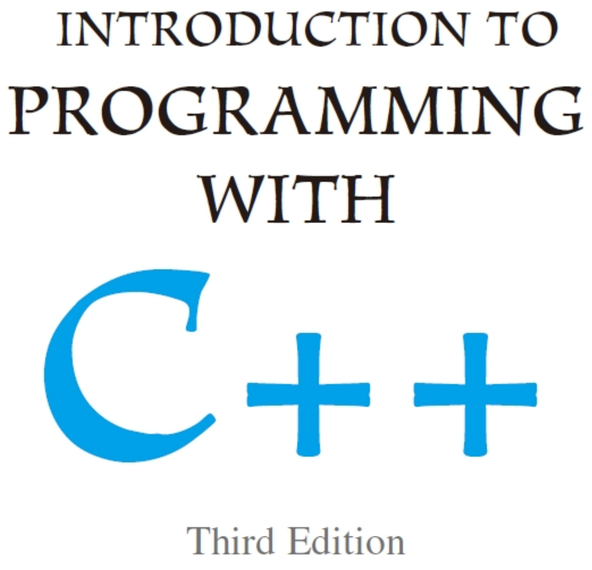
Introduction to Programming with C++

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University



INTRODUCTION TO
PROGRAMMING
WITH
C++

Third Edition

Contents are based on book by Y. Daniel Liang

#define directive

- #define is a Preprocessor Directive that has the following structure:
  `#define identifier replacement-text`
- Before compilation, the compiler replaces any occurrences of identifier that don't occur as part of a string literal or a comment with the replacement-text.

```
#define PI 3.1416 //List6_1a.cpp
int main()
{
   float fRadius;   // radius
   float fArea;     // area

   fArea = PI * fRadius * fRadius;
   cout<<"area of the disk is "<<fArea<<endl;
}
```

#define directive

- \character is used for continuing string literals.

```
#define PI 3.1416 //List6_2a.cpp
#define ERROR_MESSAGE cout<<"Radius needs \
to be positive!"<<endl
int main()
{
   float fRadius;  // radius
   float fArea;    // area
   if(fRadius < 0){
      ERROR_MESSAGE;
   }
   else{
      fArea = PI * fRadius * fRadius;
      cout<<"area of the disk is "<<fArea<<endl;
   }
}
```

#define directive

- Final version

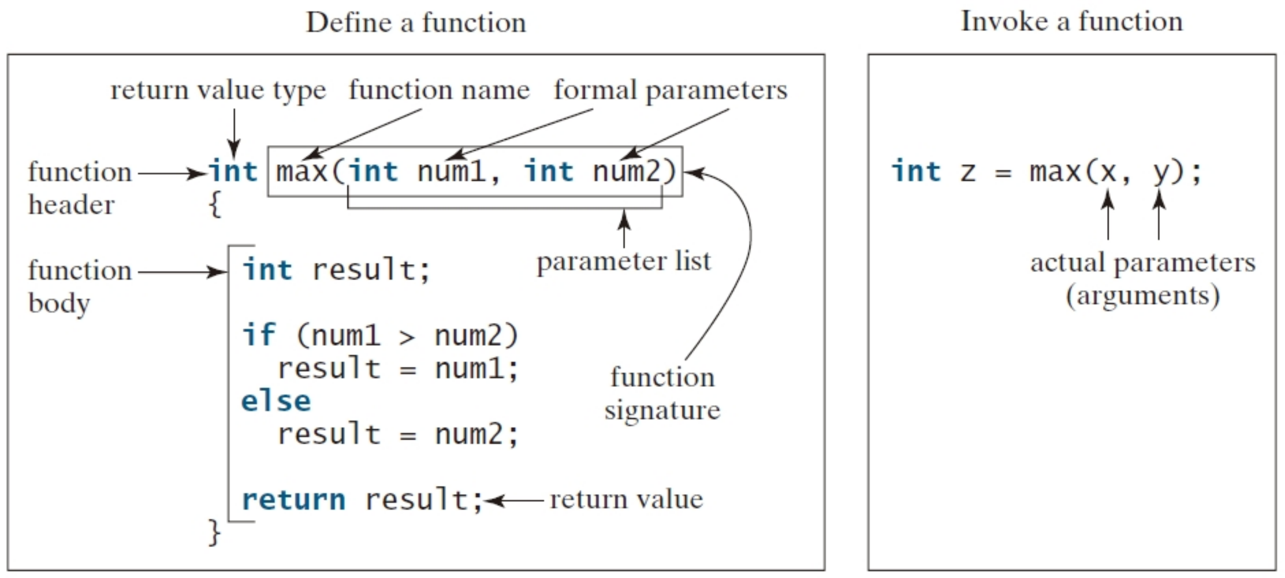
```
#define PI 3.1416 //List6_3.cpp
#define ERROR_MESSAGE cout<<"Radius needs \
to be positive!"<<endl
#define AREA(R)    (PI*R*R)
int main()
{
   float fRadius;   // radius
   float fArea;     // area
   if(fRadius < 0){
      ERROR_MESSAGE;
   }
   else{
      fArea = AREA(fRadius);
      cout<<"area of the disk is "<<fArea<<endl;
   }
}
```

In-Class Exercise: Using #define to write a code which inputs length and width of a rectangle and compute the area of the rectangle.

Function

- Functions can be used to define reusable code and organize and simplify code.
- A function definition consists of its **function name**, **parameters**, **return value type**, and **body**.

```
returnValueType functionName(list of parameters)
{
   // Function body;
}
```



```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

```
int z = max(x, y);
```

Function

- The function header specifies the function's return value type, function name, and parameters.
- A function may return a value. The returnValueType is the data type of that value.
- Some functions perform desired operations without returning a value. In this case, the returnValueType is the keyword **void**.
- The function that returns a value is called a value-returning function and the function that does not return a value is called a void function.
- When a function is invoked, you pass a **value** to the parameter. This value is referred to as an actual parameter or argument.
- The parameter list refers to the type, order, and number of the parameters of a function.

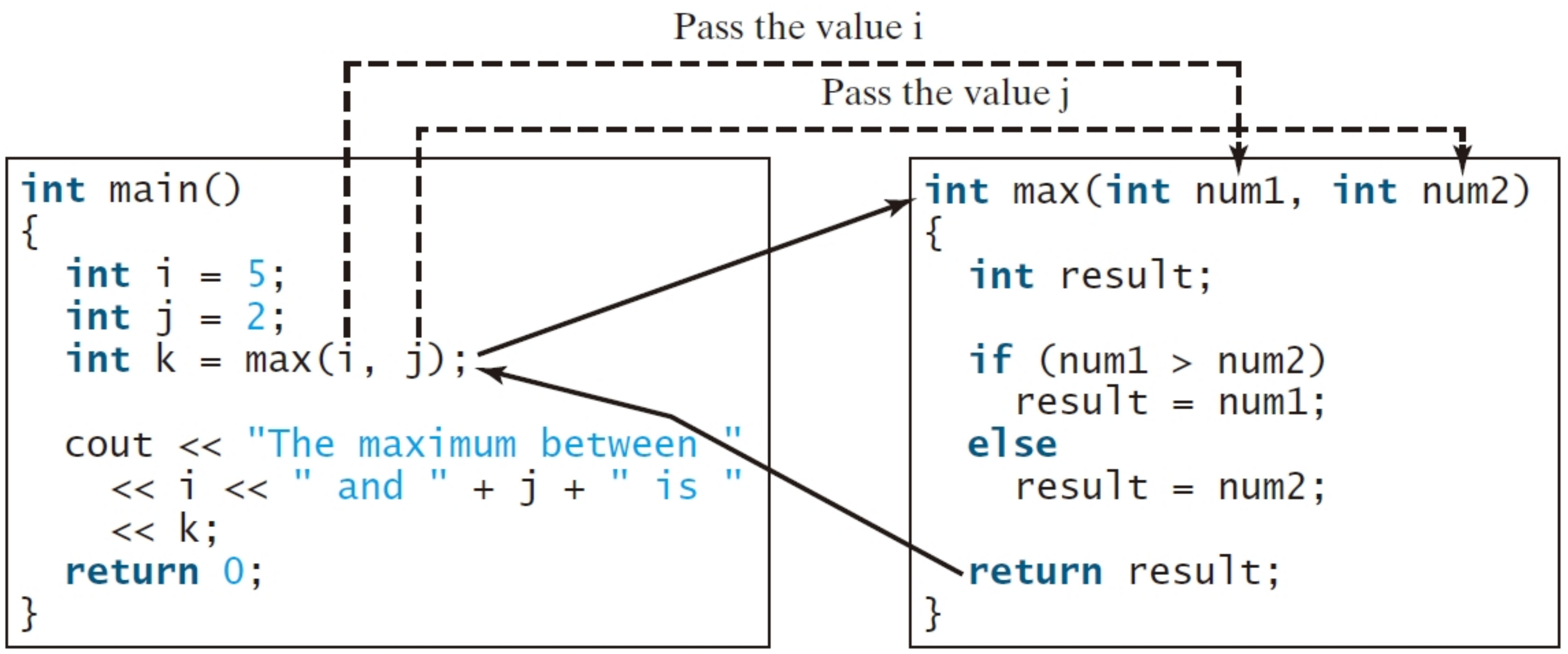
Function

- The function body contains a collection of statements that define what the function does.
- A return statement using the keyword **return** is required for a value-returning function to return a result. The function exits when a return statement is executed.
- To use a function, you have to call or invoke it.
- Occasionally you may need to terminate the program from the function immediately if an abnormal condition occurs. This can be done by invoking the **exit(int)** function defined in the **cstdlib** header.
- You can pass any integer to invoke this function to indicate an error in the program.

```
if (score < 0 || score > 100){
  cout << "Invalid score" << endl;
  // return;
  exit(1);
}
```

Function

- When a program calls a function
  1. Program control is transferred to the called function.
  2. The called function is executed.
  3. A called function returns control to the caller when its return statement is executed or when its function-ending closing brace is reached.



```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Activation record

- Each time a function is invoked, the system creates an **activation record** (also called an activation frame) that stores its arguments and variables for the function and places the activation record in an area of memory known as a **call stack**.

- A call stack is also known as an **execution stack**, **runtime stack**, or **machine stack**, and is often shortened to just "the stack."

- When a function calls another function, the caller's activation record is kept intact and a new activation record is created for the new function call.

- When a function finishes its work and returns control to its caller, its activation record is removed from the call stack.

Function example

```
void ShowErrorMsg()  //List6_4a.cpp
{
   cout<<"Radius needs to be positive!"<<endl;
   return;
}
float CalcArea(float fRadius)
{
   float fArea = PI * fRadius * fRadius;
   return fArea;
}
int main()
{
   float fRadius;
   float fArea;

   cout << "Please input the radius: ";
   cin >> fRadius;
   if(fRadius < 0)
      ShowErrorMsg();
   else{
      fArea = CalcArea(fRadius);
      cout << "Area is " << fArea << endl;
   }
```

Function Prototypes (Declaration)

- Before a function is called, its header must be declared.
- We could define a **function prototype** before the function is called.
- A function prototype, also known as function declaration, is a function header without implementation.
- The implementation is given later in the program (other files).

```
void ShowErrorMsg(); //List6_5.cpp
float CalcArea(float fRadius);

int main()
{
   if(fRadius < 0)
      ShowErrorMsg();
   else{
      fArea = CalcArea(fRadius);
   }
}
void ShowErrorMsg()
{
   cout<<"Radius needs to be positive!"<<endl;
   return;
}
float CalcArea(float fRadius)
{
   float fArea = PI * fRadius * fRadius;
   return fArea;
}
```

Function Prototypes

- In the prototype, you need not list the parameter names, only the parameter types. C++ compiler ignores the parameter names.

```
void ShowErrorMsg();
float CalcArea(float);
```

- **Declaring** a function specifies what a function is without implementing it.

- **Defining** a function gives a function body that implements the function.

Default Arguments

- C++ allows you to declare functions with default argument values.
- The default values are passed to the parameters when a function is invoked without the arguments.

```
float CalcArea(float);
int main()
{
  cout <<  CalcArea(2.0) << endl;
  cout <<  CalcArea() << endl;
}
float CalcArea(float fRadius = 1.0)
{
   float fArea = PI * fRadius * fRadius;
   return fArea;
}
```

- When a function contains a mixture of parameters with and without default values, those with default values must be declared last.

```
void t3(int x, int y = 0, int z = 0);
t3(1); // Parameters y and z are assigned a default value
```

Inline Functions

- Implementing a program using functions makes the program easy to read and easy to maintain, but function calls involve runtime overhead (i.e., pushing arguments and CPU registers into the stack and transferring control to and from a function).

- C++ provides **inline functions** for improving performance for short functions.

- Inline functions are not called; rather, the compiler copies the function code in line at the point of each invocation.

```
inline float CalcArea(float);
int main()
{
  cout <<  CalcArea(2.0) << endl;
  cout <<  CalcArea() << endl;
}
inline float CalcArea(float fRadius = 1.0)
{
   float fArea = PI * fRadius * fRadius;
   return fArea;
}
```

Modularizing Code

- Modularizing makes the code easy to maintain and debug and enables the code to be reused.

- Functions can be used to reduce redundant code and enable code reuse. Functions can also be used to modularize code and improve the program's quality.

- See List6_4.cpp

- By encapsulating the code for obtaining the GCD in a function, this program has several advantages:

  1. It isolates the problem for computing the GCD from the rest of the code in the main function. Thus, the logic becomes clear and the program is easier to read.

  2. If there are errors on computing GCD, they will be confined in the gcd function, which narrows the scope of debugging.

  3. The gcd function now can be reused by other programs.

  Homework 6.1: Apply the concept of code modularization to improve List5_17.

Overloading Functions

- Function overloading means when two functions have the same name but different parameter lists within one file.
- The C++ compiler determines which function is used based on the function signature.
- Overloading functions enables you to define the functions with the same name as long as their signatures are different.
- The max function that was used earlier works only with the int data type. Overloading function with the same name but floating-point parameters

```
double max(double num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

- See, List6_6.cpp.

Homework

Homework 6.2: Use overloading functions with default arguments to write a code to compute the product of 3 integers and floating point numbers. (What default values should be used? So that the result remain valid when less numbers are inputed.

Life scope of variables

- A variable can be declared as a local, a global, or a static local in C++.
- The scope of a variable is the part of the program where the variable can be referenced.
- A variable defined inside a function is referred to as a local variable. Local variables do not have default values.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
- Global variables are declared outside all functions and are accessible to all functions in their scope. Global variables are defaulted to zero.
- The scope of a global variable starts from its declaration and continues to the end of the program.

Life scope of variables

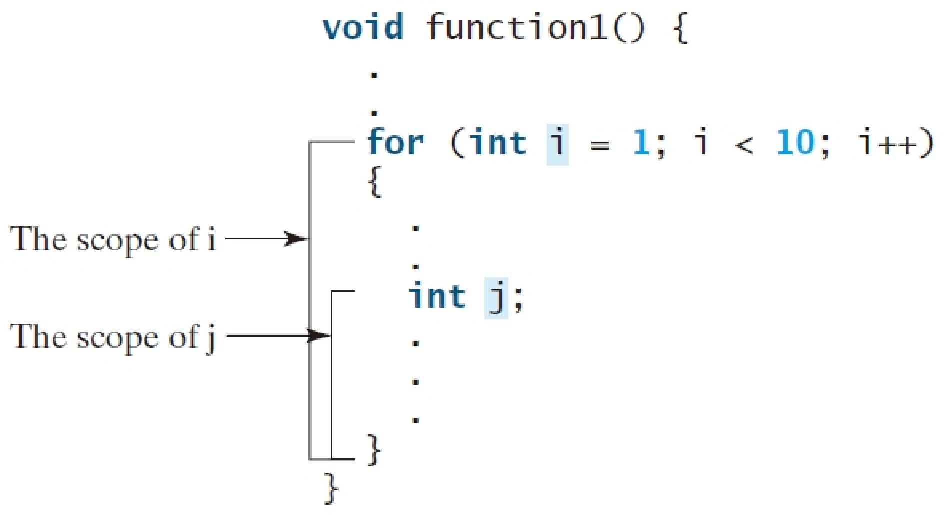
```
int x = 555; /*---1---*/  //List6_11a.cpp
void print_x(void)
{
  cout<<" in print_x() x = "<< x <<'\n';
}
int main(void)
{
  cout << " global x = " << x << '\n';
  int x = 333; /*---2---*/

  cout<<" in main() x = "<< x <<'\n';
  print_x();
  return (0);
-----------------------------------------------------
 global x = 555
 in main() x = 333
 in print_x() x = 555
}
```

The Scope of Variables in a for Loop

- A variable declared in the initial-action part of a for-loop header has its scope in the entire loop.
- However, a variable declared inside a for-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable.

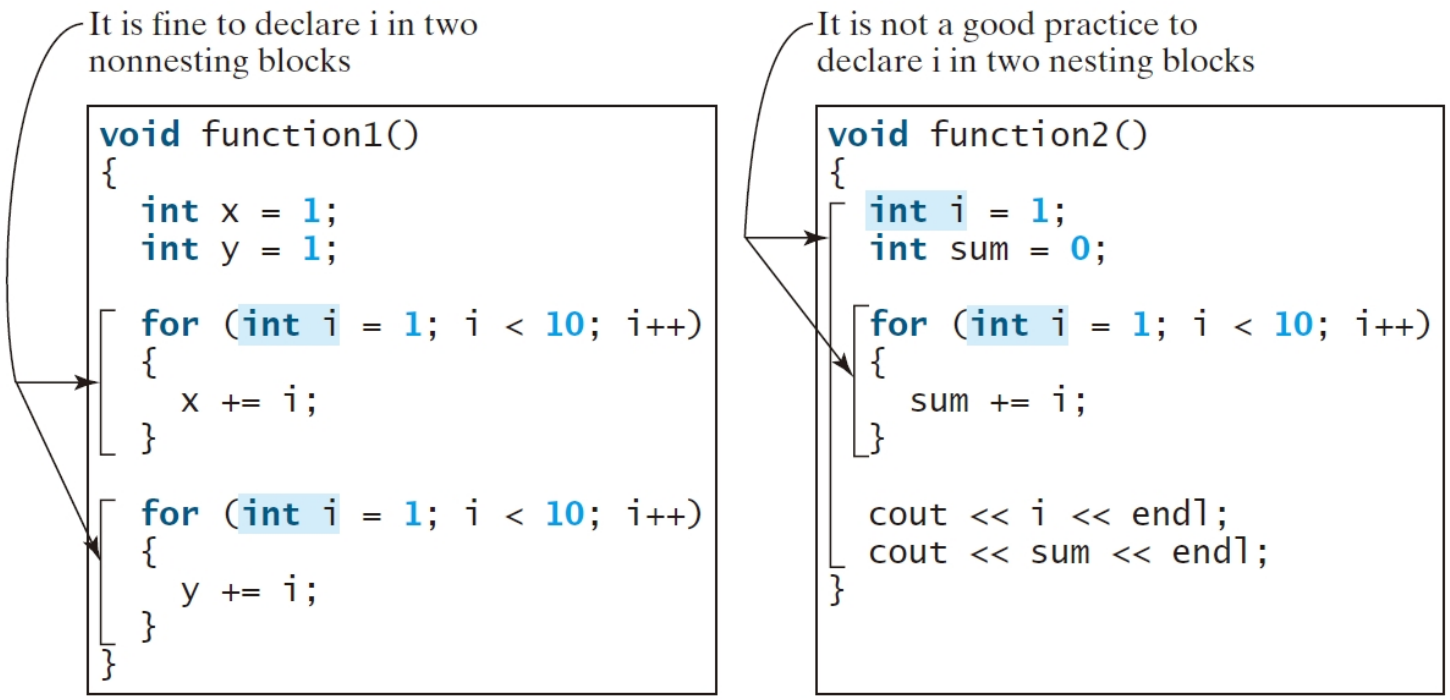
```
int x = 555;     /*---1---*/   //List6_11a.cpp
int main(void)
{
  cout << " global x = " << x << '\n';
  int x = 333; /*---2---*/
  cout << " in main() x = " << x << '\n';
  for(int i = 0; i < 5; i++)
  {
    int x = i * 100; /*---3---*/
    cout << " in loop x = " << x << '\n';
  }
  cout << " in main x = " << x << '\n';
-----------------------------------------
 global x = 555
 in main() x = 333
 in loop x = 0
 in loop x = 100
 in loop x = 200
 in loop x = 300
 in loop x = 400
 in main x = 333
```



Life scope of variables

- It is commonly acceptable to declare a local variable with the same name in different nonnesting blocks in a function.

- But it is not a good practice to declare a local variable twice in nested blocks, even though it is allowed in C++.


It is fine to declare i in two nonnesting blocks
It is not a good practice to declare i in two nesting blocks

```
void function1()
{
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++)
  {
    x += i;
  }

  for (int i = 1; i < 10; i++)
  {
    y += i;
  }
}
```

```
void function2()
{
  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
  {
    sum += i;
  }

  cout << i << endl;
  cout << sum << endl;
}
```

- Do not declare a variable inside a block and then attempt to use it outside the block.

```
for (int i = 0; i < 10; i++){
}
cout << i << endl;
```

Static Local Variables

- After a function completes its execution, all its local variables are destroyed. These variables are also known as automatic variables.
- C++ allows you to declare static local variables. Static local variables are permanently allocated in the memory for the lifetime of the program.
- The initialization of static variables happens only once in the first call.

```
void t1(); // Function prototype List6_12.cpp
int main()
{
 t1();
 t1();
}
void t1()
{
 static int x = 1;
 int y = 1;
 x++; y++;
 cout << "x is " << x << endl;
 cout << "y is " << y << endl;
}
------------------------------------
x is 2
y is 2
x is 3
y is 2
```

Passing Arguments by Value

- When you invoke a function with a parameter, the **value** of the argument is passed to the parameter.
- This is referred to as **pass-by-value**.
- The variable is not affected, regardless of the changes made to the parameter inside the function.

```
void increment(int n) \\List6-13.cpp
{
  n++;
  cout << "\t inside the function is " << n << endl;
 }
int main()
{
  int x = 1;
  cout << "Before the call, x is " << x << endl;
  increment(x);
  cout << "After the call, x is " << x << endl;
}
-------------------------------------------------------
Before the call, x is 1
         inside the function is 2
after the call, x is 1
```

Passing Arguments by Reference

- The parameter is a variable in the function with its own memory space. The variable is allocated when the function is invoked, and it disappears when the function is returned to its caller.
- The function can accomplish this by passing a reference to the variables.
- C++ provides a special type of variable-a reference variable- which can be used as a function parameter to reference the original variable.

```
void increment(int &n) \\List6-13a.cpp
{
  n++;
  cout << "\t inside the function is " << n << endl;
 }
int main()
{
  int x = 1;
  cout << "Before the call, x is " << x << endl;
  increment(x);
  cout << "After the call, x is " << x << endl;
}
-------------------------------------------------------
Before the call, x is 1
         inside the function is 2
After the call, x is 2
```

Reference variable

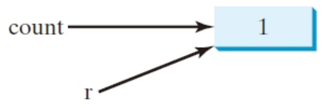
- To declare a reference variable, place the ampersand (&) in front of the variable or after the data type for the variable.
- A reference variable is an alias for another variable.
  `int &r = count;` or
  `int& r = count;`

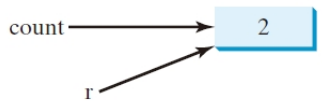
```
int main()  //List6_15.cpp
{
  int count = 1;
  int& r = count;
  cout << "count is " << count << endl;
  cout << "r is " << r << endl;

  r++;
  cout << "count is " << count << endl;
  cout << "r is " << r << endl;

  count = 10;
  cout << "count is " << count << endl;
  cout << "r is " << r << endl;
}
```

```
count is 1
r is 1
count is 2
r is 2
count is 10
r is 10
```





- You can use a reference variable as a parameter in a function and pass a regular variable to invoke the function. The parameter becomes an alias for the original variable. This is known as pass-by-reference.

Passing Arguments by Reference

- In pass-by-value, the actual parameter and its formal parameter are independent variables.
- In pass-by-reference, the actual parameter and its formal parameter refer to the same variable.
- Pass-by-reference is more efficient than pass-by-value for object types such as strings, because objects can take a lot of memory.
- However, the difference is negligible for parameters of primitive types such int and double.
- So, if a primitive data type parameter is not changed in the function, you should simply declare it as pass-by-value parameter.

  Homework 6.2: Write a code containing a function to swap the two integers.
  Homework 6.3: Programming exercise: 6.15, 6.33.