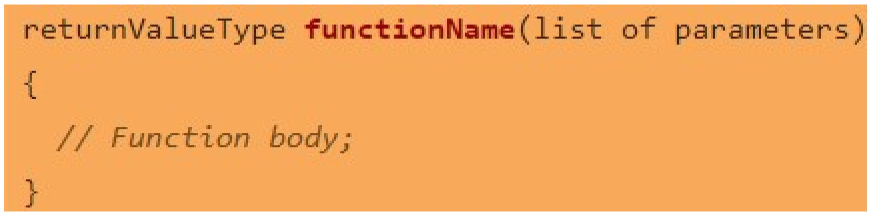
6.1 Intro

Monday, February 6, 2023 10:11 AM

- Functions, define reusable code, organize and simplify code
- Function is collection of statements grouped together to perform an operation

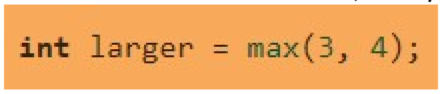
6.2 Defining a Function

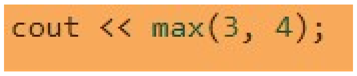
Monday, February 6, 2023 10:14 AM

- Syntax:
    - 
      ```
      returnValueType functionName(list of parameters)
      {
        // Function body;
      }
      ```
- Function header specifies function's return value type, function name, and parameters
- Function may return a value, returnValueType is data type of that value, if nothing than can just be void, function is called void function
- Variables declared in the function header known as parameters, like a placeholder, when function used, u pass in a val to the parameter
- Parameter list – type, order, and number of parameters of a function
- Function name + parameter list = function signature
- Function body has collection of statements that define what function does

6.3 Calling a Function

Monday, February 6, 2023 10:33 AM

- To use function, have to call it / invoke it
- Pgrm that calls the function = caller, 2 ways to call:
  - 

    ```
    int larger = max(3, 4);
    ```
  -
  - 

    ```
    cout << max(3, 4);
    ```
- When pgrm calls a function, transfer of ctrl, called function executed
- Sys makes activation record that stores arguments and vars for the function & puts activation record in area of mem known as call stack (aka machine stack, …, the stack)
- Stack stores activation records in list in first out

6.4 void Functions

Monday, February 6, 2023 10:58 AM

- Its void, so don't return anything
- Can use cstdlib header to invoke exit(int) function to terminate pgrm immediately

6.5 Passing Arguments by Value

Monday, February 6, 2023 11:07 AM

- Pwr of function is they work w/ parameters

6.6 Modularizing Code

Monday, February 6, 2023 11:17 AM

- Functions used to lower redundant code

6.7 Overloading Functions

Monday, February 6, 2023 11:20 AM

- Can have many functions w/ same name but diff parameters
- Make pgrm more readable, but they gotta have diff parameter lists

6.8 Function Prototypes

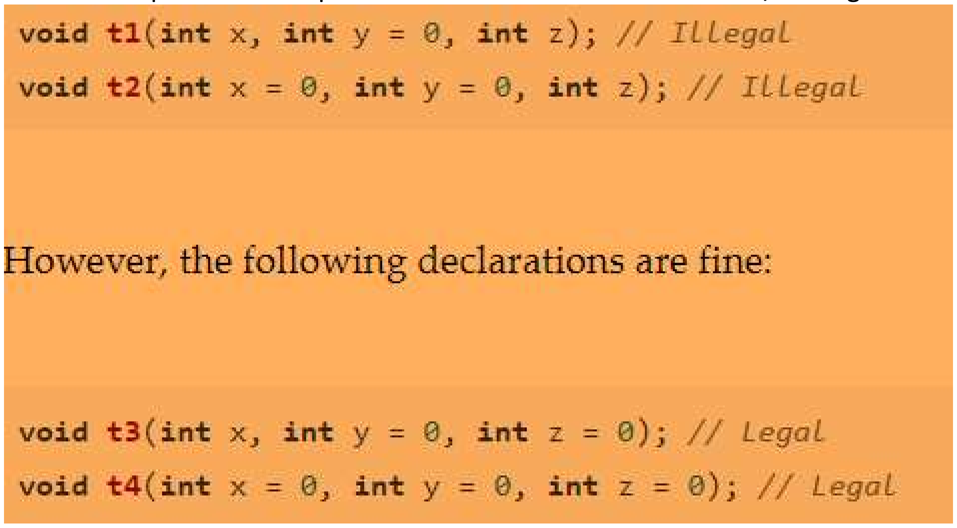
Monday, February 6, 2023 1:06 PM

- B4 function called, declare header
- Easy to just put definition b4 all function calls, or can do function prototype (aka function declaration)
- Function declaration- function header w/out implementation, impl later in pgrm
- Don't need to list parameter names, only parameter types
- Declaring function- specifies what a function is w/ implementing
- Define a function- gives function body that implements the function

6.9 Default Arguments

Monday, February 6, 2023 1:13 PM

- Default vals passed to the parameters when function invoked w/out arguments



```
void t1(int x, int y = 0, int z); // Illegal
void t2(int x = 0, int y = 0, int z); // Illegal
```

- However, the following declarations are fine:

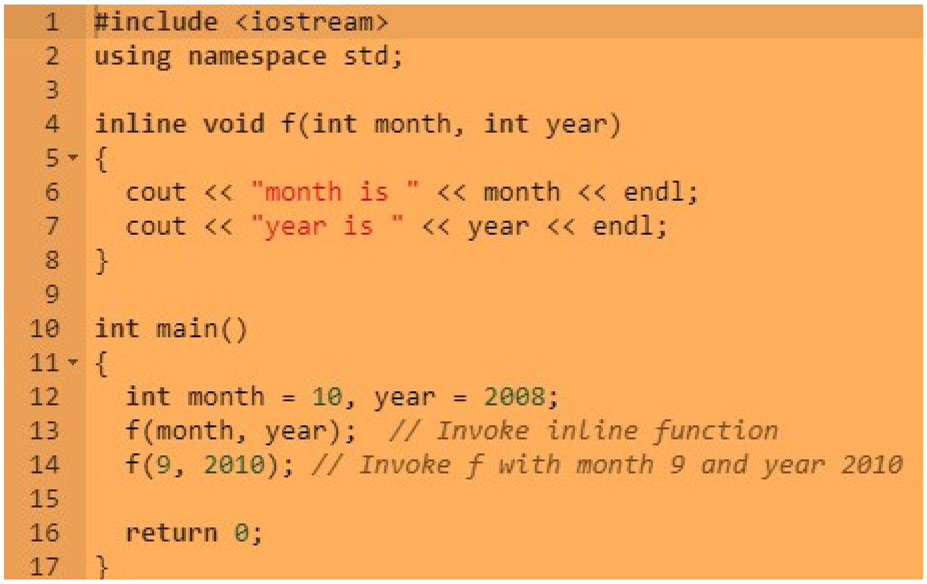
```
void t3(int x, int y = 0, int z = 0); // Legal
void t4(int x = 0, int y = 0, int z = 0); // Legal
```

-

6.10 Inline Functions

Monday, February 6, 2023 1:24 PM

- Function calls involve runtime overhead
  - Where things do too much, like pushing arguments and CPU registers into the stack and transferring ctrl to and from a function
- C++ has inline functions, avoids function calls, they aren't called, but compiler copies function code in line @ pt of each invocation
- To specify inline function, proceed the function declaration w/ inline keyword
- Looks same for prgmer, but easier for pgrm itself
- Inline functions good for short function, not long ones in many places in pgrm, multiple copies will increase execute time
- C++ lets compiler skip inline keyword is function is too long, so its just a request, lets compiler decide
-

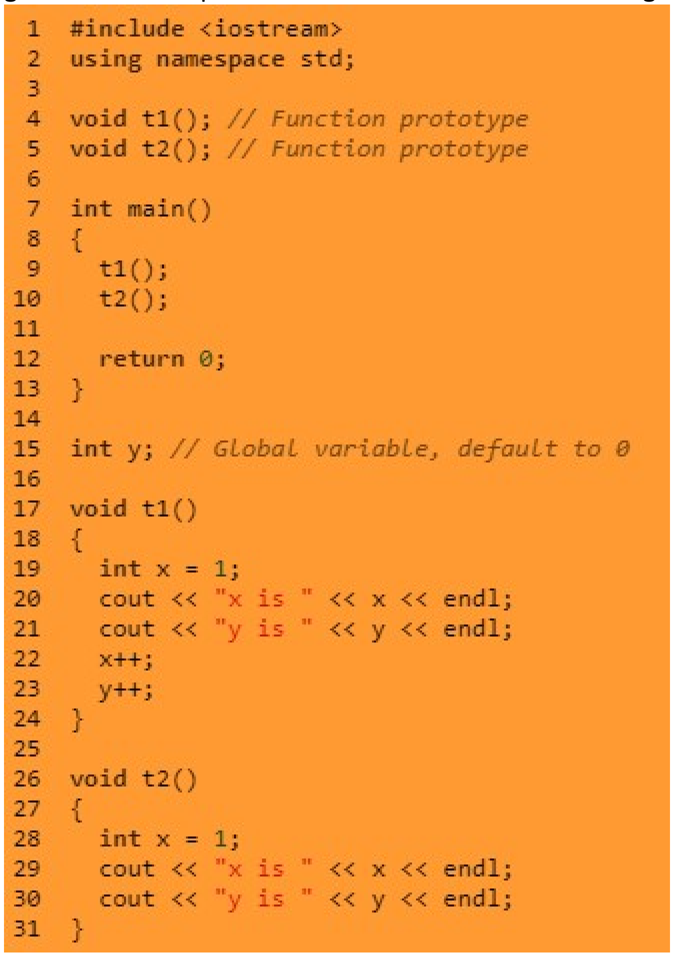


```cpp
#include <iostream>
using namespace std;

inline void f(int month, int year)
{
  cout << "month is " << month << endl;
  cout << "year is " << year << endl;
}

int main()
{
  int month = 10, year = 2008;
  f(month, year);  // Invoke inline function
  f(9, 2010); // Invoke f with month 9 and year 2010

  return 0;
}
```

-

6.11 Local, Global, and Static Local Variables

Monday, February 6, 2023 2:08 PM

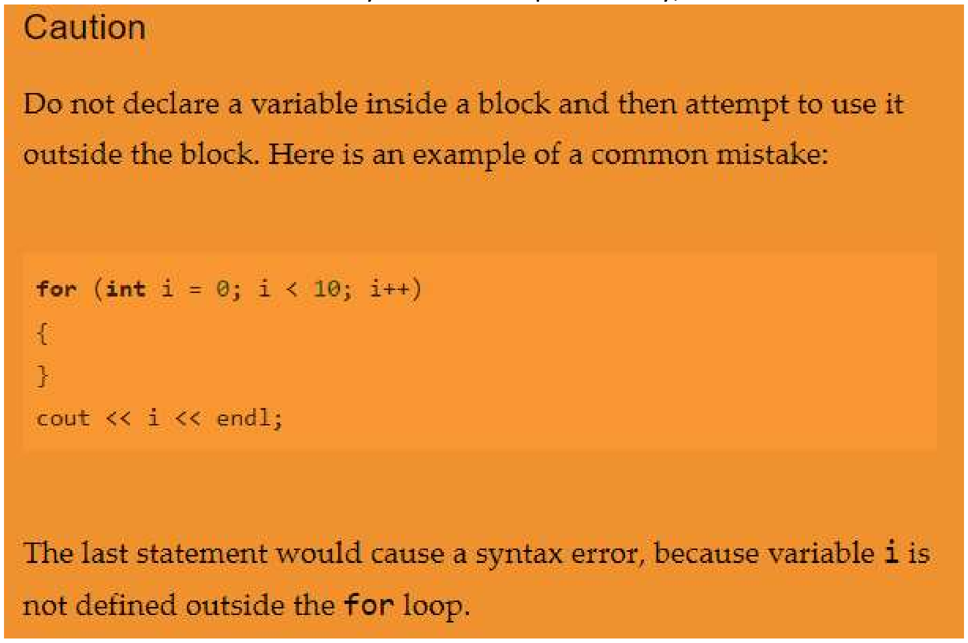
- Scope of var- part of pgrm where var can be referenced
- Var in a function = local var
- Global var= declared outside all functions, accessible to all functions in their scope, have default vals of 0 (locals don't have default val)
- Var must be declared b4 used, scope of local var starts from declaration, cont to end of block w/ the var
- Scope of glbl var starts @ declaration till end of pgrm
- Parameter is actually local var, scope is entire function
- Bad practice to declare glbl var once then use it all over bc hard to debug, avoid using glbl vars, glbl constants is permitted since constants don't change
- 

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

void t1(); // Function prototype
void t2(); // Function prototype

int main()
{
  t1();
  t2();

  return 0;
}

int y; // Global variable, default to 0

void t1()
{
  int x = 1;
  cout << "x is " << x << endl;
  cout << "y is " << y << endl;
  x++;
  y++;
}

void t2()
{
  int x = 1;
  cout << "x is " << x << endl;
  cout << "y is " << y << endl;
}
```

-

6.11.1 The Scope of Variables in a for Loop

Monday, February 6, 2023 6:30 PM

- Var declared in initial-action pt of for-loop header has scope of whole loop
- Var in for-loop body has scope in loop body
- Can name vars the same if they have diff scopes entirely, not recommended
- 

  Caution

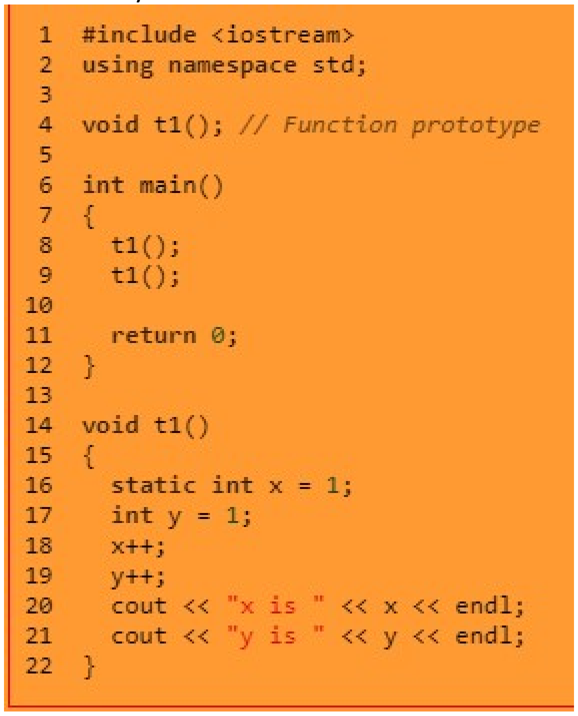
  Do not declare a variable inside a block and then attempt to use it outside the block. Here is an example of a common mistake:

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

  The last statement would cause a syntax error, because variable **i** is not defined outside the **for** loop.
-

6.11.2 Static Local Variables

Monday, February 6, 2023 6:33 PM

- After function complete execution, all local vars are destroyed
- These called automatic variables
- If wanna keep vals stored in local vars, can declare static local variables= permanently allocated in mem for life of pgrm, var val kept for next function call
- Just use keyword static
- 

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

void t1(); // Function prototype

int main()
{
  t1();
  t1();

  return 0;
}

void t1()
{
  static int x = 1;
  int y = 1;
  x++;
  y++;
  cout << "x is " << x << endl;
  cout << "y is " << y << endl;
}
```

-

- The following program invokes p() three times. What is the output from the last call of p()?

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

int j = 40;

void p()
{
  int i = 5;
  static int j = 5;
  i++;
  j++;

  cout << "i is " << i << " j is " << j << endl;
}

int main()
{
  p();
  p();
  p();

  return 0;
}
```

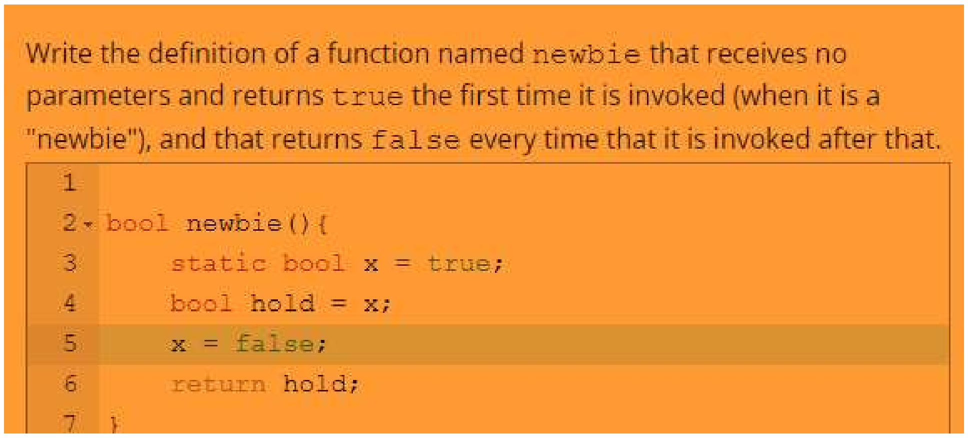
-

- ( ) i is 6 j is 6
  ( ) i is 6 j is 7
  [✓ i is 6 j is 8]
  ( ) i is 6 j is 9

  Fantastic!

  See Section 6.11.2.

-

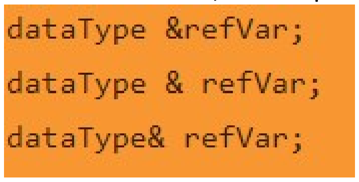
- 

  Write the definition of a function named `newbie` that receives no parameters and returns `true` the first time it is invoked (when it is a "newbie"), and that returns `false` every time that it is invoked after that.

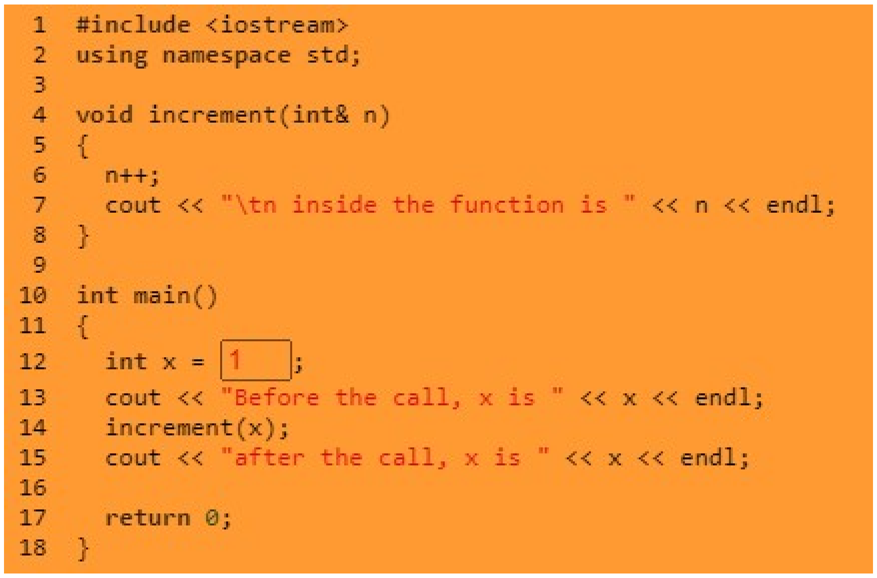
  ```

  bool newbie(){
      static bool x = true;
      bool hold = x;
      x = false;
      return hold;
  }
  ```
-

6.12 Passing Arguments by Reference

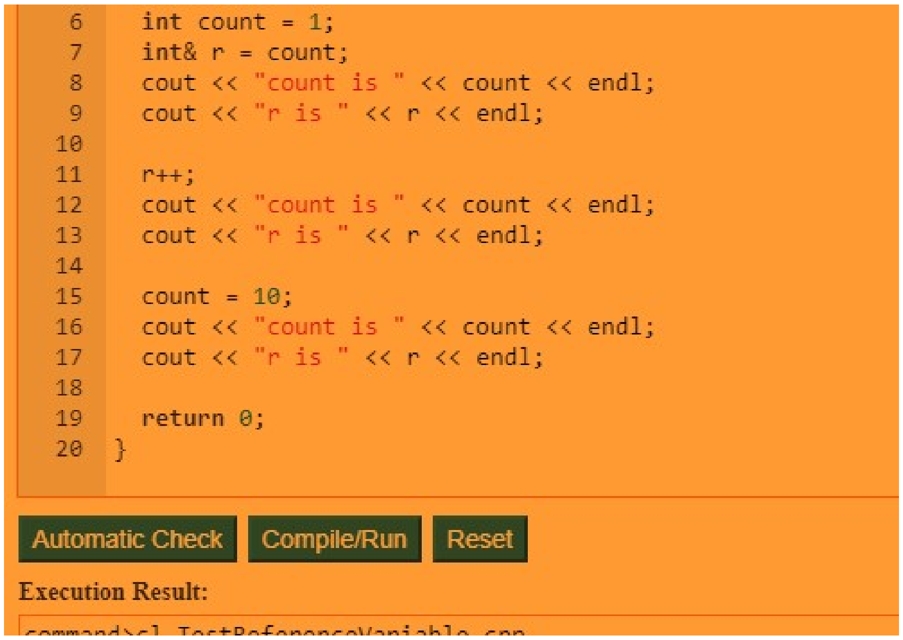
Monday, February 6, 2023 7:24 PM

- When invoke functions w/ parameter, val of argument is passed to the parameter = pass-by-value
- If argument is var instead of literal val, the var passed to the parameter, var not affected, even when change made to parameter inside of function
- If want to have function to actually swap 2 vars, use reference vars= function parameter that references the original var, can be used as a function parameter to reference the og var
- Access and mod the og data stored in that var thru reference var, its alias for another var
- To declare ref var, use ampersand sign (&) in front of the var or after the data type of var
- 

```
dataType &refVar;
dataType & refVar;
dataType& refVar;
```

- Can use reference var as parameter in function & pass a reg var to invoke the function= pass-by-reference
- 

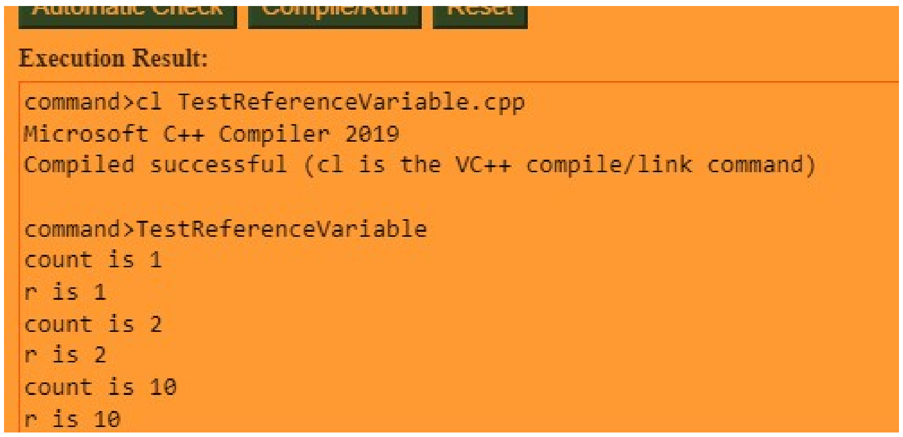
```
#include <iostream>
using namespace std;

void increment(int& n)
{
  n++;
  cout << "\tn 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;

  return 0;
}
```

- 

```
  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;

  return 0;
}
```

Execution Result:

```
command>cl TestReferenceVariable.cpp
```



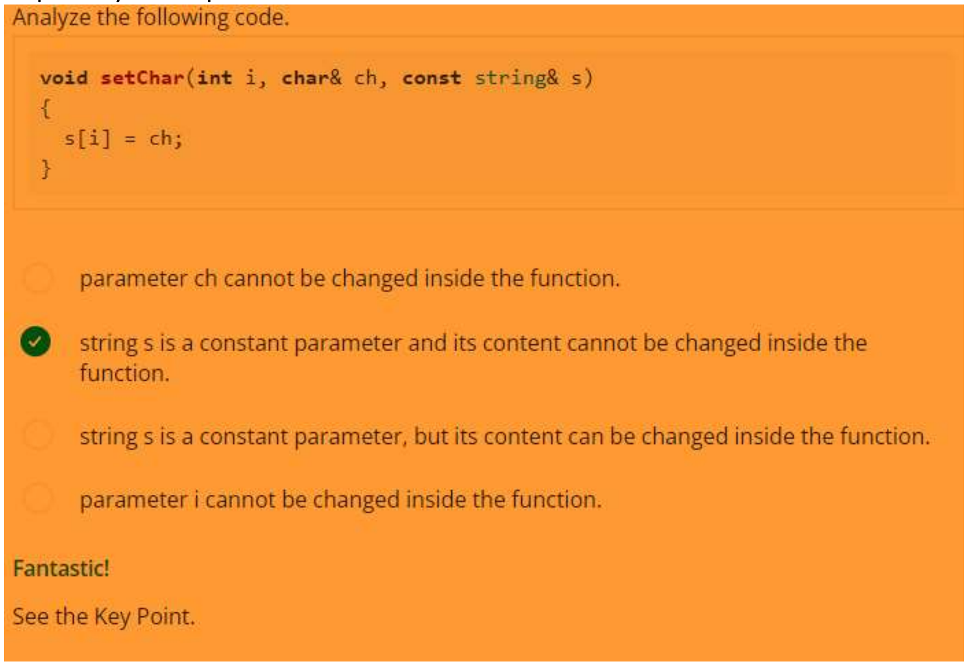
Execution Result:

```
command>cl TestReferenceVariable.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestReferenceVariable
count is 1
r is 1
count is 2
r is 2
count is 10
r is 10
```

- When pass arg by ref, formal parameter and argument must have same type
    -

6.13 Constant Reference Parameters

Monday, February 6, 2023 7:43 PM

- If ur pgrm uses pass-by-ref parameter & parameter not changed in function, mark it constant so compiler know not to change that parameter.
- Do w/ keywork const b4 parameter in function declaration
- It called constant reference parameter
- Pass-by-ref is more efficient than pass-by-val for objects like strings (bc objects take lot of mem)
- So, if a primitive data type parameter is not changed in the function, you should simply declare it as pass-by-value parameter.
- 

  Analyze the following code.

  ```
  void setChar(int i, char& ch, const string& s)
  {
    s[i] = ch;
  }
  ```

  parameter ch cannot be changed inside the function.

  [✓ string s is a constant parameter and its content cannot be changed inside the function.]

  string s is a constant parameter, but its content can be changed inside the function.

  parameter i cannot be changed inside the function.

  Fantastic!

  See the Key Point.
-
-

6.14 Case Study: Converting Hex to Decimals

Monday, February 6, 2023 7:50 PM

- Pgrm prompt user to enter hex numb as string and convert to decimal using function:
    - `int hex2Dec(const string& hex)`
- Note that

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_1 \times 16^1 + h_0 \times 16^0$$
$$= (\ldots((h_n \times 16 + h_{n-1}) \times 16 + h_{n-2}) \times 16 + \ldots + h_1) \times 16 + h_0$$

-

6.15 Function Abstraction & Stepwise Refinement

Monday, February 6, 2023 7:54 PM

- Levels of abstraction
- Function abstraction achieved by sep use of function from implementation, information hiding (aka encapsulation, when details of implementation of function hidden from client)
    - We don't know how rand() and time(0) are implemented
- Fun abstract can be applied to dev pgrms, when write large pgrm, use "divide and conquer" strat (aka stepwise refinement), make prob to subproblems, manageable
- Isolate details from design, later use to implement details
- Now look @ implementation, usually subprob go w/ function in implementation, decide which modules to implement as function and which to combo w/ other functions (based on easy to read code)
    - Top down implementation- implement 1 function @ a time, stubs (simple, incomplete version of function) used for the functions waiting for implementation
    - Bottom-up implements 1 function @ time, write test pgrm (driver)
    - Both implement functions incrementally, help isolate errors
- Benefits of Stepwise refinement
    - Simpler pgrm
    - Reusing functions
    - Easier dev, debugging, & testing
    - Better for teamwork

Ch 6 Summary

Monday, February 6, 2023 8:06 PM

1. Making programs modular and reusable is a goal of software engineering. Functions can be used to develop modular and reusable code.
2. The function header specifies the return value type, function name, and parameters of the function.
3. A function may return a value. The returnValueType is the data type of the value that the function returns.
4. If the function does not return a value, the returnValueType is the keyword void.
5. The parameter list refers to the type, order, and number of the parameters of a function.
6. The arguments that are passed to a function should have the same number, type, and order as the parameters in the function signature
7. The function name and the parameter list together constitute the function signature.
8. Parameters are optional; that is, a function may contain no parameters.
9. A value-returning function must return a value when the function is finished.
10. A return statement can be used in a void function for terminating the function and returning control to the function's caller.
11. When a program calls a function, program control is transferred to the called function.
12. A called function returns control to the caller when its return statement is executed or its function-ending closing brace is reached.
13. A value-returning function also can be invoked as a statement in C++. In this case, the caller simply ignores the return value.
14. A function can be overloaded. This means that two functions can have the same name as long as their function parameter lists differ.
15. Pass-by-value passes the value of the argument to the parameter.
16. Pass-by-reference passes the reference of the argument.
17. If you change the value of a pass-by-value argument in a function, the value is not changed in the argument after the function finishes.
18. If you change the value of a pass-by-reference argument in a function, the value is changed in the argument after the function finishes.
19. A constant reference parameter is specified using the const keyword to tell the compiler that its value cannot be changed in the function.
20. The scope of a variable is the part of the program where the variable can be used.
21. Global variables are declared outside functions and are accessible to all functions in their scope.
22. Local variables are defined inside a function. After a function completes its execution, all of its local variables are destroyed.
23. Local variables are also called automatic variables.
24. Static local variables can be defined to retain the local variables for use by the next function call.
25. C++ provides inline functions to avoid function calls for fast execution.
26. Inline functions are not called; rather, the compiler copies the function code in line at the point of each invocation.
27. To specify an inline function, precede the function declaration with the inline keyword.
28. C++ allows you to declare functions with default argument values for pass-by-value parameters.
29. The default values are passed to the parameters when a function is invoked without the arguments.
30. Function abstraction is achieved by separating the use of a function from its implementation.
31. Programs written as collections of concise functions are easier to write, debug, maintain, and modify than would otherwise be the case.
32. When implementing a large program, use the top-down or bottom-up coding approach.
33. Do not write the entire program at once. This approach seems to take more time for coding

(because you are repeatedly compiling and running the program), but it actually saves time and facilitates debugging.

34.
$$\text{futureInvestmentValue} = \text{investmentAmount} \times (1 + \text{monthlyInterestRate})^{numberOfYears*12}$$

35.