is a reference parameter
  - It receives the memory address of the corresponding actual parameter
- A reference parameter stores the address of the corresponding actual parameter
- During program execution to manipulate data
  - The address stored in the reference parameter directs it to the memory space of the corresponding actual parameter

REFERENCE VARIABLES AS PARAMETERS (CONTINUED)

- Reference parameters can:
  - Pass one or more values from a function
  - Change the value of the actual parameter
- Reference parameters are useful in three situations:
  - Returning more than one value
  - Changing the actual parameter
  - When passing the address would save memory space and time

CALCULATE GRADE (CONTINUED)



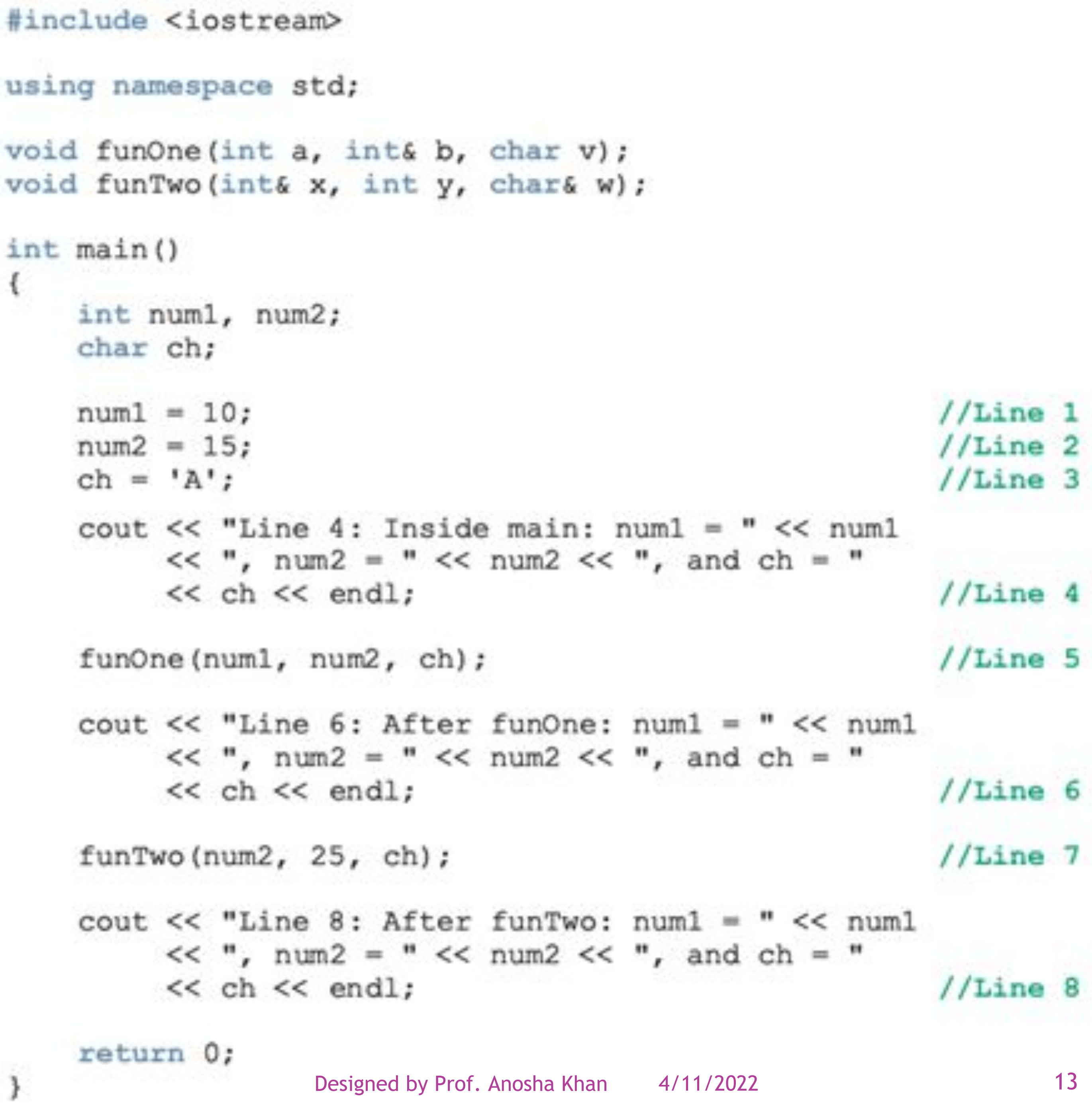
```
void getScore(int& score)
{
    cout << "Line 4: Enter course score: ";            //Line 4
    cin >> score;                                        //Line 5
    cout << endl << "Line 6: Course score is "
         << score << endl;                               //Line 6
}

void printGrade(int cScore)
{
    cout << "Line 7: Your grade for the course is ";   //Line 7

    if (cScore >= 90)                                    //Line 8
        cout << "A." << endl;
    else if (cScore >= 80)
        cout << "B." << endl;
    else if(cScore >= 70)
        cout << "C." << endl;
    else if (cScore >= 60)
        cout << "D." << endl;
    else
        cout << "F." << endl;
}
```

VALUE AND REFERENCE PARAMETERS AND MEMORY ALLOCATION

- When a function is called
  - Memory for its formal parameters and variables declared in the body of the function (called local variables) is allocated in the function data area
- In the case of a value parameter
  - The value of the actual parameter is copied into the memory cell of its corresponding formal parameter

VALUE AND REFERENCE PARAMETERS AND MEMORY ALLOCATION (CONTINUED)

- In the case of a reference parameter
  - The address of the actual parameter passes to the formal parameter
- Content of formal parameter is an address
- During execution, changes made by the formal parameter permanently change the value of the actual parameter



```cpp
#include <iostream>

using namespace std;

void funOne(int a, int& b, char v);
void funTwo(int& x, int y, char& w);

int main()
{
    int num1, num2;
    char ch;

    num1 = 10;                                                  //Line 1
    num2 = 15;                                                  //Line 2
    ch = 'A';                                                   //Line 3

    cout << "Line 4: Inside main: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                         //Line 4

    funOne(num1, num2, ch);                                     //Line 5

    cout << "Line 6: After funOne: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                         //Line 6

    funTwo(num2, 25, ch);                                       //Line 7

    cout << "Line 8: After funTwo: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                         //Line 8

    return 0;
}
```

```
void funOne(int a, int& b, char v)
{
    int one;

    one = a;                                        //Line 9
    a++;                                            //Line 10
    b = b * 2;                                      //Line 11
    v = 'B';                                        //Line 12

    cout << "Line 13: Inside funOne: a = " << a
         << ", b = " << b << ", v = " << v
         << ", and one = " << one << endl;          //Line 13
}

void funTwo(int& x, int y, char& w)
{
    x++;                                            //Line 14
    y = y * 2;                                      //Line 15
    w = 'G';                                        //Line 16

    cout << "Line 17: Inside funTwo: x = " << x
         << ", y = " << y << ", and w = " << w
         << endl;                                   //Line 17
}
```

Sample Run:

```
Line 4: Inside main: num1 = 10, num2 = 15, and ch = A
Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10
Line 6: After funOne: num1 = 10, num2 = 30, and ch = A
Line 17: Inside funTwo: x = 31, y = 50, and w = G
Line 8: After funTwo: num1 = 10, num2 = 31, and ch = G
```

REFERENCE PARAMETERS AND VALUE-RETURNING FUNCTIONS

- You can also use reference parameters in a value-returning function
  - Not recommended
- By definition, a value-returning function returns a single value
  - This value is returned via the return statement
- If a function needs to return more than one value, you should change it to a void function and use the appropriate reference parameters to return the values

SCOPE OF AN IDENTIFIER

- The scope of an identifier refers to where in the program an identifier is accessible
- Local identifier: identifiers declared within a function (or block)
- Global identifier: identifiers declared outside of every function definition
- C++ does not allow nested functions
  - The definition of one function cannot be included in the body of another function

In general, the following rules apply when an identifier is accessed:

1. Global identifiers (such as variables) are accessible by a function or a block if:
    a. The identifier is declared before the function definition (block),
    b. The function name is different from the identifier,
    c. All parameters of the function have names different than the name of the identifier, and
    d. All local identifiers (such as local variables) have names different than the name of the identifier.
2. **(Nested Block)** An identifier declared within a block is accessible:
    a. Only within the block from the point at which it is declared until the end of the block, and
    b. By those blocks that are nested within that block if the nested block does not have an identifier with the same name as that of the outside block (the block that encloses the nested block).
3. The scope of a function name is similar to the scope of an identifier declared outside any block. That is, the scope of a function name is the same as the scope of a global variable.

SCOPE OF AN IDENTIFIER (CONTINUED)

- Some compilers initialize global variables to default values
- The operator `::` is called the scope resolution operator
- By using the scope resolution operator
  - A global variable declared before the definition of a function (block) can be accessed by the function (or block) even if the function (or block) has an identifier with the same name as the variable
- C++ provides a way to access a global variable declared after the definition of a function
  - In this case, the function must not contain any identifier with the same name as the global variable

SAMPLE PROGRAM

```
#include<iostream>
Using namespace std;
Int number = 7;
Int main()
{
Double number=10.5;
Cout<<“local double value of number
   =”<<number<<“\nglobal int value of number”
<<::number<<endl;
}
```

GLOBAL VARIABLES, NAMED CONSTANTS, AND SIDE EFFECTS

- Using global variables has side effects
- A function that uses global variables is not independent
- If more than one function uses the same global variable and something goes wrong
  - It is difficult to find what went wrong and where
  - Problems caused in one area of the program may appear to be from another area
- Global named constants have no side effects