Functions

Functions

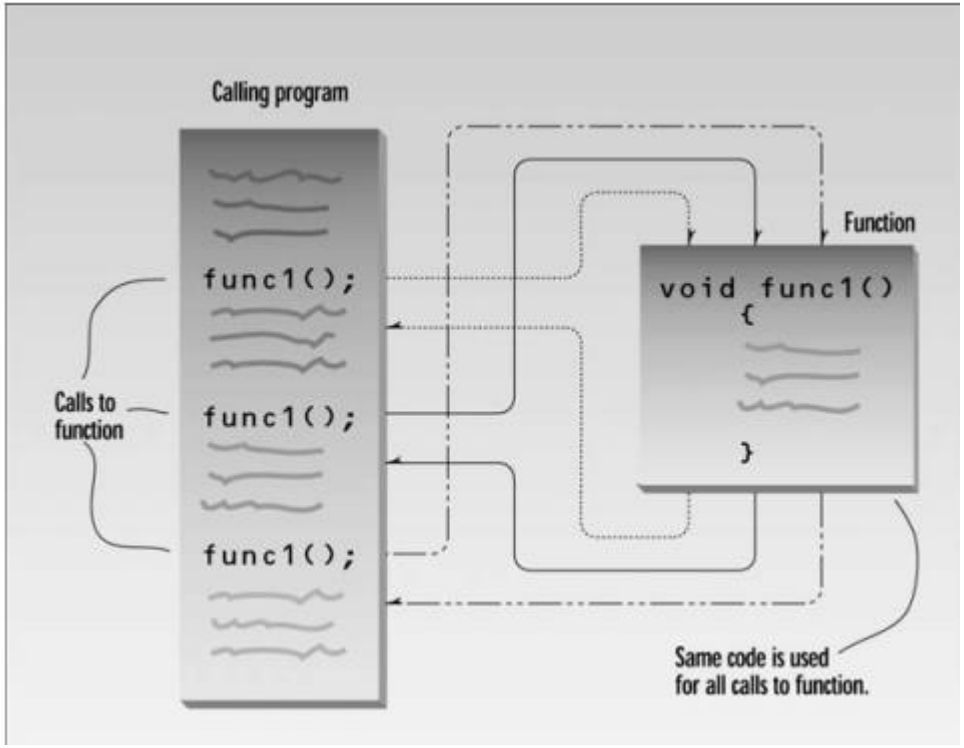
- A function groups a number of program statements into a unit.
- Function allows conceptual organization of a program (Dividing a program into functions).
- Functions are used to reduce program size.
  - Any sequence of instructions that appears in a program more than once is a candidate for being made into a function.

Benefits of functions

- Easy to read
- Easy to modify
- Avoid rewriting of same code
- Easy to debug
- Better memory utilization

Functions

- The function's code is stored in only one place in memory, even though the function is executed (called) many times in the program.



Functions : Example

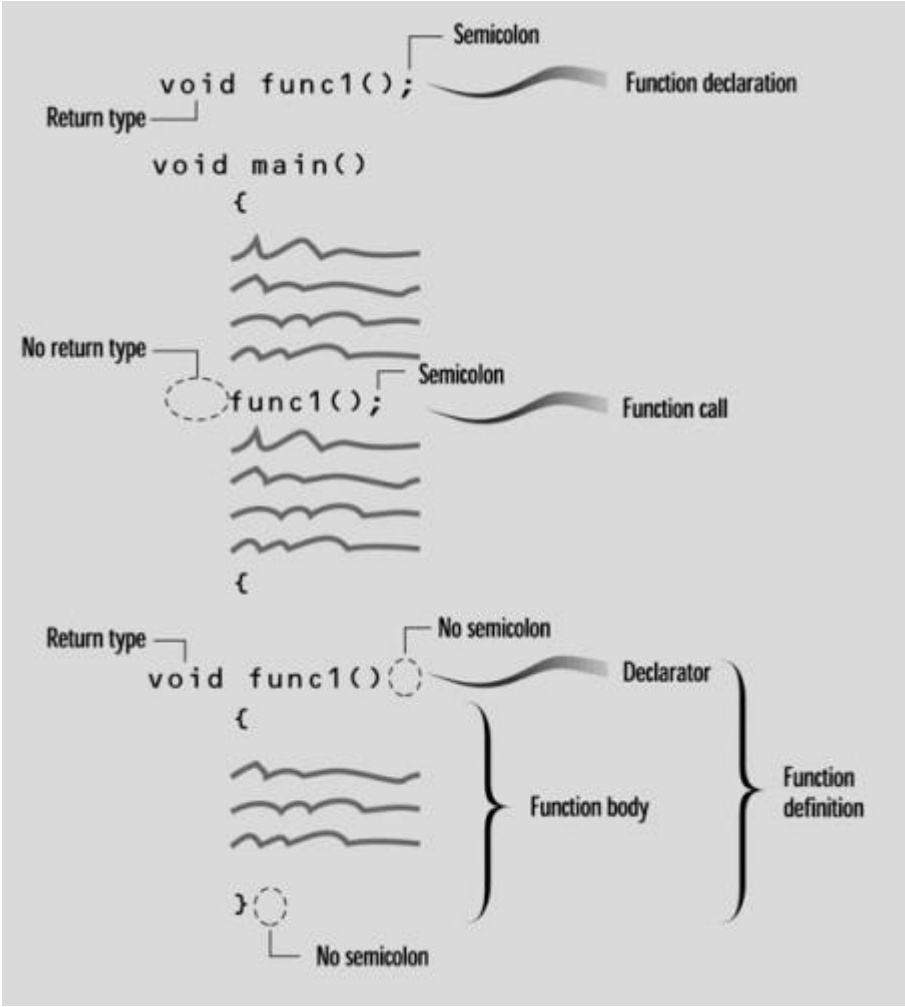
```
#include <iostream>
using namespace std;
int max (int , int );
int main() {
    int a = 10, b = 20;
    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);
    cout << "m is " << m;
    return 0;
}
int max(int x, int y)
{
    if (x > y)
    return x;
    else
    return y;
}
```

Functions

- There are two approaches to tell compiler about the functions
  - Declare the function before it is called.
    - The declaration tells the compiler that at some later point we plan to present a function.
    - Function declarations are also called prototypes.
  - The other approach is to define it before it's called.

Functions

- The information in the declaration (the return type and the number and types of any arguments) is referred to as the function signature.



Functions

Component	*Purpose*	*Example*
Declaration (prototype)	Specifies function name, argument types, and return value. Alerts compiler (and programmer) that a function is coming up later.	`void func();`
Call	Causes the function to be executed.	`func();`

Definition	The function itself. Contains the lines of code that constitute the function.	`void func()`
		`{`
		`// lines of code`
		`}`

Declarator	First line of definition.	`void func()`

Library Functions

- We have used (called) library functions in our program.
  - For ex. getche()
  - When we use a library function we don't need to write the declaration or definition
- Where are the declaration and definition for this library function?
  - The declaration is in the header file specified at the beginning of the program (CONIO.H, for getche()).
  - The definition is in a library file that's linked automatically to your program when you build it.

Formal and Actual Arguments

```
#include <iostream>
using namespace std;
int max (int , int );
int main() {
      int a = 10, b = 20;
      // Calling above function to find max of 'a' and 'b'
      int m = max(a, b);
      cout << "m is " << m;
      return 0;
}
int max(int x, int y)
{
      if (x > y)
      return x;
      else
      return y;
}
```

Actual arguments

Formal arguments

Types of Formal Arguments

- Formal arguments can be of three types:
  - Ordinary variables of any type
  - Pointer variables
  - Reference variables

Call by value

When formal arguments are ordinary variables, it is function call by value

```
#include <iostream>
using namespace std;
int max (int , int );
int main() {
      int a = 10, b = 20;
      // Calling above function to find max of 'a' and 'b'
      int m = max(a, b);
      cout << "m is " << m;
      return 0;
}
int max (int x, int y)
{
      if (x > y)
      return x;
      else
      return y;
}
```

Call by Address

When formal arguments are pointer variables, it is function call by address

```
#include <iostream>
using namespace std;
int sum (int* , int* );
int main() {
    int a = 10, b = 20;
    // Calling above function to find max of 'a' and 'b'
    int m = sum(&a, &b);
    cout << "m is " << m;
    return 0;
}
int sum (int *x, int *y)
{
    return (*x + *y);
}
```

Call by Reference

When formal arguments are reference variables, it is function call by reference

```
#include <iostream>
using namespace std;
int sum (int& , int& );
int main() {
    int a = 10, b = 20;
    // Calling above function to find max of 'a' and 'b'
    int m = sum(a, b);
    cout << "m is " << m;
    return 0;
}
int sum (int &x, int &y)
{
    return (x + y);
}
```

Reference variable

- When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting ‘&’ in the declaration.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

Reference variable

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

int main()
{
int x = 10;

// ref is a reference to x.
int& ref = x;

// Value of x is now changed to 20
ref = 20;
cout << "x = " << x << endl ;

// Value of x is now changed to 30
x = 30;
cout << "ref = " << ref << endl ;

return 0;
}
```

Contd..

- Modify the passed parameters in a function

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

void swap (int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

int main()
{
    int a = 2, b = 3;
    swap( a, b );
    cout << a << " " << b;
    return 0;
}
```

Find Output

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

int fun(int &x)
{
        return x;
}
int main()
{
        cout << fun(10);
        return 0;
}
```

- What is the return value of f(p, p) if the value of p is initialized to 5 before the call?

```
int f(int &x, int c)
 {
   c  = c - 1;
   if (c == 0) return 1;
   x = x + 1;
   return f(x, c) * x;
}
```

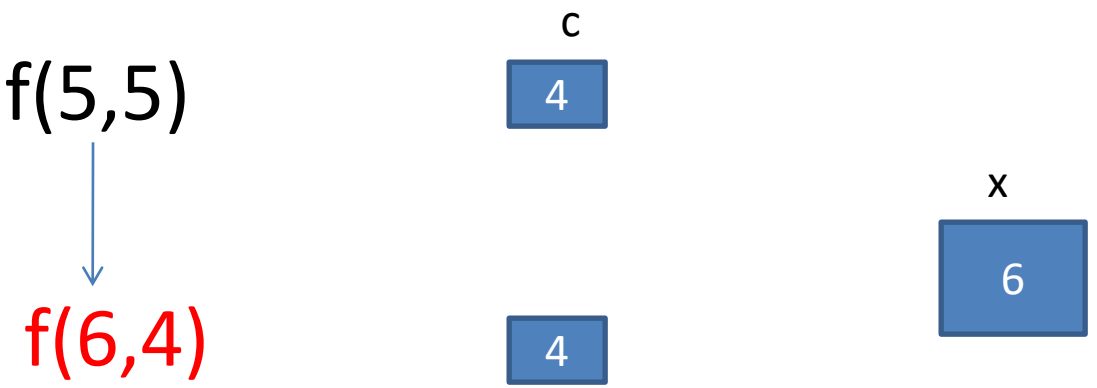
Solution

f(5,5)

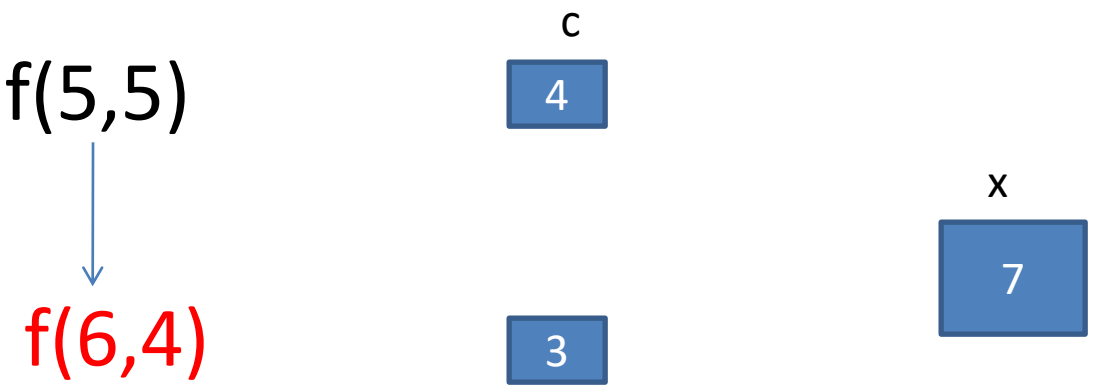
Solution

f(5,5)

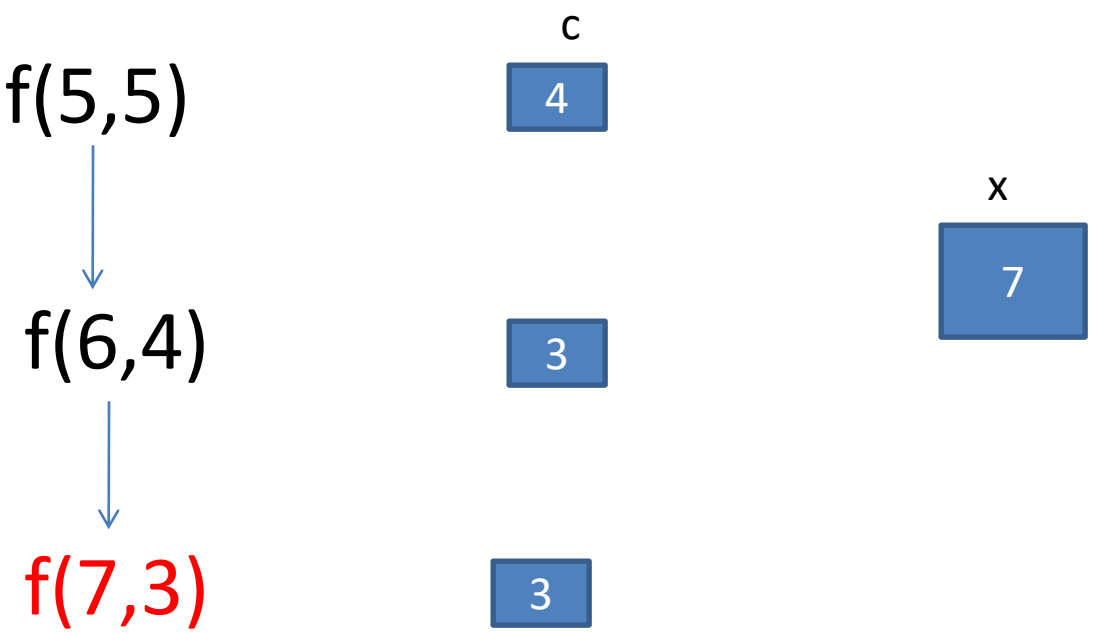
Solution



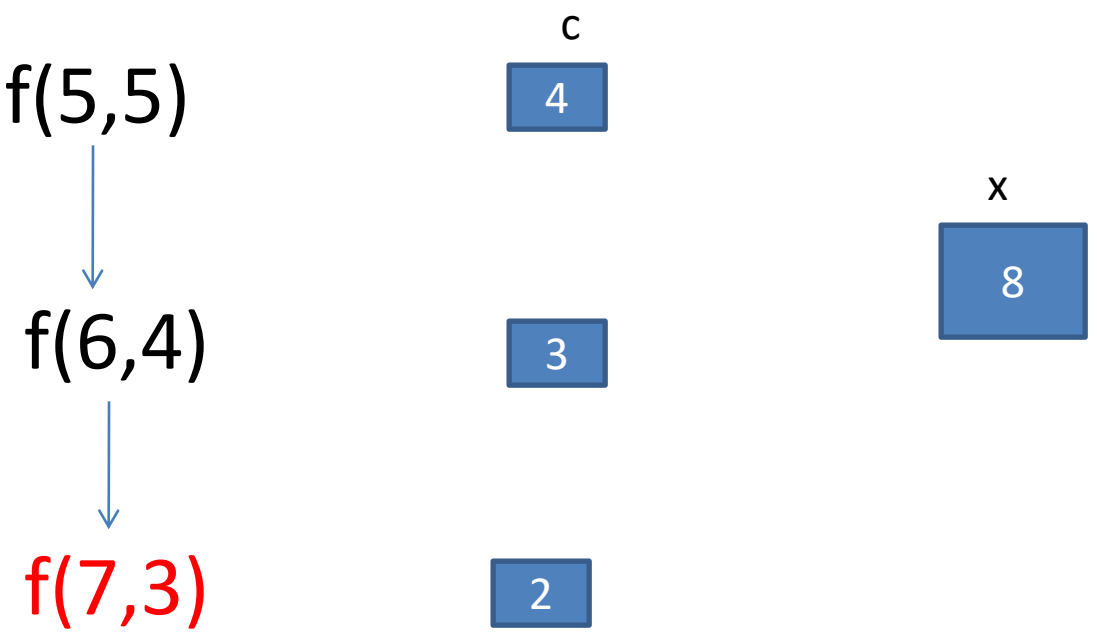
Solution

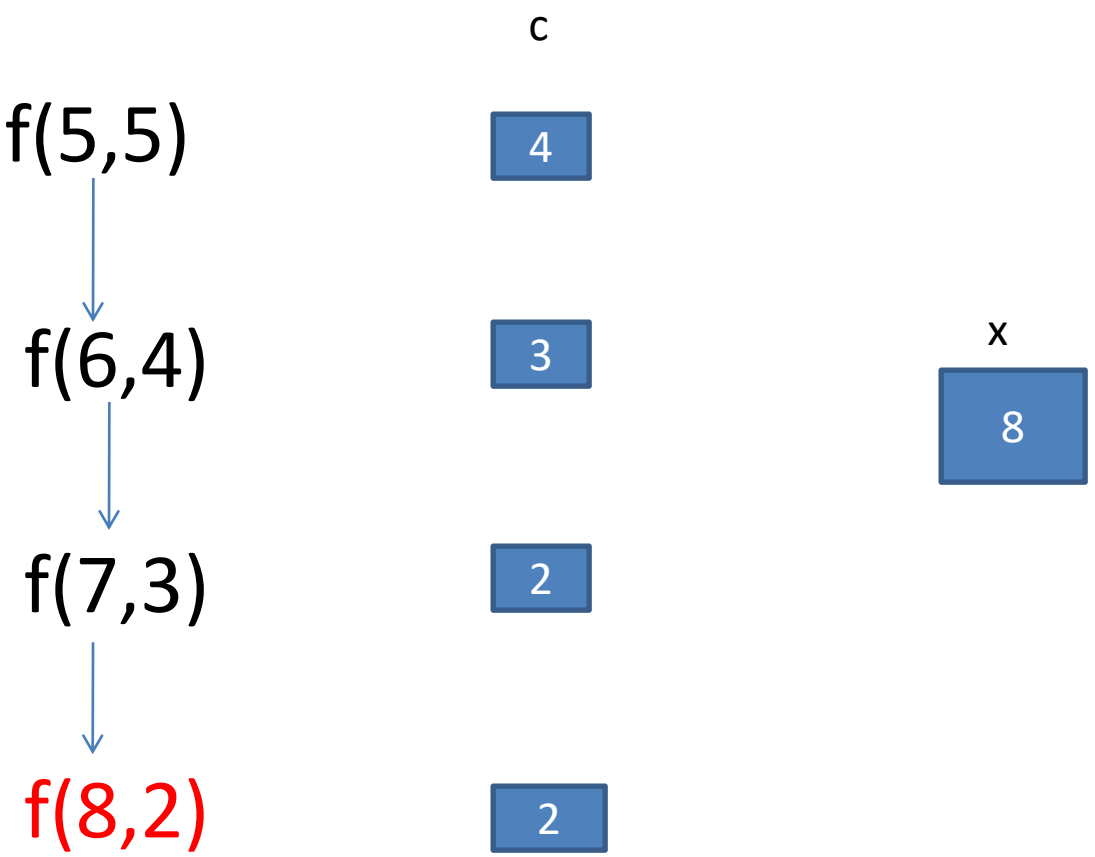


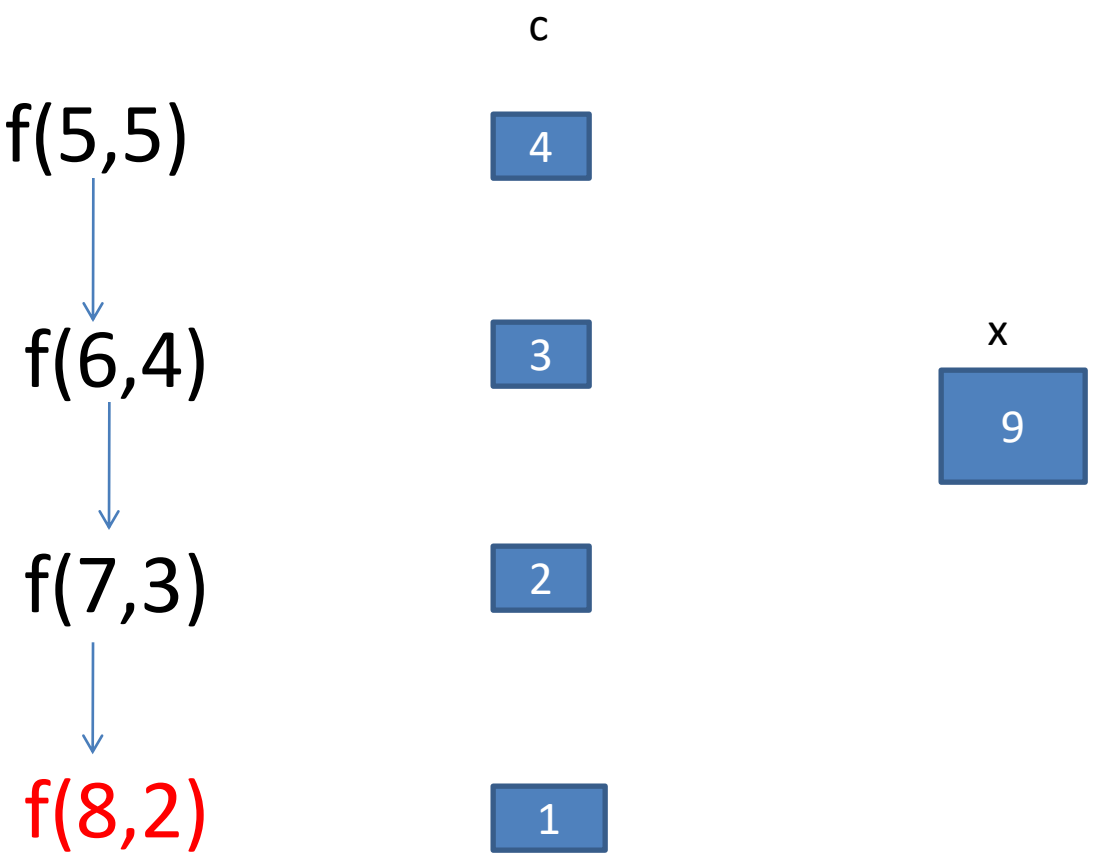
Solution

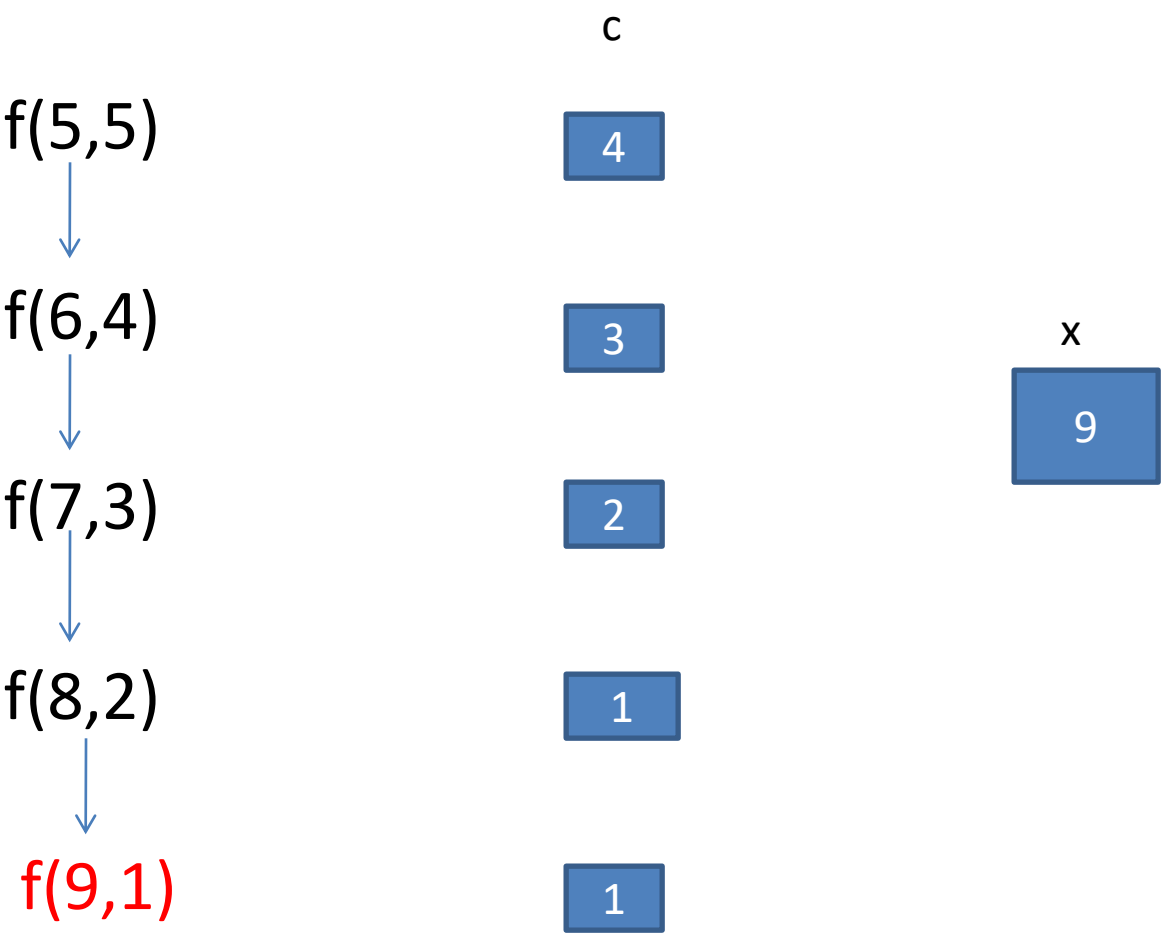


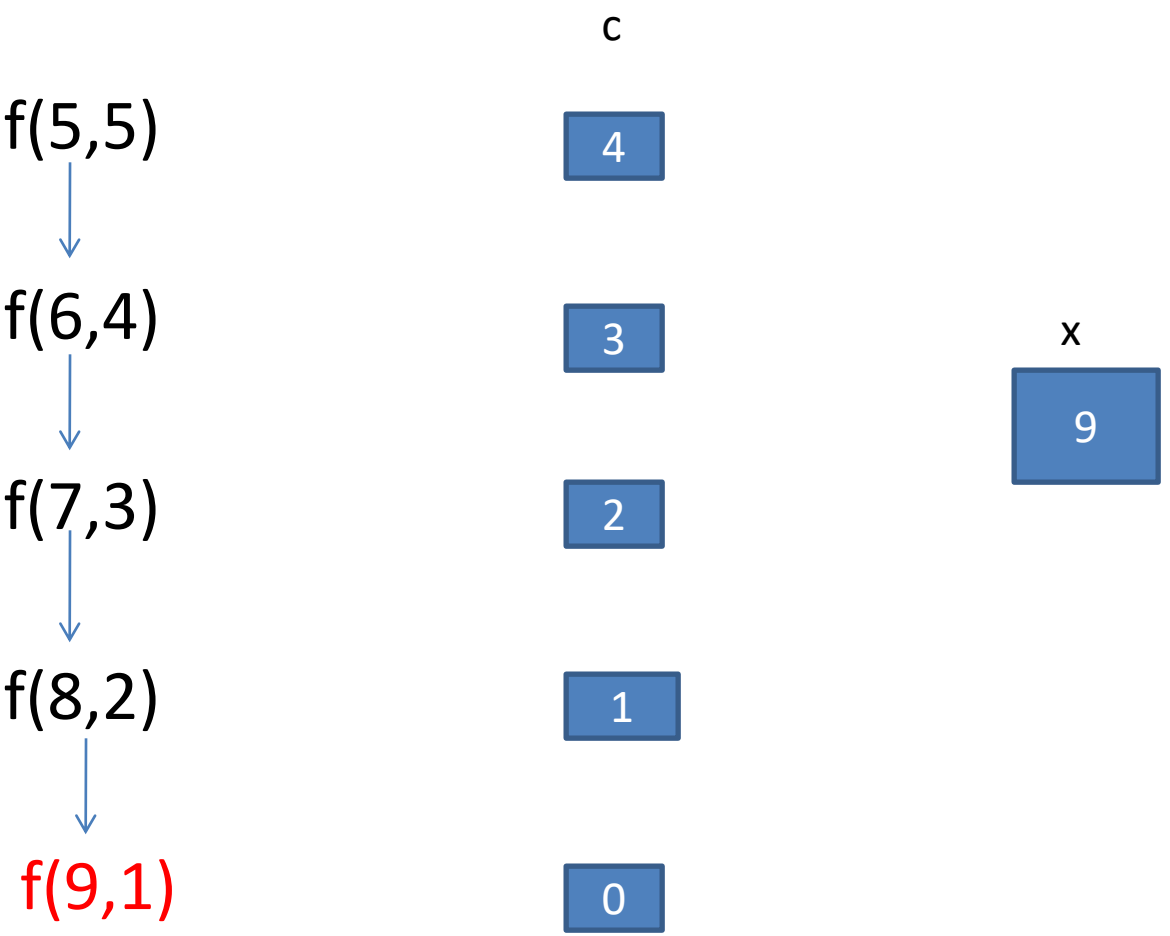
Solution

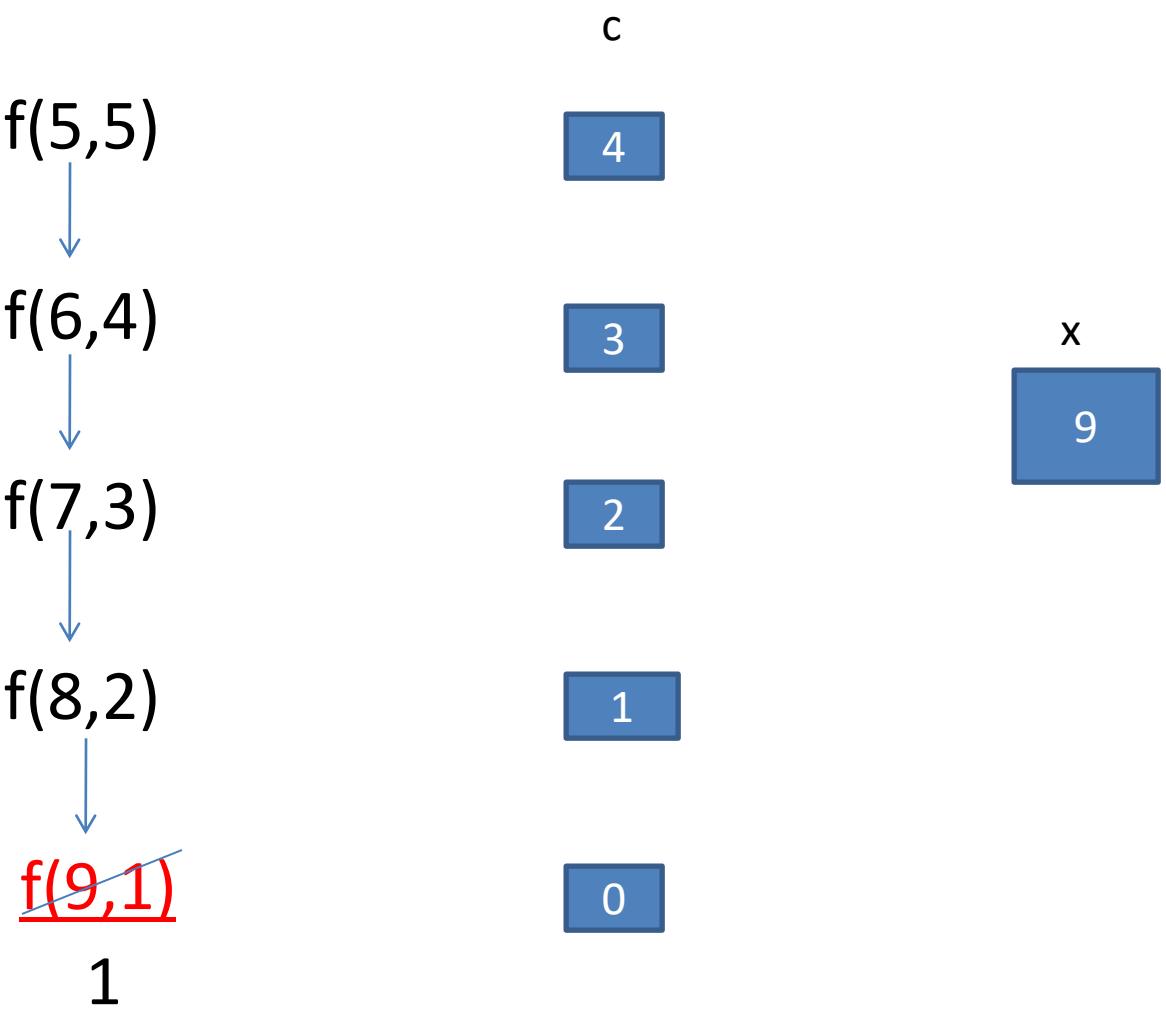


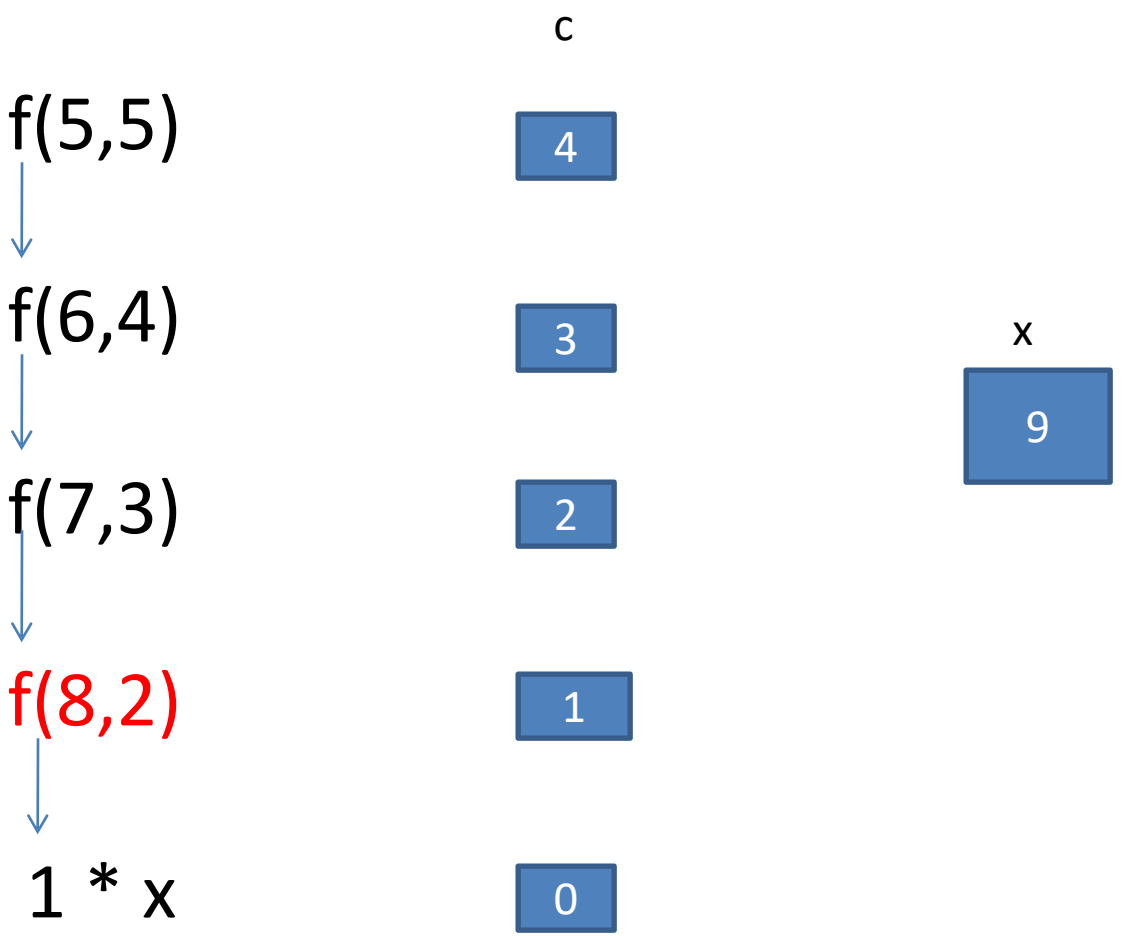

c
f(5,5)
4
f(6,4)
3
x
8
f(7,3)
2
f(8,2)
2

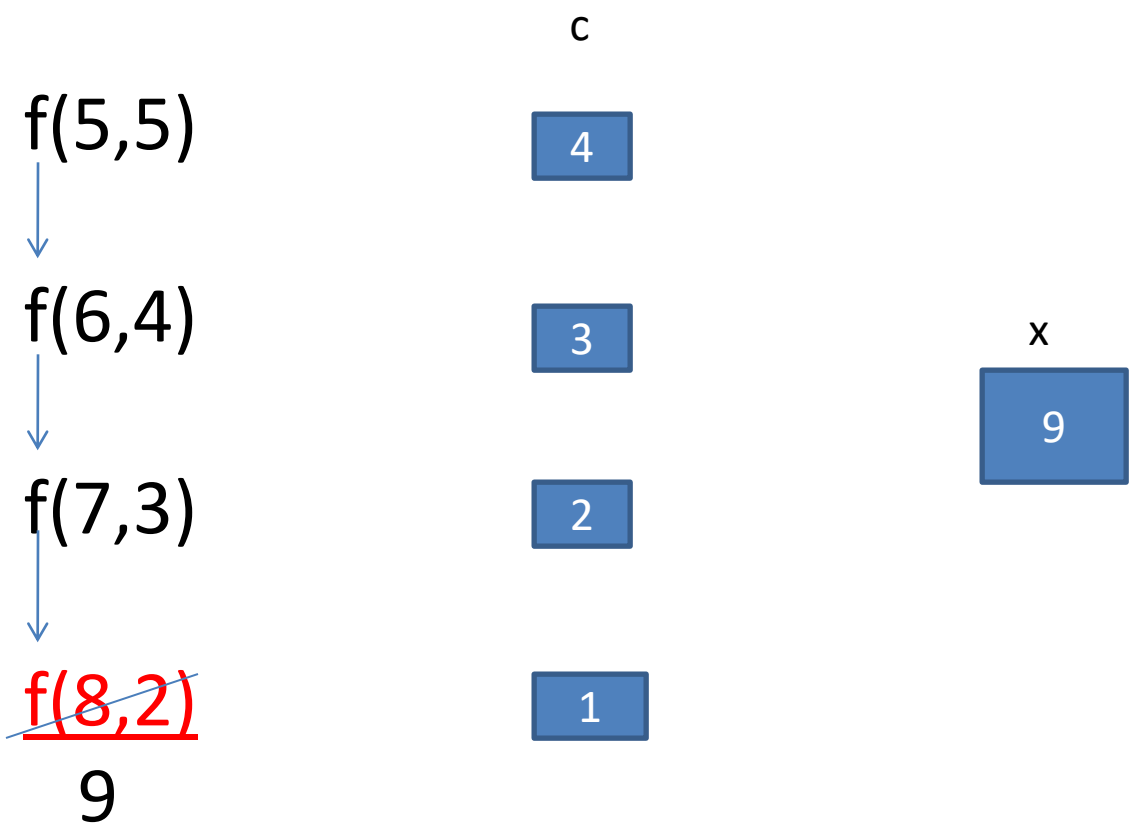

c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
f(8,2)
1

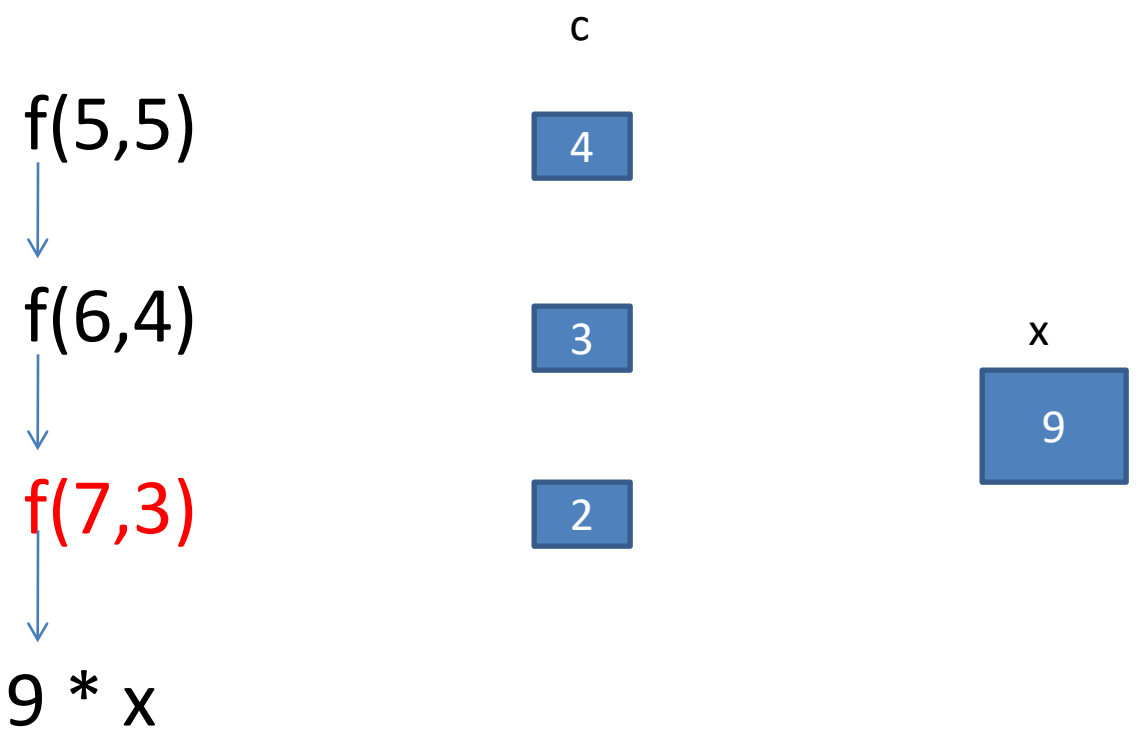

c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
f(8,2)
1
f(9,1)
1

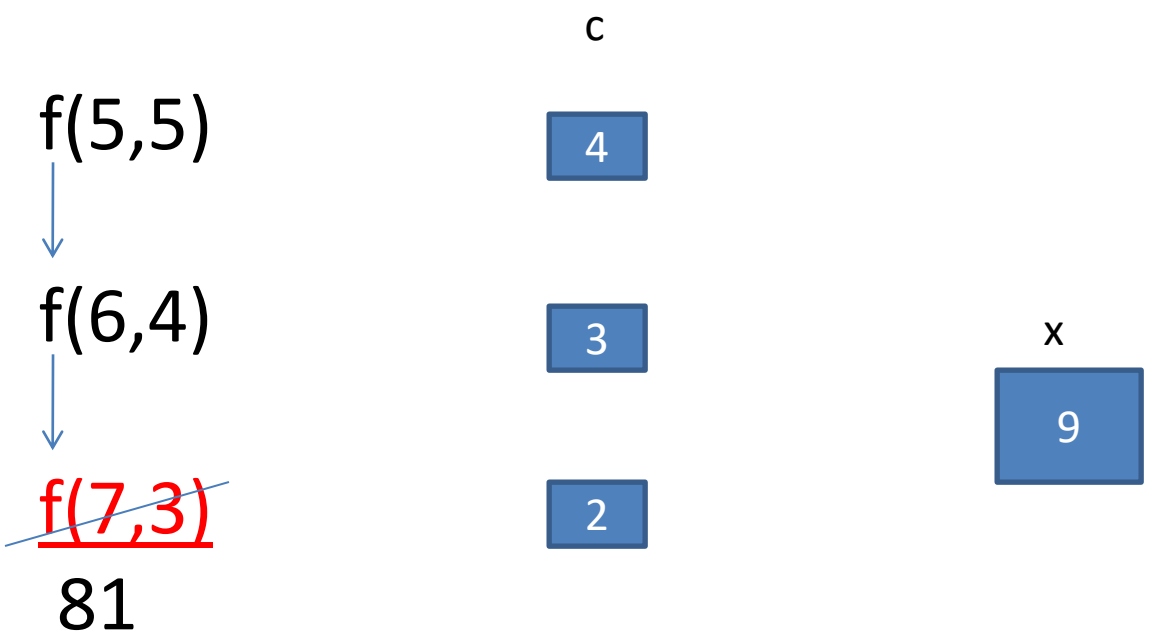

c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
f(8,2)
1
f(9,1)
0

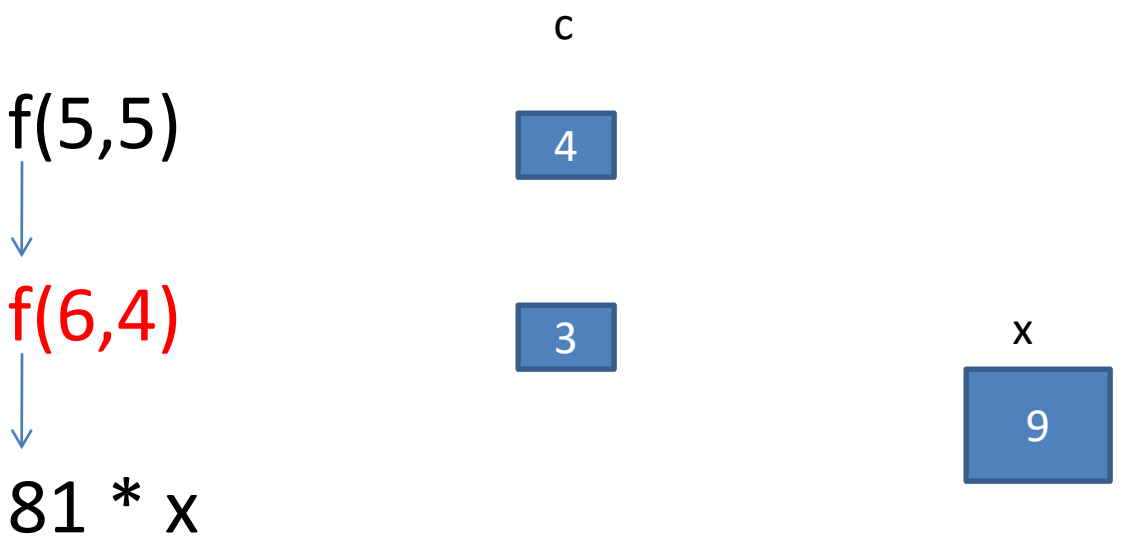

c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
f(8,2)
1
f(9,1)
0
1

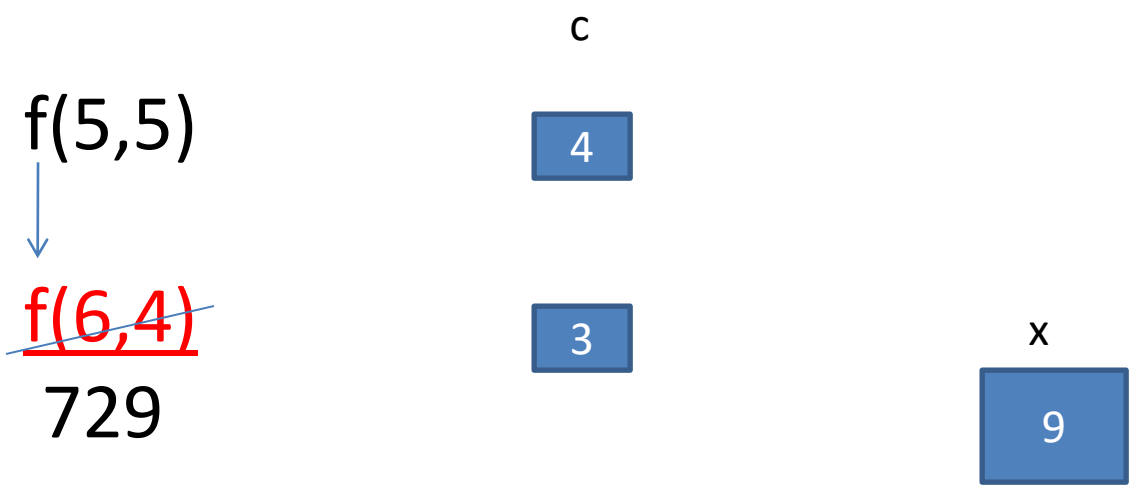

c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
f(8,2)
1
1 * x
0

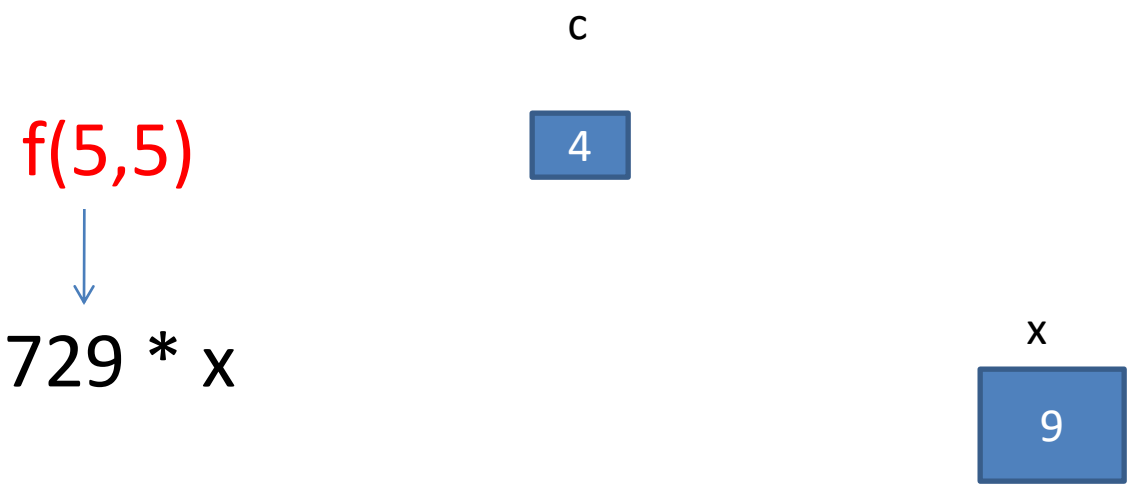

c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
f(8,2)
1
9


c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
9 * x


c
f(5,5)
4
f(6,4)
3
x
9
f(7,3)
2
81


c
f(5,5)
4
f(6,4)
3
x
9
81 * x


c
f(5,5)
4
f(6,4)
3
729
x
9


c
f(5,5)
4
729 * x
x
9

c

f(5,5)

6561

Find output

```
void fun(int,int);
int i ;
int main ()
{
    int j = 60;
    i = 50;
    fun (i, j);
    cout<< i<< j;
}
void fun (int x,int y)
{
    i = 100;
    x = 10;
    y = y + i ;
}
```

Find output

```
void fun(int&,int&);
int i ;
int main ()
{
    int j = 60;
    i = 50;
    fun (i, j);
    cout<< i<< j;
}
void fun (int& x,int& y)
{
    i = 100;
    x = 10;
    y = y + i ;
}
```

- Structures can be passed as arguments to functions.

Structure as Argument of Function

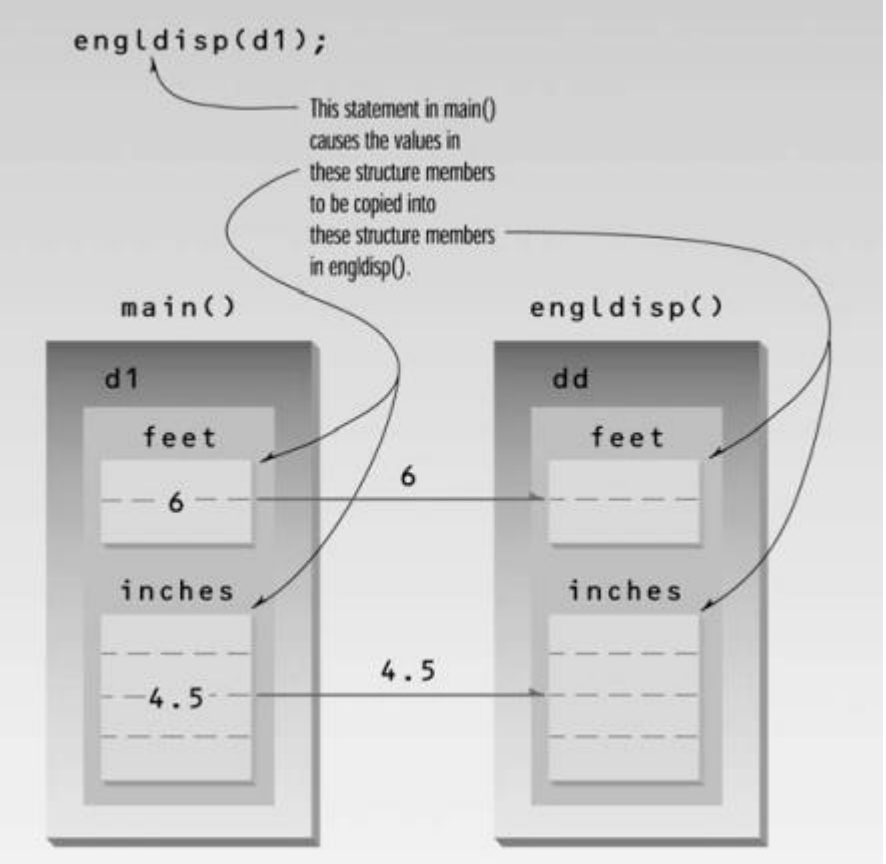
```
struct Distance     {          //English distance
      int feet;
      float inches;
};

void engldisp( Distance );    //declaration

int main()     {
      Distance d1, d2;          //define two lengths
      //get length d1 from user
      cout << "Enter feet: ";  cin >> d1.feet;
      cout << "Enter inches: ";  cin >> d1.inches;
      cout << "\nEnter feet: ";  cin >> d2.feet;
      cout << "Enter inches: ";  cin >> d2.inches;
      cout << "\n d1 = ";
      engldisp(d1);             //display length 1
      cout << "\n d2 = ";
      engldisp(d2);             //display length 2
      cout << endl;
      return 0;
}
```

```
// display structure of type Distance in feet and inches
void engldisp( Distance dd )      {

                  cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

Structure as Argument of Function

The function engldisp() uses a parameter that is a structure of type Distance, which it names dd.

This structure variable (dd) is automatically initialized to the value of the structure passed from main().



Return Values of Function

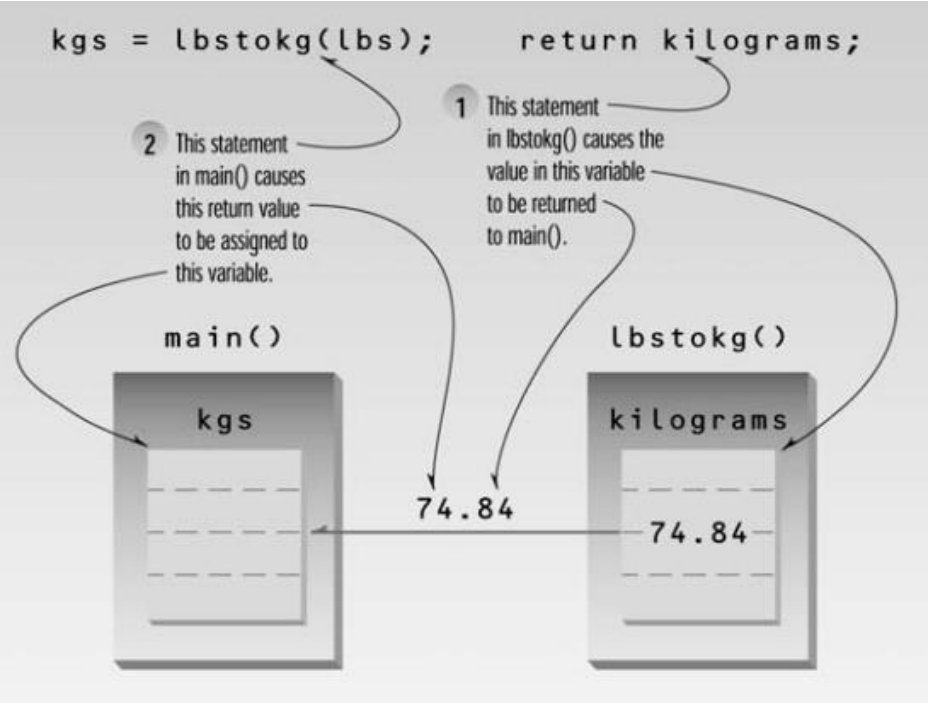
- When a function completes its execution, it can return a single value to the calling program.
- When a function returns a value, the data type of return value must be specified.
- The function declaration does this by placing the data type, before the function name in the declaration and the definition.

Returning Values from Function

```
float lbstokg(float);   //declaration

int main()
{
      float lbs, kgs;
      cout << "\nEnter your weight in pounds: ";
      cin >> lbs;
      kgs = lbstokg(lbs);
      cout << "Your weight in kilograms is " << kgs << endl;
      return 0;
}

// lbstokg()   :   converts pounds to kilograms
float lbstokg(float pounds)      {
      float kilograms =  0.453592 * pounds;
      return kilograms;
}
```

Return Value of Function

When the function returns, the value in kilograms variable is
copied into kgs variable.



Overloaded Functions

- An overloaded function appears to perform different activities depending on the kind of data sent to it.
- It is difficult for programmers to remember names of functions providing similar functionality .
  - For ex. starline(), repchar(), and charline() – functions perform similar activity, printing character.

- It will be more convenient to use the same name for all three functions, even though they each have different arguments.
  - void repchar();
  - void repchar(char);
  - void repchar(char, int);

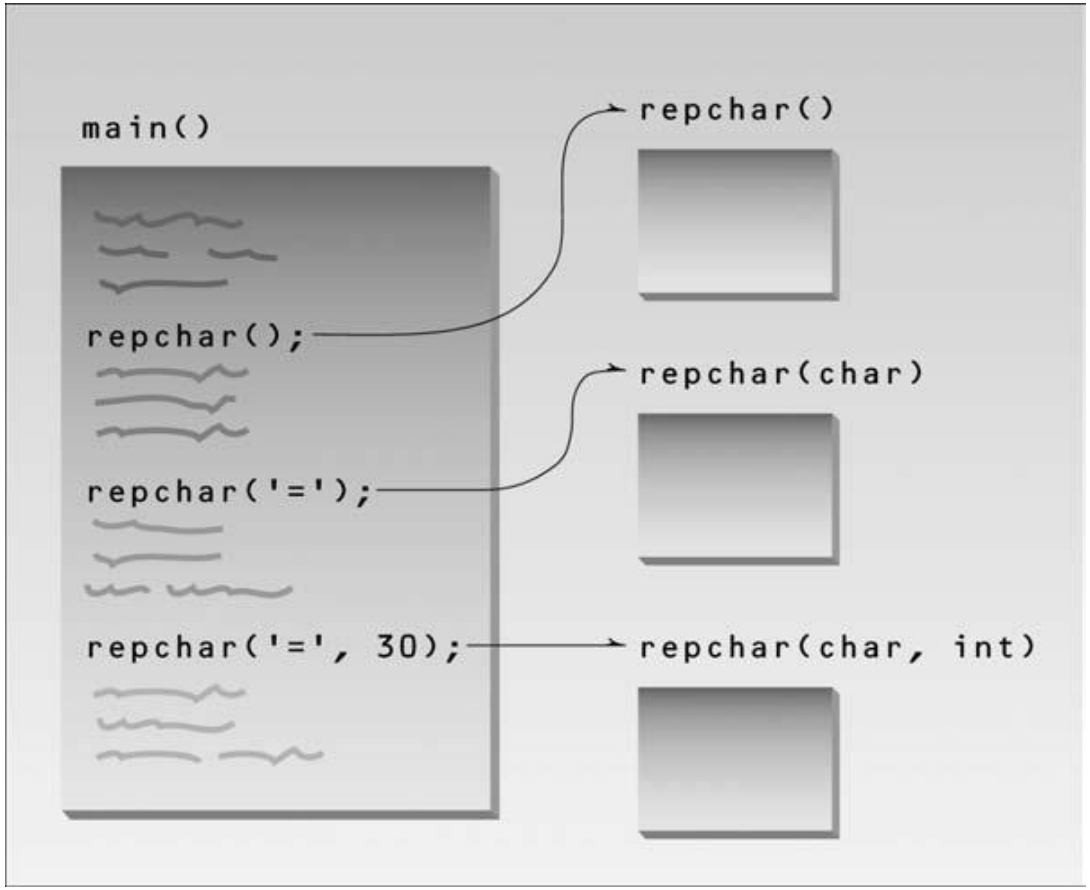
Overloaded Functions

```
void repchar();          //declarations
void repchar(char);
void repchar(char, int);

int main()    {
      repchar();
      repchar('=');
      repchar('+', 30);
      return 0;
}

// repchar() : displays 45 asterisks
void repchar()    {
      for(int j=0; j<45; j++)  // always loops 45 times
            cout << '*';       // always prints asterisk
      cout << endl;
}
```

```
// displays 45 copies of specified character

void repchar(char ch)    {
            for(int j=0; j<45; j++)
                  cout << ch;
            cout << endl;
}

/* repchar() : displays specified number of
copies of specified character */

void repchar(char ch, int n)    {
            for(int j=0; j<n; j++)
                  cout << ch;
            cout << endl;
}
```

Overloaded Functions

What keeps the compiler from becoming confused?
It uses the function signature—the number of arguments, and their data types—to distinguish one function from another



How function overloading is resolved?

- First c++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions.
- If no exact match is found, c++ tries to find match through promotion.
  - Char and short is promoted to int
  - Float is promoted to double
- If no promotion is found, c++ tries to find a match through standard conversion.

Inline Functions

- Functions save memory space because call to function cause the same code to be executed
- When a function is called, jump to function is made. After the call, jump back to instruction following the call takes some extra time for jump to function
  - Slows down the program
- To save execution time for short functions (having one or two statements)
  - Make them inline

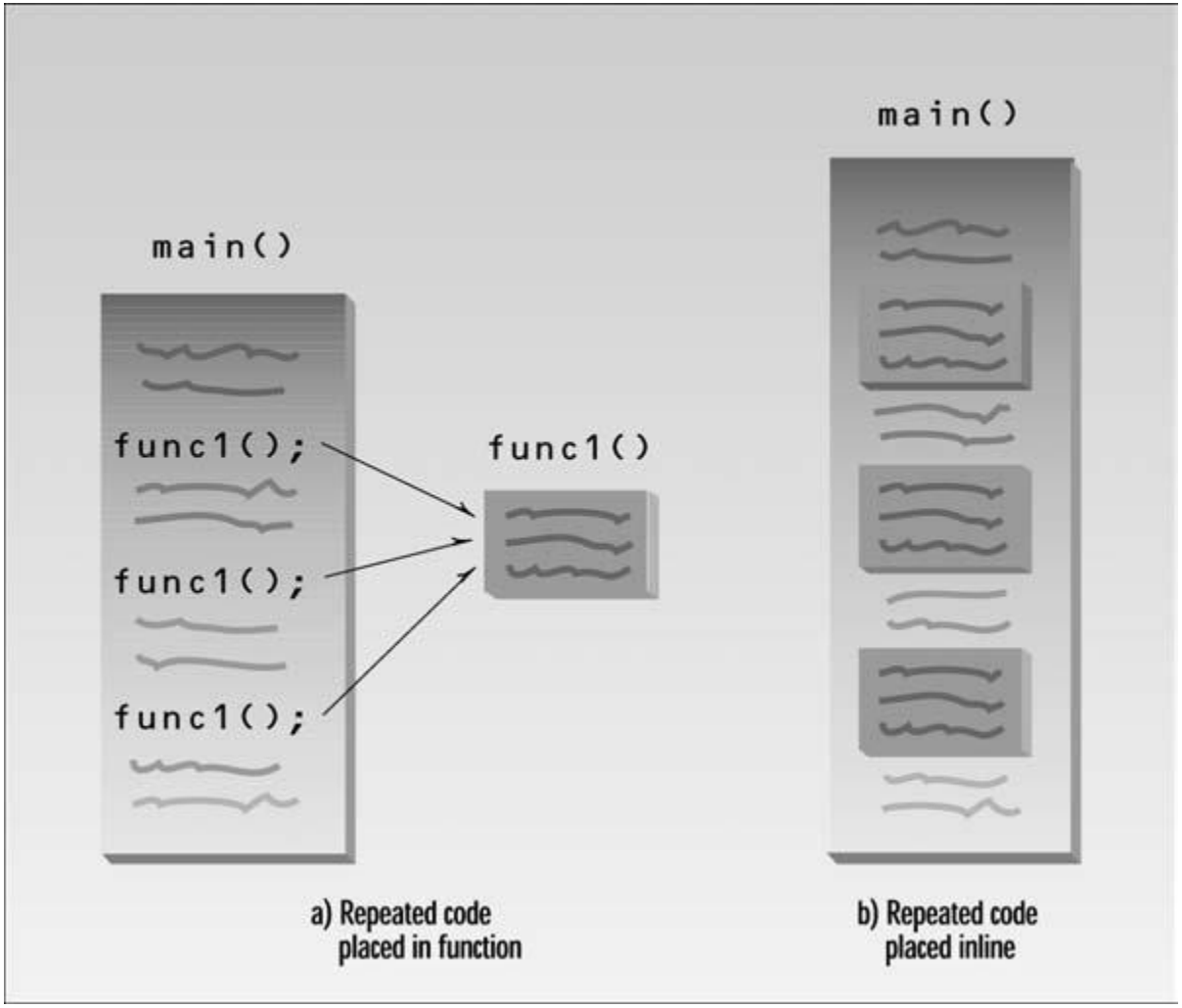
Contd..

- To eliminate the cost of calls to small functions, c++ proposes a new feature called inline function.
- An inline function is a function that is expanded in line when it is invoked.
- Compiler replaces the function call with its corresponding function code.

Inline is a request

- Inline is a request, not a command.
- The benefit of speed of inline function reduces as function grows in size.
- So the compiler may ignore the request in some situations:
  - Function containing loop, switch, goto
  - Functions with recursion
  - Function containing static variable

Inline Functions



Inline functions: Example

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

inline float lbstokg(float pounds) // converts pounds to kilograms
{
    return 0.453592 * pounds;
}

int main()    {
    float lbs;
    cou    t << "\nEnter your weight in pounds: ";
    cin >> lbs;
    cout << "Your weight in kilograms is " << lbstokg(lbs) << endl;
    return 0;
}
```

Inline functions: Example

- inline float lbstokg(float pounds) // inline function

- Sometimes the compiler will ignore the request and compile the function as a normal function.
- It might decide the function is too long to be inline, for instance.

Default Arguments

- 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.

Example

```
#include <iostream>
using namespace std;
void repchar(char='*', int=45);    // default arguments
int main()
{
  repchar();                 // prints 45 asterisks
  repchar('=');              // prints 45 equal signs
  repchar('+', 30);          // prints 30 plus signs
  return 0;
}

void repchar(char ch, int n)
{
  for(int j=0; j<n; j++)
     cout << ch;
  cout << endl;
}
```

Invalid Statement

```
// Invalid because z has default value, but w
   //after it  doesn't have default value
  int sum(int x, int y, int z=0, int w)
```

Scope and Storage Class

- Scope:
  - The scope of a variable determines which parts of the program can access it.

- Storage Class:
  - Storage class of a variable determines how long it stays in existence in program.

Local Variables

- Variables defined within a function body are called local variables because they have local scope.
- They are also sometimes called Automatic variables.

```
void somefunc()
{
   int a;       //variables defined within
   float b;     //the function body
// other statements
}
```

- Lifetime:
  - The time period between the creation and destruction of a variable is called its lifetime.
- Scope:
  - A variable's scope, also called visibility, describes the locations within a program from which it can be accessed.

- Lifetime and scope of local variable:
  - The lifetime of a local variable coincides with the time when the function in which it is defined is executing.
  - The scope of a local variable is within the function in which it is defined.

```
Int func1()
{
    int a,b;
    a=10;                // ok
    b=20;                //ok
    c=30;                // illegal
}
Int func2()
{
    int c;
    a=5;                // illegal
    b=4;                // illegal
    c=10;               //ok
}
```

Global Variables

- Variables defined outside of any function are called global variables.
- A global variable is visible to all the functions in a file.
- Global variables are also sometimes called external variables, since they are defined external to any function.

```
#include <iostream>
using namespace std;
#include <conio.h>
char ch = ‘a’;                    //global variable ch
void getachar();
void putachar();
int main()
{
      while( ch != ‘\r’ )
      {
            getachar();
            putachar();
      }
      cout << endl;
      return 0;
}

void getachar()
{
      ch = getch();
}

void putachar()
{
      cout << ch;
}
```

- Lifetime and scope of global variables:
  - Lifetime of global variable is same as life of the program.
  - Global variables are visible in the file in which they are defined, starting at the point where they are defined.

Static local variables

- Static local variables are used when it’s necessary for a function to remember a value when it is not being executed

- Lifetime and scope of static local variables:
  - Its lifetime is the same as that of a global variable (that is, for the life of program).
  - A static local variable has the visibility of an automatic local variable (that is, inside the function containing it).

```
#include <iostream>
using namespace std;
float getavg(float);
int main()
{
	float data=1, avg;
	while( data != 0 )
	{
		cout << “Enter a number: “;
		cin >> data;
		avg = getavg(data);
		cout << “New average is “ << avg << endl;
	}
	return 0;
}

float getavg(float newdata)
{
	static float total = 0;
	static int count = 0;
	count++;
	total += newdata;
	return total / count;
}
```

- Enter a number: 10
- New average is 10 --- total is 10, count is 1
- Enter a number: 20
- New average is 15 --- total is 30, count is 2
- Enter a number: 30
- New average is 20 --- total is 60, count is 3

```
#include<iostream>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  cout<<fun();
  cout<<fun();
  return 0;
}
```

Return by Reference

```
#include <iostream>
using namespace std;
int x;          // global variable
int& setx();
int main()
{
    setx() = 92;        // function call on left side
    cout << “x=” << x << endl;
    return 0;
}

int& setx()
{
    return x;
}
```

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

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues( int i ) {
  return vals[i];          // return a reference to the ith element
}

int main ()
{
  cout << "Value before change" << endl;
  for ( int i = 0; i < 5; i++ ) {
     cout << vals[i] << endl;
  }

  setValues(1) = 20.23;         // change 2nd element
  setValues(3) = 70.8;           // change 4th element

  cout << "Value after change" << endl;
  for ( int i = 0; i < 5; i++ ) {
    cout << vals[i] << endl;
  }
  return 0;
}
```

- you can’t return a constant from a function that returns by reference

```
int& setx()
{
    return 3;
 }
```

- you can’t return a reference to a local variable:

```
int& setx()
{
   int x = 3;
   return x;      // error
}
```