Programming Language 1 (MT261)

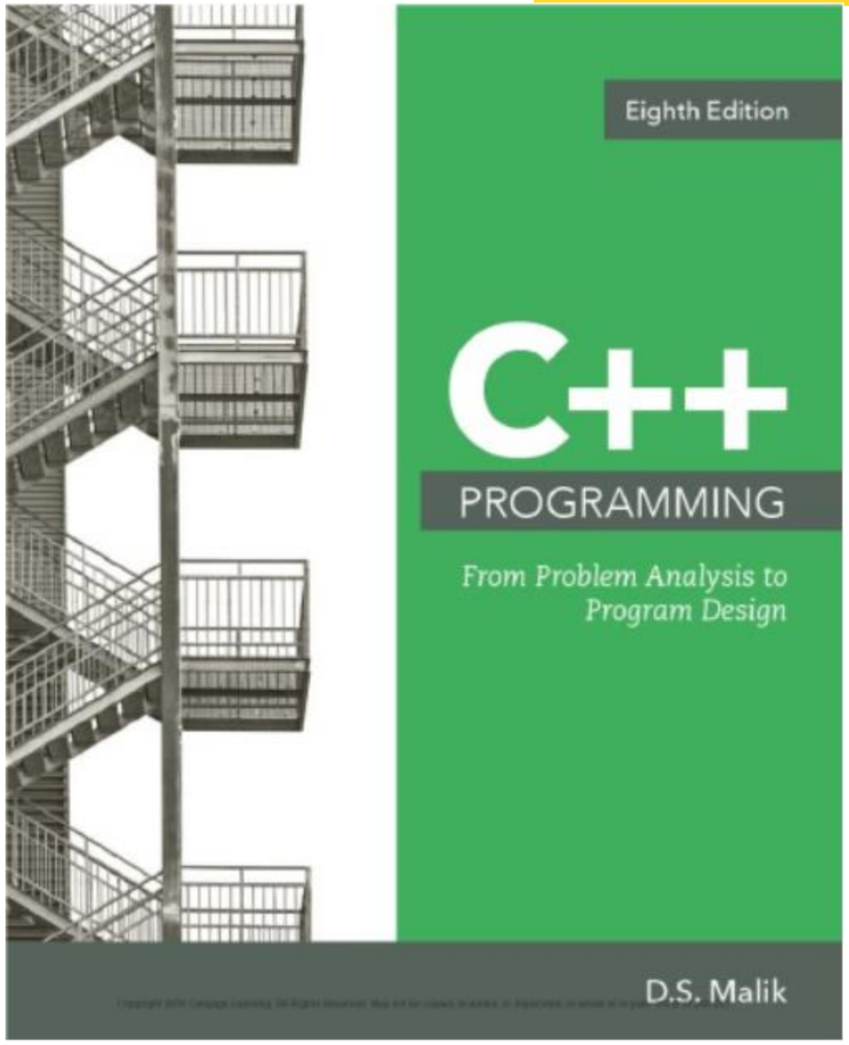
Lecture 4

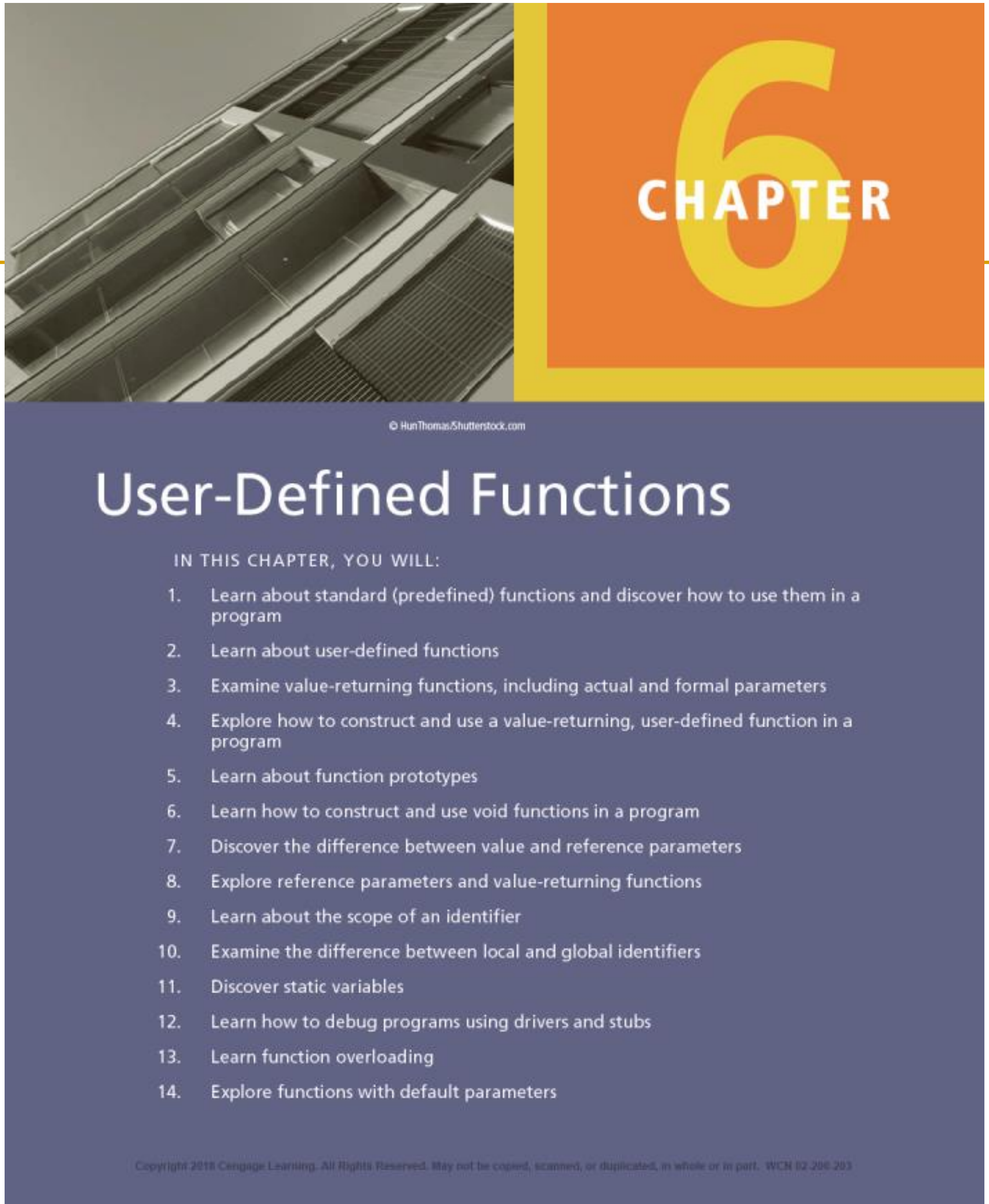
Dr. Ahmed Fathalla

Resources and References

Book:

"C++ Programming: From Problem Analysis to Program Design"





CHAPTER 6

© HunThomas/Shutterstock.com

User-Defined Functions

IN THIS CHAPTER, YOU WILL:

1. Learn about standard (predefined) functions and discover how to use them in a program
2. Learn about user-defined functions
3. Examine value-returning functions, including actual and formal parameters
4. Explore how to construct and use a value-returning, user-defined function in a program
5. Learn about function prototypes
6. Learn how to construct and use void functions in a program
7. Discover the difference between value and reference parameters
8. Explore reference parameters and value-returning functions
9. Learn about the scope of an identifier
10. Examine the difference between local and global identifiers
11. Discover static variables
12. Learn how to debug programs using drivers and stubs
13. Learn function overloading
14. Explore functions with default parameters

Content

Exercise

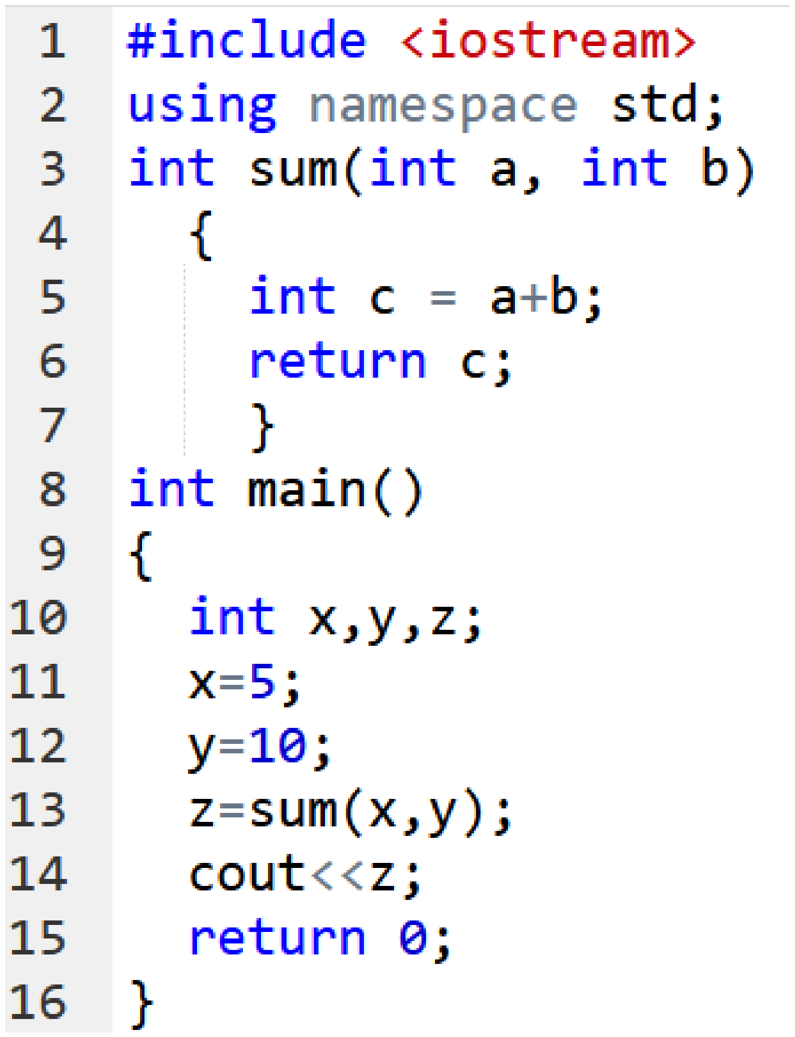
What is the returning value when score equals to 60?

```
char courseGrade(int score)
{
    switch (score / 10)
    {
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        return 'F';
    case 6:
        return 'D';
    case 7:
        return 'C';
    case 8:
        return 'B';
    case 9:
    case 10:
        return 'A';
    }
}
```

Exercise

Write a function that returns the sum of two input parameters.

Scope of an Identifier

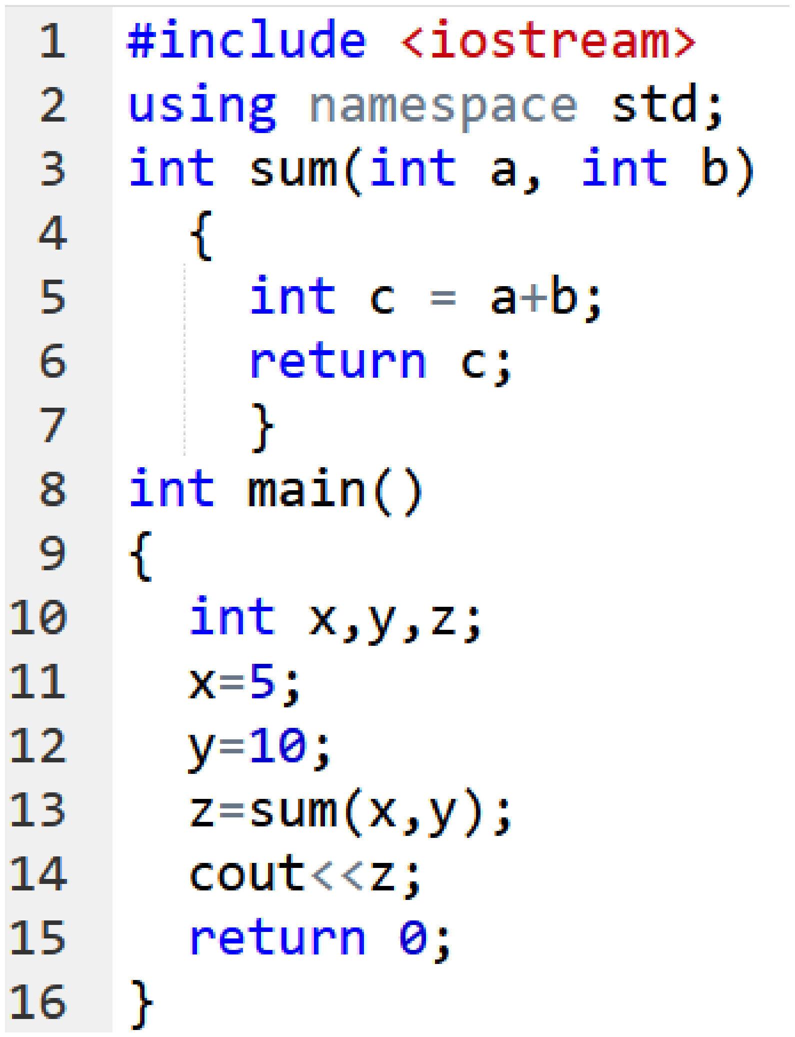


```
#include <iostream>
using namespace std;
int sum(int a, int b)
   {
      int c = a+b;
      return c;
      }
int main()
{
   int x,y,z;
   x=5;
   y=10;
   z=sum(x,y);
   cout<<z;
   return 0;
}
```

Scope of an Identifier

- Any identifier used in a C++ program (such as the name of a variable or object, the name of a type or class, or the name of a named constant) has a ***scope***, i.e., <u>a region of the program in which that name can be used</u>.
- **<u>Local identifier</u>**: identifiers declared within a function (or block)
- **<u>Global identifier</u>**: identifiers declared outside of every function definition

Scope of an Identifier

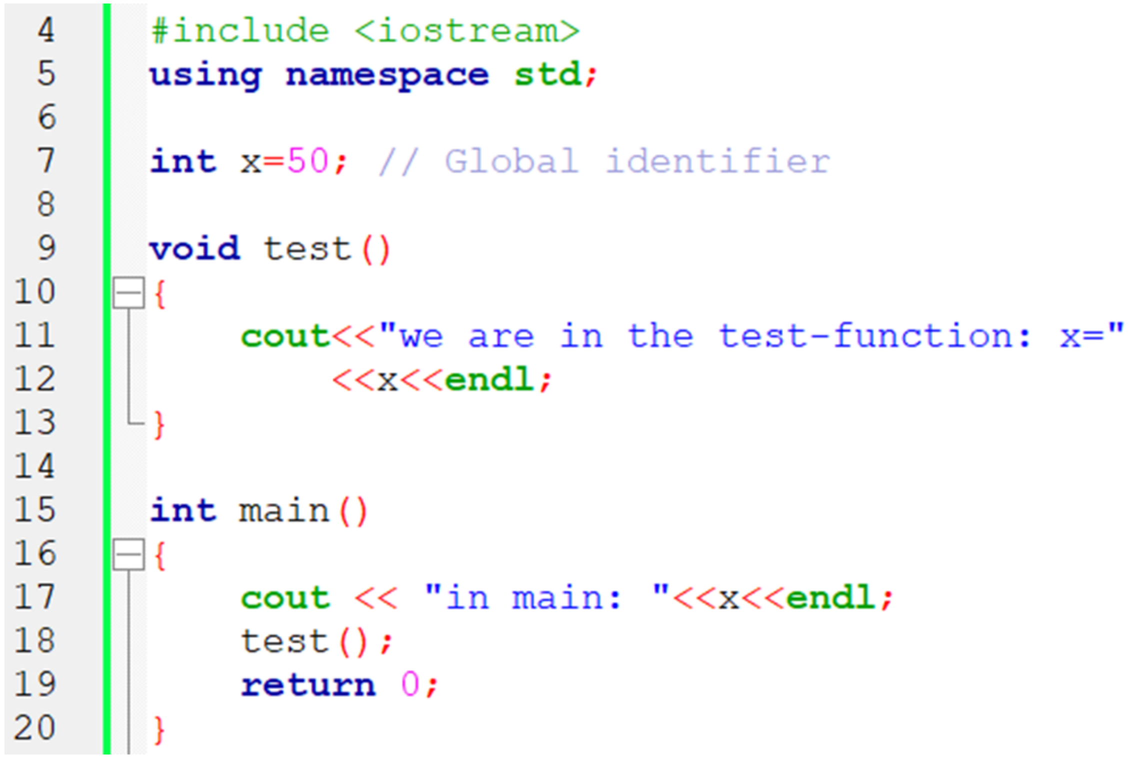


```
#include <iostream>
using namespace std;
int sum(int a, int b)
   {
      int c = a+b;
      return c;
      }
int main()
{
   int x,y,z;
   x=5;
   y=10;
   z=sum(x,y);
   cout<<z;
   return 0;
}
```

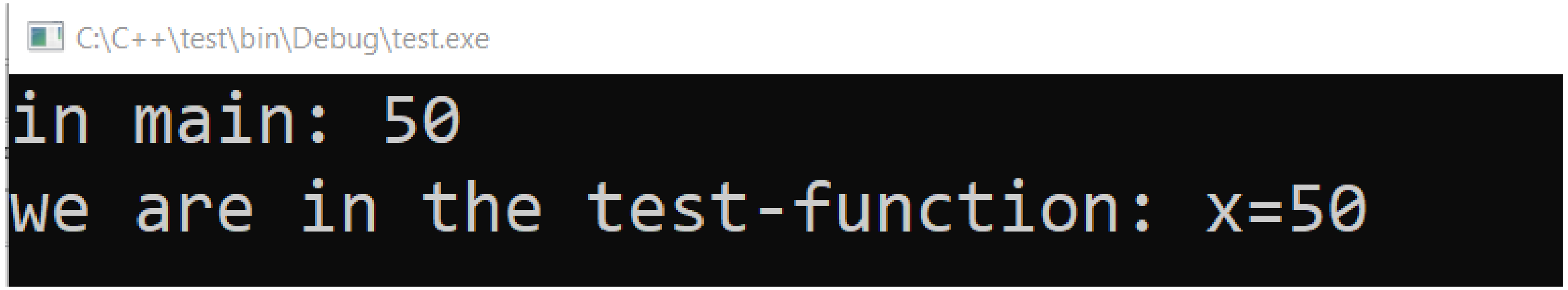
a, b, and c are local variables

x, y, and z are local variables

Global Identifier



```
#include <iostream>
using namespace std;

int x=50; // Global identifier

void test()
{
    cout<<"we are in the test-function: x="
        <<x<<endl;
}

int main()
{
    cout << "in main: "<<x<<endl;
    test();
    return 0;
}
```



C:\C++\test\bin\Debug\test.exe

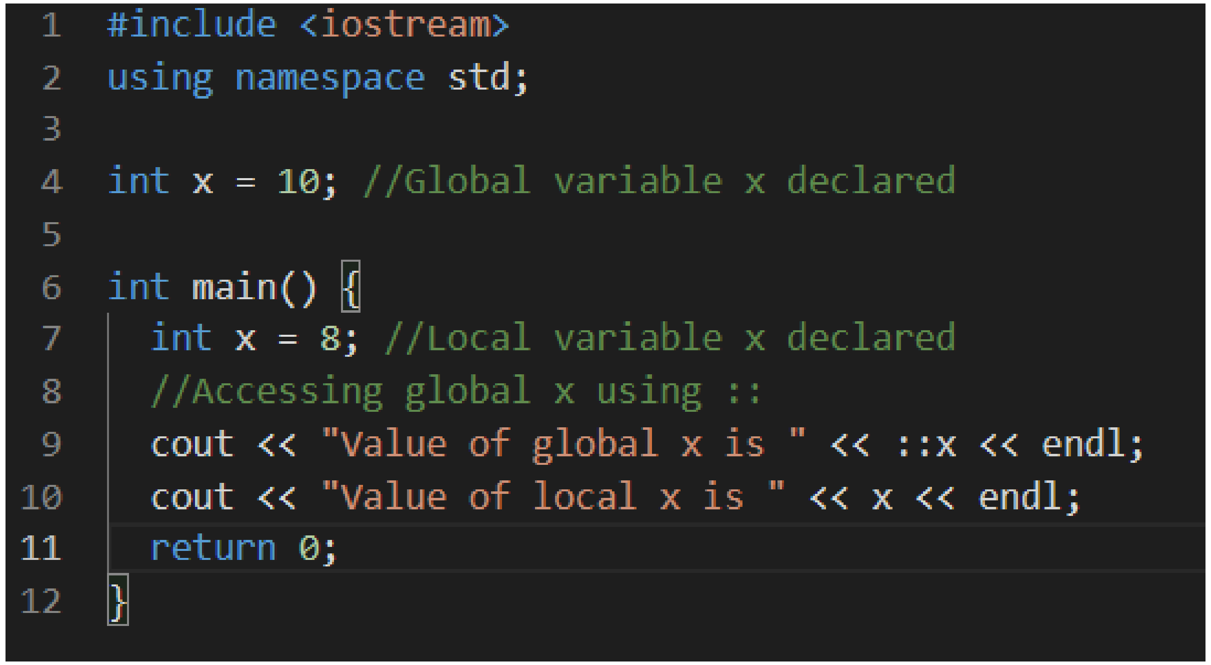
```
in main: 50
we are in the test-function: x=50
```

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.

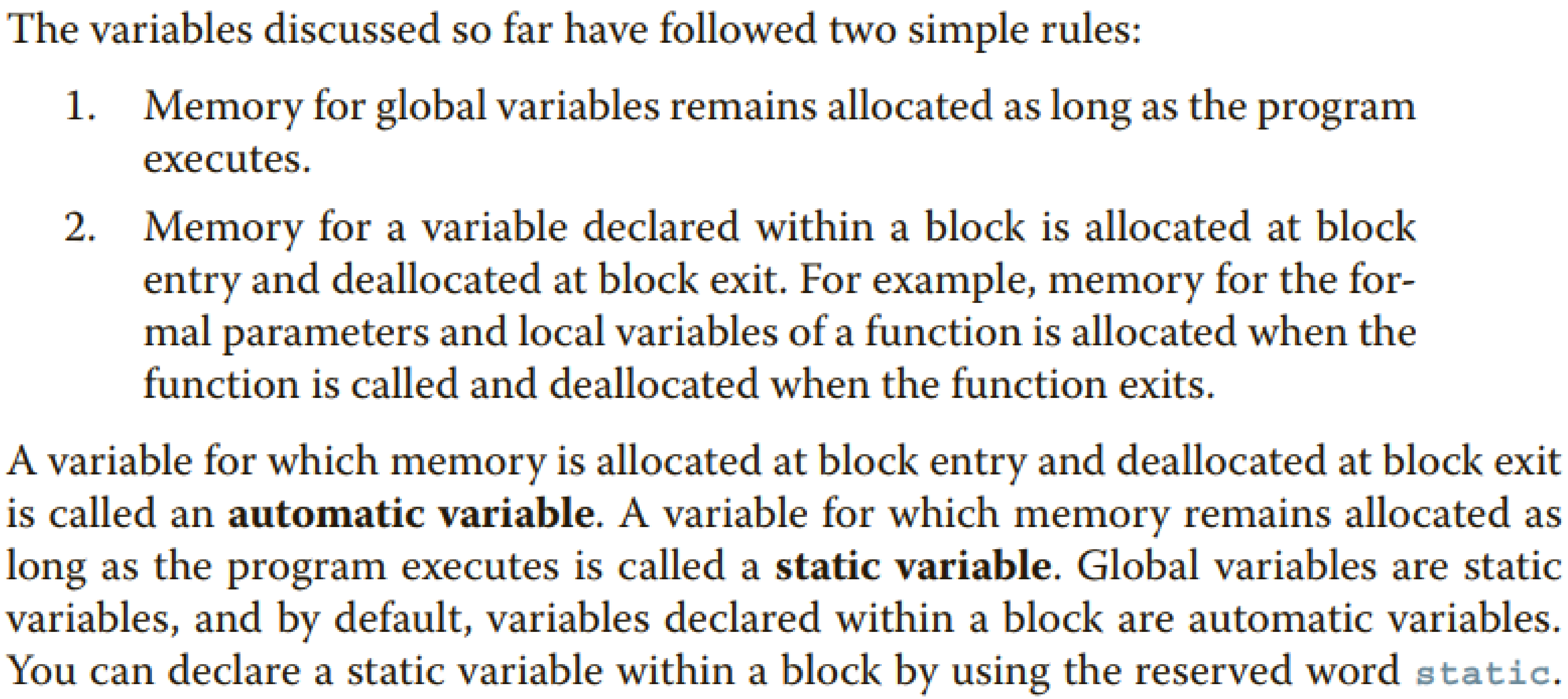
Scope resolution operator ::

- The **scope resolution operator (::)** specifies the scope of a referenced identifier.
- Distinguishing a global variable from a local variable of the same name.



```cpp
#include <iostream>
using namespace std;

int x = 10; //Global variable x declared

int main() {
  int x = 8; //Local variable x declared
  //Accessing global x using ::
  cout << "Value of global x is " << ::x << endl;
  cout << "Value of local x is " << x << endl;
  return 0;
}
```

Static and Automatic Variables



The variables discussed so far have followed two simple rules:

1. Memory for global variables remains allocated as long as the program executes.
2. Memory for a variable declared within a block is allocated at block entry and deallocated at block exit. For example, memory for the formal parameters and local variables of a function is allocated when the function is called and deallocated when the function exits.

A variable for which memory is allocated at block entry and deallocated at block exit is called an **automatic variable**. A variable for which memory remains allocated as long as the program executes is called a **static variable**. Global variables are static variables, and by default, variables declared within a block are automatic variables. You can declare a static variable within a block by using the reserved word `static`.

Task: Check the following C++ code using Online_compiler

```
#include <iostream>
using namespace std;
void fun(void)
{
        int a=0;
        static int b=0;
        cout<<"a ="<<a<<
                " b="<<b<<endl;
        a++;
        b++;
}
```

```
int main()
{
        int loop;
        //calling function 10 times
        for(loop=0; loop<5; loop++)
                fun();
        return 0;
}
// further_reading
```

Functions with Default Parameters

Functions with Default Parameters

- In C++ programming, we can provide default values for function parameters.
- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn’t provide a value for the argument with a default value.

Functions with Default Parameters

```
int volume(int l=1, int w=1, int h=1)
{
    return l * w * h;
}
int main()
{
    cout << volume() << endl;
    cout << volume(10) << endl;
    cout << volume(10, 15) << endl;
    cout << volume(10, 15, 25) << endl;
    return 0;
}
```

Functions with Default Parameters

This section discusses functions with default parameters. Recall that when a function is called, the number of actual and formal parameters must be the same. C++ relaxes this condition for functions with default parameters. You specify the value of a default parameter when the function name appears for the first time, usually in the prototype. In general, the following rules apply for functions with default parameters:

- If you do not specify the value of a default parameter, the default value is used for that parameter.
- All of the default parameters must be the far-right parameters of the function.
- Suppose a function has more than one default parameter. In a function call, if a value to a default parameter is not specified, then you must omit all of the arguments to its right.
- Default values can be constants, global variables, or function calls.
- The caller has the option of specifying a value other than the default for any default parameter.
- You cannot assign a constant value as a default value to a reference parameter.

Value and Reference Parameters

Reading task: Chapter_6 page 390:399

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. Recall that in the case of a value parameter, the value of the actual parameter is copied into the memory cell of its corresponding formal parameter. In the case of a reference parameter, the address of the actual parameter passes to the formal parameter. That is, the content of the formal parameter is an address. During data manipulation, the address stored in the formal parameter directs the computer to manipulate the data of the memory cell at that address. Thus, in the case of a reference parameter, both the actual and formal parameters refer to the same memory location. Consequently, during program execution, changes made by the formal parameter permanently change the value of the actual parameter.

Value and Reference Parameters

```
#include <iostream>
using namespace std;

void increment(int a)
{
    a++;
}
```

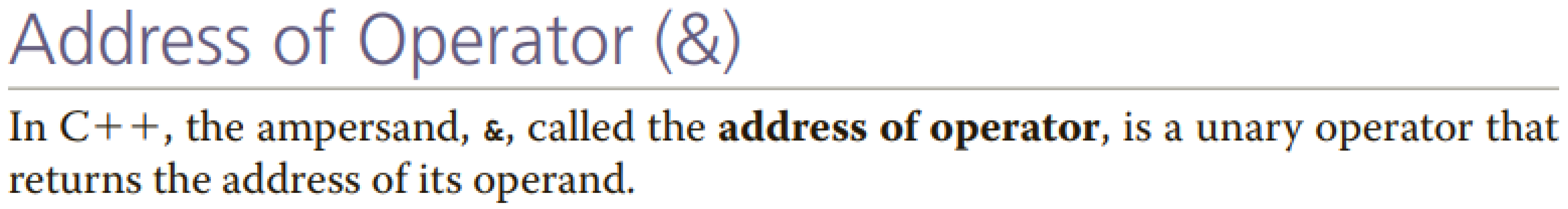
```
int main()
{
    int x = 5;
    cout<<x;
    increment(x);
    cout<<x;
}
```

https://pythontutor.com/visualize.html#mode=edit

Value and Reference Parameters

- **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.
  - The "**address of**" operator (ampersand) is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.



Address of Operator (&)

In C++, the ampersand, &, called the **address of operator**, is a unary operator that returns the address of its operand.

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.

Reference Parameters and Value-Returning Functions

Earlier in this chapter, in the discussion of value-returning functions, you learned how to use value parameters only. You can also use reference parameters in a value-returning function, although this approach is 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, as a rule of good programming style, you should change it to a void function and use the appropriate reference parameters to return the values.

Calculate Grade

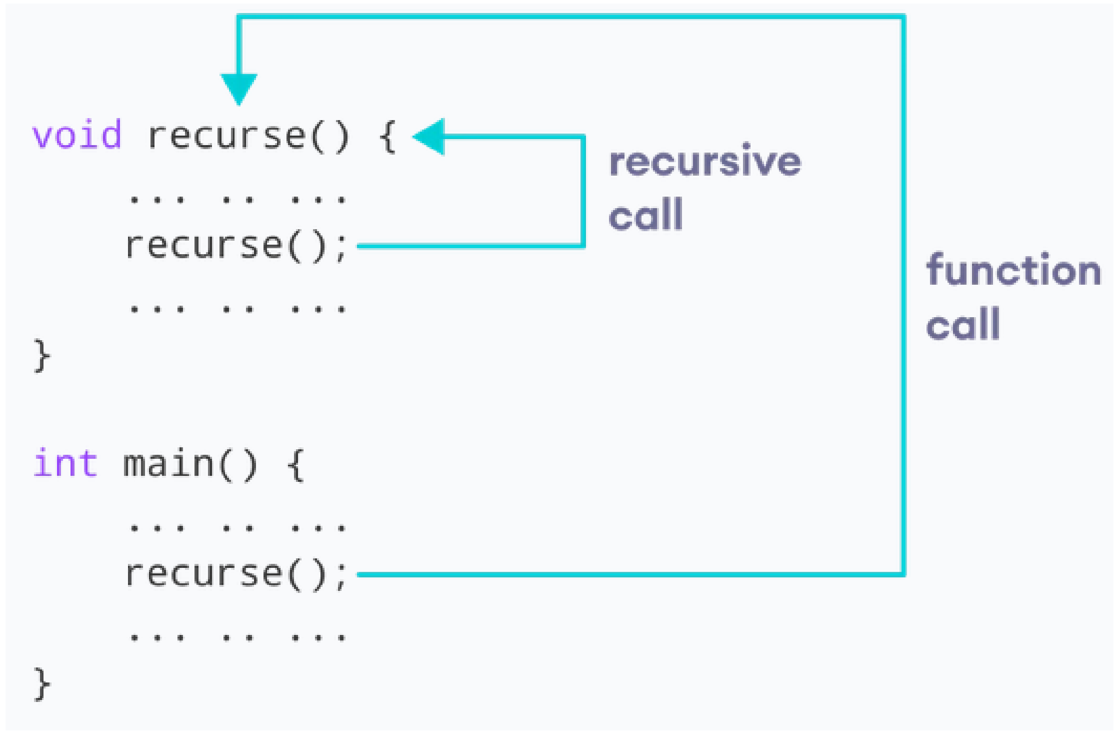
```
//This program reads a course score and prints the
//associated course grade.

#include <iostream>
using namespace std;

void getScore(int& score);
void printGrade(int score);

int main()
{
    int courseScore;

    cout << "Line 1: Based on the course score, \n"
         << "    this program computes the "
         << "course grade." << endl;

    getScore(courseScore);

    printGrade(courseScore);

    return 0;
}
```

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

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

    if (cScore >= 90)
        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;
}
```

Recursion

Recursion

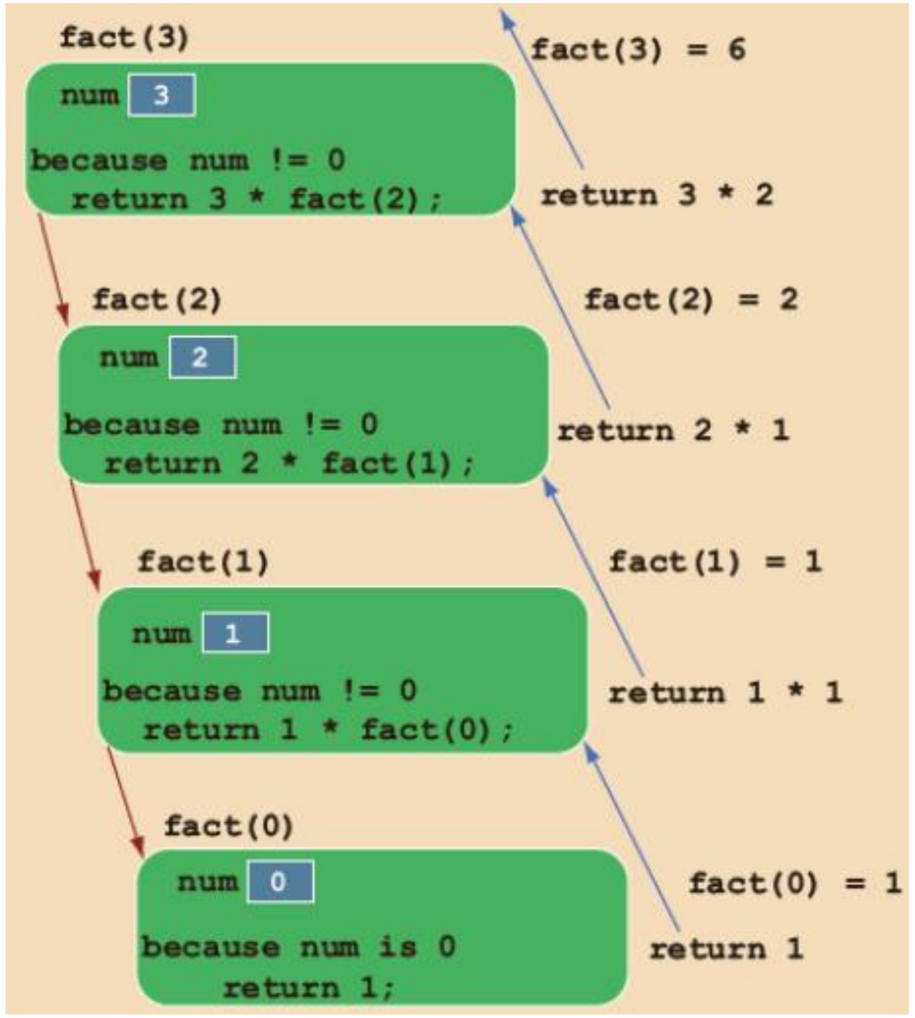
- A function that calls itself is known as a recursive function. And, this technique is known as **recursion**.



```
void recurse() {
    ... .. ...
    recurse();
    ... .. ...
}

int main() {
    ... .. ...
    recurse();
    ... .. ...
}
```

Recursion: Factorial Example

```
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}

int main() {
    int n, result;

    cout << "Enter a non-negative number: ";
    cin >> n;

    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}
```



Exercise

Write a program (using a recursive function) that accept a number n, then print numbers in reverse from n to 1.

For example if the actual parameter is 5, the program should print 5 4 3 2 1.

Function Overloading

Function Signature

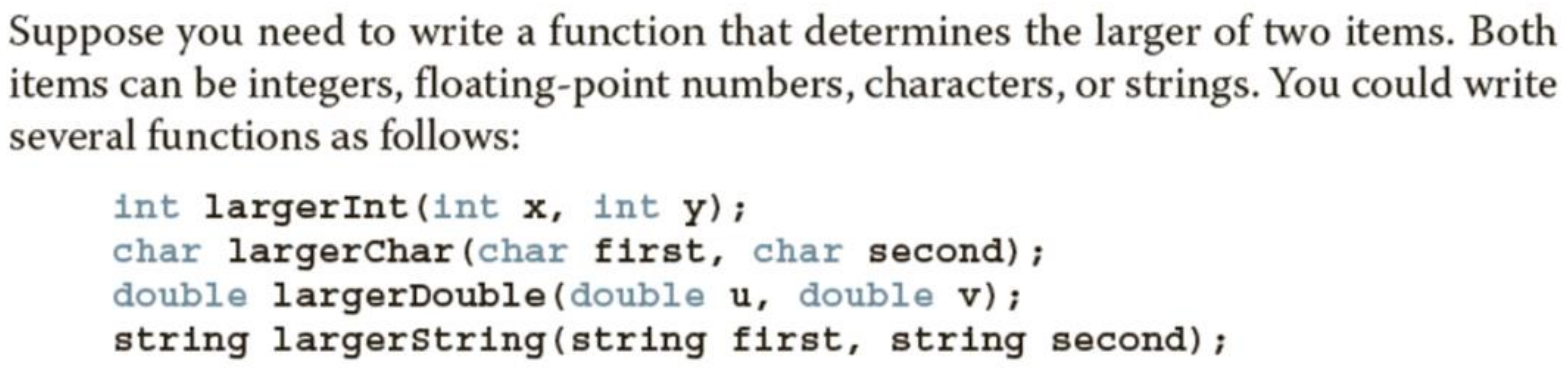
- The **signature** of a function consists of the function name and its formal parameter list.
- Two functions have different signatures if they have either
  - different names, or
  - different formal parameter lists.

  (Note that the signature of a function does not include the return type of the function.)

Function Overloading

- **Overloading a function** refers to having several functions with the same name but different parameter lists. The parameter list determines which function will execute.
- In a C++ program, several functions can have the same name. This is called **function overloading**, or **overloading a function name**.
- Two functions are said to have **different formal parameter lists** if both functions have:
  - A different number of formal parameters, or
  - The same number of formal parameters and the data types of the formal parameters, in the order listed, differ in at least one position.

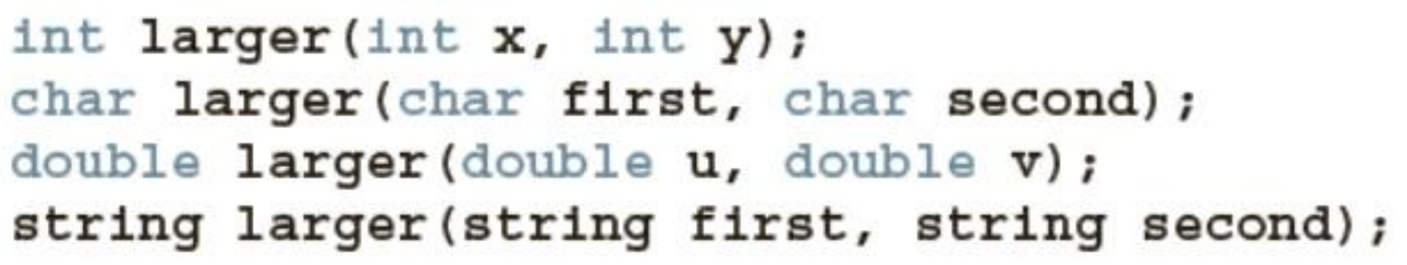
Function Overloading



Suppose you need to write a function that determines the larger of two items. Both items can be integers, floating-point numbers, characters, or strings. You could write several functions as follows:

```
int largerInt(int x, int y);
char largerChar(char first, char second);
double largerDouble(double u, double v);
string largerString(string first, string second);
```

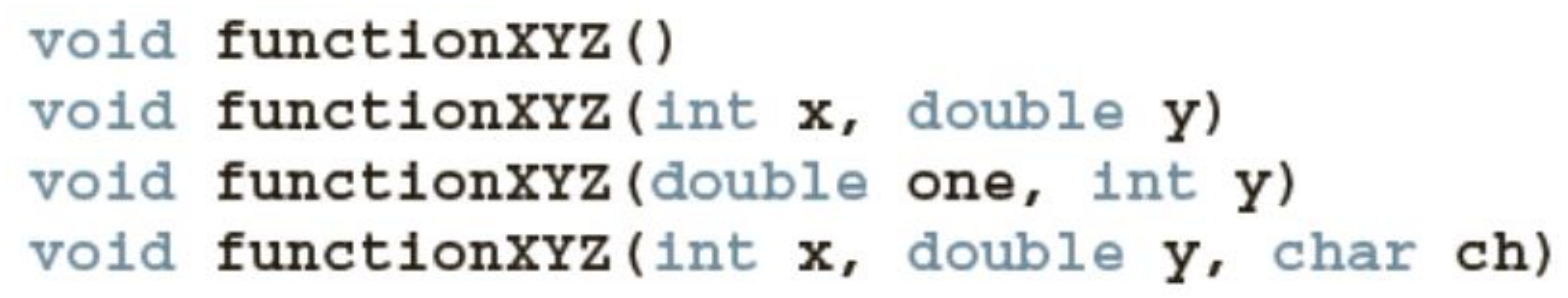
- **You can overload the function larger. Thus, you can write the previous function prototypes simply as follows:**



```
int larger(int x, int y);
char larger(char first, char second);
double larger(double u, double v);
string larger(string first, string second);
```

Function Overloading

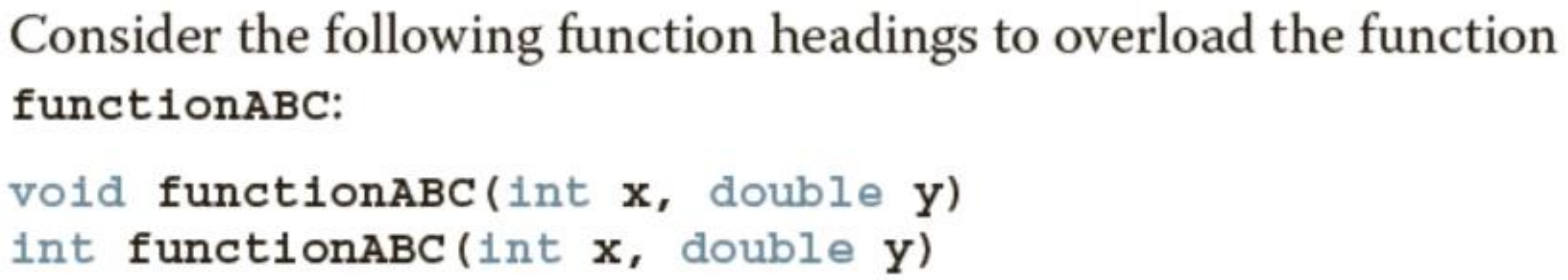
Function overloading using different formal parameter lists:



```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

Exercise

Write a C++ Program to Find Area of Shapes (Circle or Rectangle) using Function Overloading.



- Consider the following function headings to overload the function `functionABC`:

```
void functionABC(int x, double y)
int functionABC(int x, double y)
```

Both of these function headings have the same name and same formal parameter list. Therefore, these function headings to overload the function **functionABC** are **incorrect**.
In this case, the compiler will generate a syntax error. (Notice that the return types of these function headings are different.)

Function Template

(a part of Chapter 13)

Templates

Templates are a very powerful feature of C++. They allow you to write a single code segment for a set of related functions, called a **function template**, and for a set of related classes, called a **class template**. The syntax we use for templates is

```
template <class Type>
declaration;
```

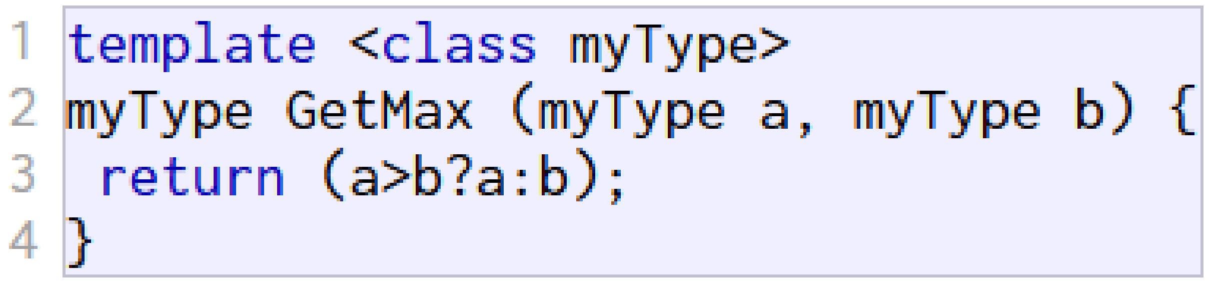
in which `Type` is the name of a data type, built-in or user-defined, and `declaration` is either a function declaration or a class declaration. In C++, `template` is a reserved word. The word `class` in the heading refers to any user-defined type or built-in type. `Type` is referred to as a formal parameter to the template. (Note that in the first line, `template <class Type>`, the keyword `class` can be replaced with the keyword `typename`.)

Function Templates

- **Function templates** are special functions that can operate with *generic types*.
- This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.
- In C++ this can be achieved using ***template parameters***. A template parameter is a special kind of parameter that can be used to pass a **type** as **argument**

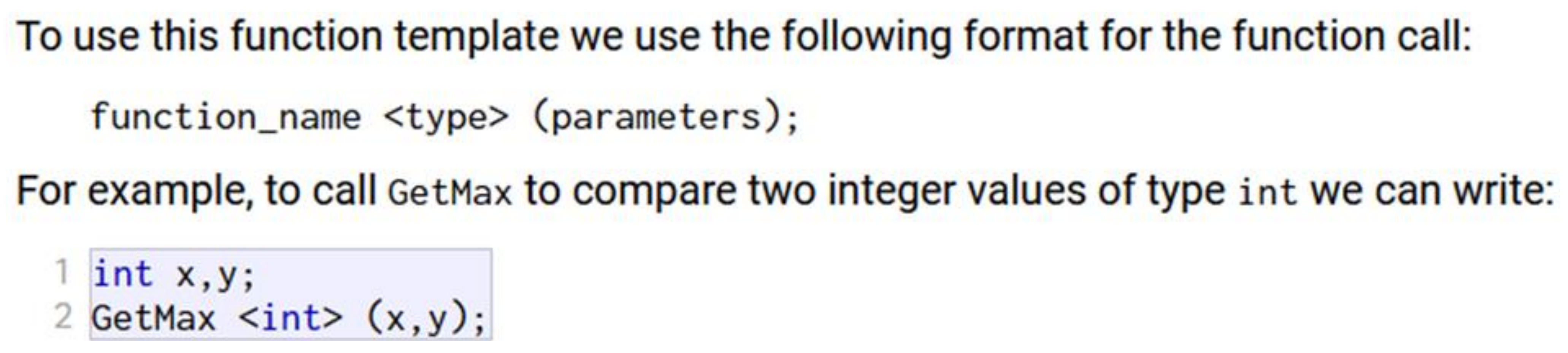
- The format for declaring function templates with type parameters is:
  ```
  template <class identifier> function_declaration;
  template <typename identifier> function_declaration;
  ```
- The only difference between both prototypes is the use of either the keyword class or the keyword typename.
- Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

- For example, to create a template function that returns the greater one of two objects we could use:



```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

- Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.
  Source: https://cplusplus.com/doc/oldtutorial/templates/



To use this function template we use the following format for the function call:

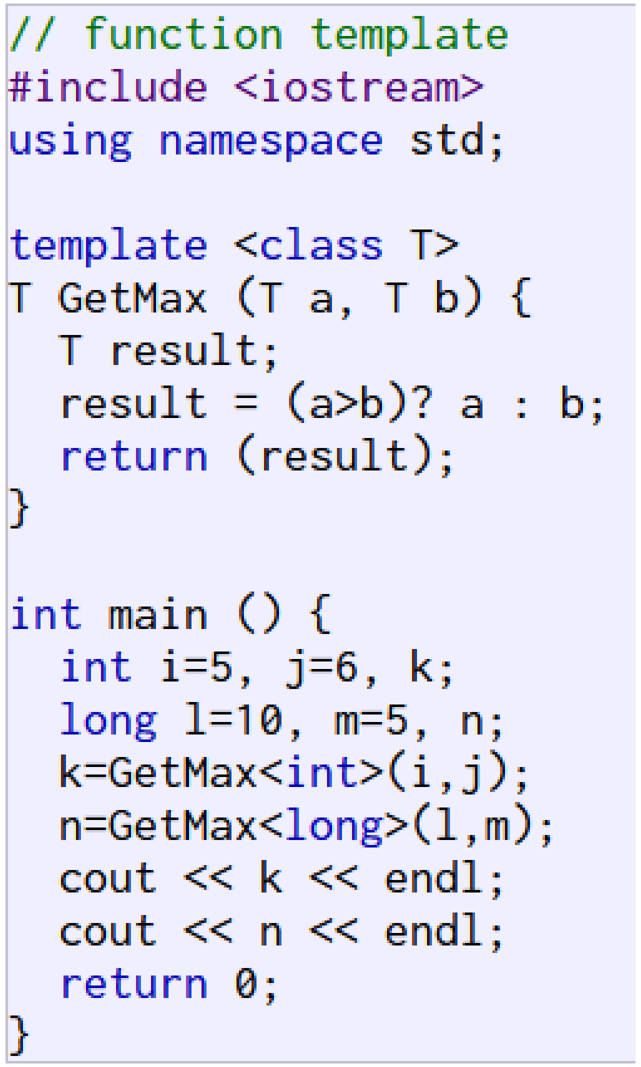
```
function_name <type> (parameters);
```

For example, to call `GetMax` to compare two integer values of type `int` we can write:

```
int x,y;
GetMax <int> (x,y);
```

- When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Function Templates (Example)

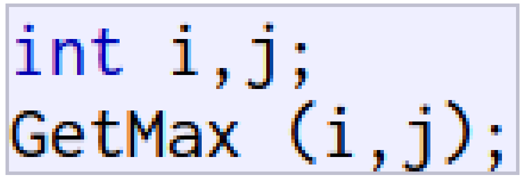


```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

In this case, we have used **T** as the **template parameter name** instead of myType because it is shorter and in fact is a **very common template parameter name**. But you can use any identifier you like.

Source: https://cplusplus.com/doc/oldtutorial/templates/

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:



```
int i,j;
GetMax (i,j);
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Function Templates

```
#include <iostream>                                  //Line 1
                                                     //Line 2

using namespace std;                                 //Line 3

template <class Type>                                //Line 4
Type larger(Type x, Type y);                         //Line 5

int main()                                           //Line 6
{                                                    //Line 7
    cout << "Line 8: Larger of 5 and 6 = "
         << larger(5, 6) << endl;                    //Line 8
    cout << "Line 9: Larger of A and B = "
         << larger('A', 'B') << endl;                //Line 9
    cout << "Line 10: Larger of 5.6 and 3.2 = "
         << larger(5.6, 3.2) << endl;                //Line 10

    return 0;
}
```

```
template <class Type>
Type larger(Type x, Type y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Sample Run:

```
Line 8: Larger of 5 and 6 = 6
Line 9: Larger of A and B = B
Line 10: Larger of 5.6 and 3.2 = 5.6
```

What is the difference between function overloading and templates?

- Both function overloading and templates are examples of polymorphism features of OOP.
  - Function overloading is used when multiple functions do quite similar (not identical) operations,
  - Templates are used when multiple functions do identical operations.

Source: https://www.geeksforgeeks.org/templates-cpp/