Computer Programming

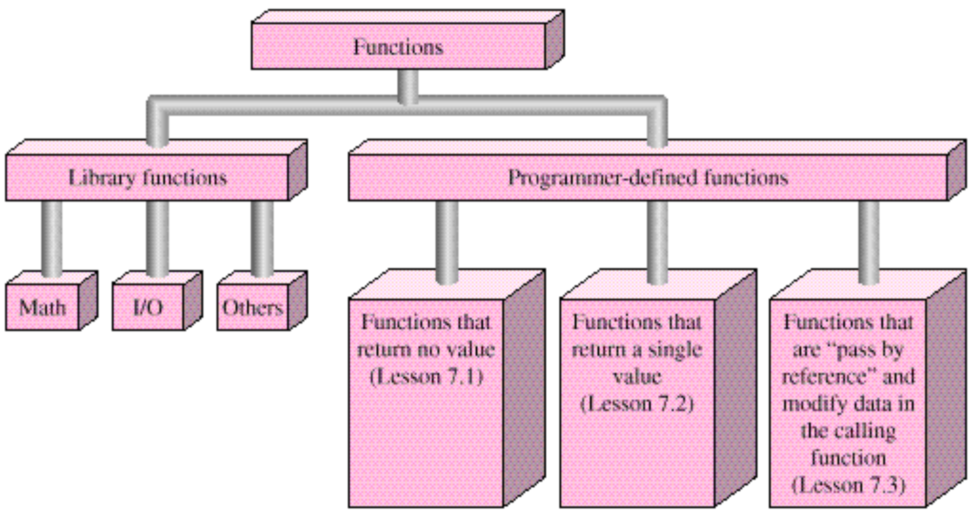
Function

Hung-Yun Hsieh
October 12, 2022

Functions

Every C++ program has at least one function – `main()`

- Functions
  - Building blocks of computer programs
  - ☞ Divide and conquer: divide a large task into small, separate parts (modules)
  - ☞ Well-written functions can be reused in different programs and can help program maintenance
  - ☞ It is important to know the change of program control with the use of functions



C++ standard library functions

Program Structure

- A program with user-defined functions

goes to header → function declaration:

```
#include <iostream>
using namespace std;
double kinetic_energy(int m, double v);

int main( )
{
    int mass=15;
    double energy=0, velocity=300;
    energy = kinetic_energy(mass, velocity);
    cout<<"The value of the kinetic energy is: "<<energy<<endl;
    return 0;
}
double kinetic_energy(int m, double v)
{
    double ke;
    ke = 0.5*m*v*v;
    return ke;
}
```

goes to header

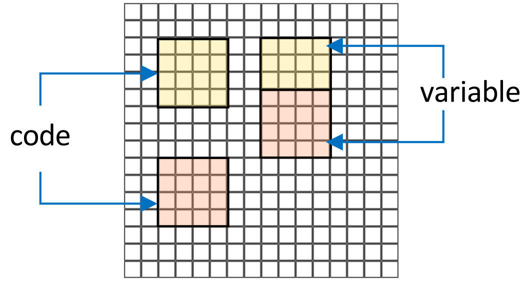
goes to library

function declaration

function invocation

function definition

Program Structure (cont.)



- A program with user-defined functions

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

double kinetic_energy(int m, double v)
{
    double ke;
    ke = 0.5*m*v*v;
    return ke;
}

int main( )
{
    int mass=15;
    double energy=0, velocity=300;
    energy = kinetic_energy(mass, velocity);
    cout<<"The value of the kinetic energy is: "<<energy<<endl;
    return 0;
}
```

Each function has its *variable space*

Values of `mass` & `velocity` are used to *initialize* variables `m` and `v`

These 3 variables are *localized* inside the function itself

m	15
v	300
ke	675000

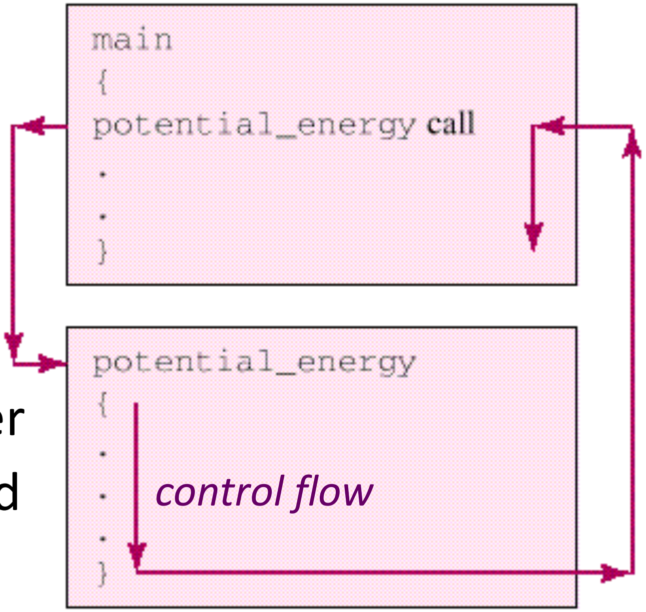
value

value

mass	15
velocity	300
energy	675000

Writing and Using Functions

- Three different parts
  - ① Function declaration (prototype)
    - Function must be declared or defined before they are used
    - Function name, return value, and argument list
  - ② Function definition
    - Function header followed by the function body enclosed in {}
  - ③ Function call (invocation)
    - A statement that causes control to transfer from one function to another
    - Data may also be passed to the called function
    - A function may return a value to the caller (result of function evaluation)



① Declaring Functions

- Function declaration

```
double energy (int mass, double velocity);
```

data type returned by the function to the caller

arguments passed from the caller to the function

  - Function name is an identifier
    - Follow the same rule for the variable name
  - Function return type is needed
    - `void`, `int`, `double`, `char`, …
  - Function argument can be empty
    - Argument name can be omitted

      ```
      double energy (int, double);
      ```

    - The whole argument list can be empty

      `double energy ();` or `double energy (void);`

② Defining Functions

- Function definition
  - The function's executable statements

Function return type. Function name. Function argument declarations.

```
void kinetic_energy(int m, double v)
{
    double ke;

    ke = 0.5 * m * v * v;

    cout << "Kinetic energy = " << ke << endl;
}
```

Function body.

  - Definition is consistent with function declaration
    - ☞ Argument "name" does not matter (only "type" matters)

☞ The `inline` qualifier can be placed before the *return type* (explained later)

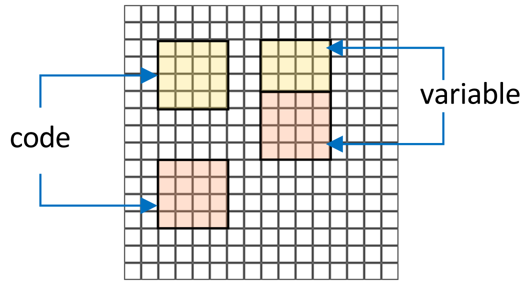
More on Function Definitions

Recall the use of the header file

- Arrangement of function definition
  - No declaration needed if functions are *defined before* they are called
    - ☞ However, declaring functions before use is still a good practice
  - Declaration is effective *from the point* of declaration
    - ☞ No nested function definition (a function defined inside another function)

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

void kinetic_energy (int m, double v)
{
     function body
}

void potential_energy ( )
{
     function body
}

int main ( )
{
     function body
}
```

③ Calling Functions



- Function call
  - Program control transfer from the caller to the called function (callee)
    - *Control returns to the caller* when the called function returns
  - Interaction between the caller and callee
    - Return type
    - Argument list

```
int a = myfunction(c, d);
```

- Function declaration, definition, and call must be consistent with each other
  - ☞ In particular, the argument list must agree in terms of the (*number*, *order*, *type*) of arguments

An Example (1/2)

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

void potential_energy();
void kinetic_energy(int, double);

int main( )
{
    int mass=15;
    double velocity=300;

    cout<<"The value of mass in main is: "<<mass<<endl;

    potential_energy();
    kinetic_energy(mass, velocity);

    cout<<"Now the value of mass in main is: "<<mass<<endl;
}
```

An Example (2/2)

Variables defined inside a function can be seen and used only by that function -- they cannot be used inside other functions (so variables inside `main()` cannot be seen by other functions)

```
void potential_energy()
{
    int mass=6;
    double pe, height=5.2;
    double g=9.81;
    pe = mass*g*height;

    cout<<"Potential energy="<<pe<<endl;
}

void kinetic_energy(int m, double v)
{
    double ke;
    ke = 0.5*m*v*v;
    cout<<"Kinetic energy="<<ke<<endl;
}
```

```
int main( )
{
    int mass=15;
    double velocity=300;

    kinetic_energy(mass, velocity);
}
```

pass-in variables that can be used in the function

local variable of function

Variables `ke`, `m` and `v` can be used *only inside* the body of the function `kinetic_energy()`

Returning Control

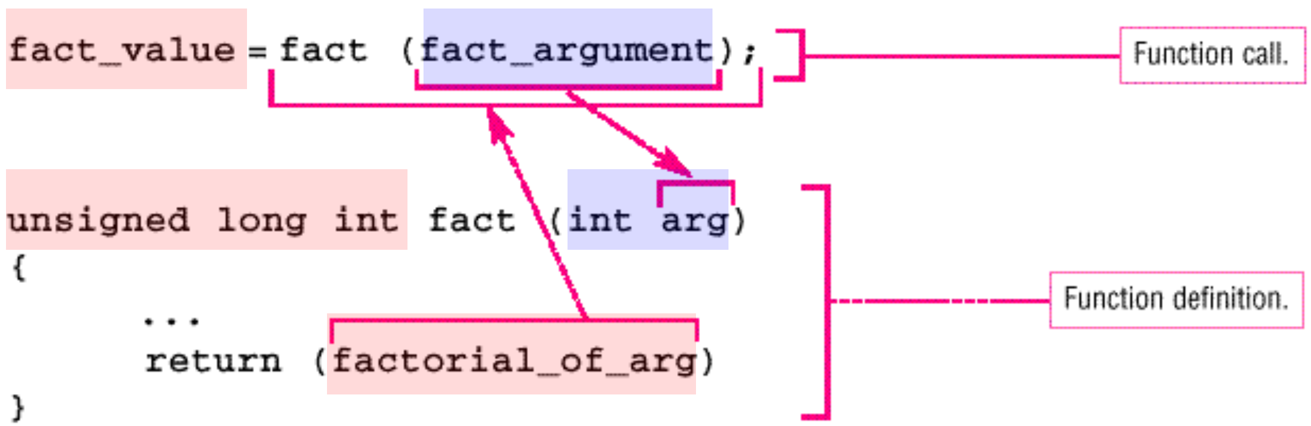
Flowing off the end of a function is equivalent to a return with no value; this results in *undefined behavior* in a value-returning function

- The return keyword
  - Return to the caller with the value indicated

```
return (expression);
```

use "`return;`" to return without any value (for function of `void` type)

  - A function can return anywhere in the function body
  - A function returns (with no value) *automatically* when the end of the function body (}) is reached



An Example with `return`

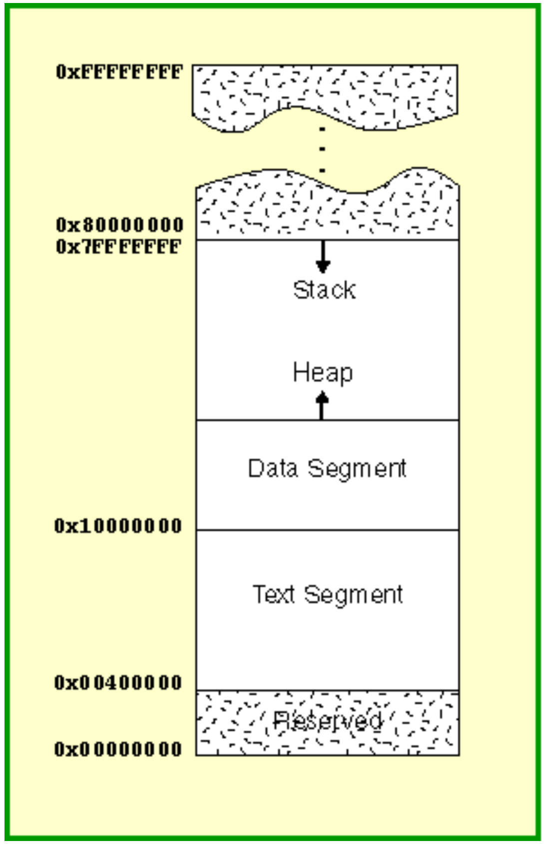
```
#include <iostream>
using namespace std;
double kinetic_energy(int, double);

int main( )
{
    int mass=15;
    double velocity=300;

    double energy = kinetic_energy(mass, velocity);
    cout<<"The value of the kinetic energy is:"<<energy<<endl;
}

double kinetic_energy(int m, double v)
{
    double ke;
    ke = 0.5*m*v*v;
    return ke;
}
```

The **value** of the variable `ke` is returned back to `main()`

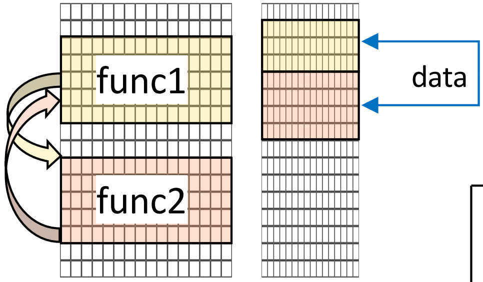
We say the **evaluation result** of `kinetic_energy()` is `ke`

Computer Programming

Function Call Stack



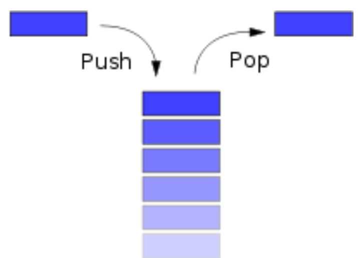
Calling Functions

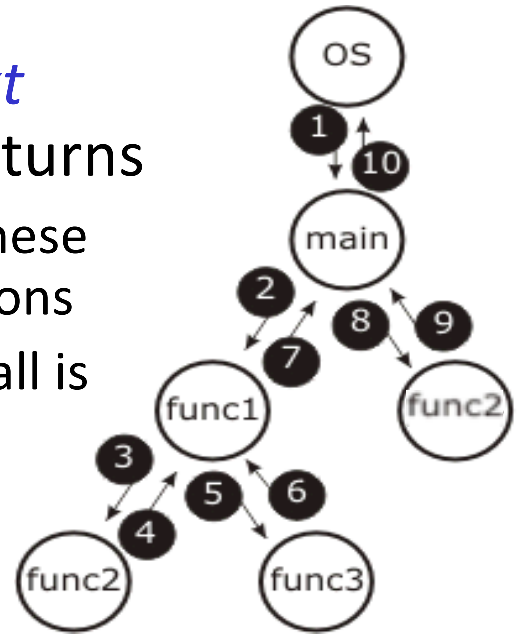


func1
```
…
func2();
cout << "…";
…
```

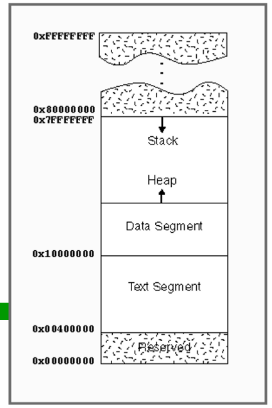
return address for function `func2()`

- Function call
  - A called function may call another function (before it returns), which may in turn call another function
    - Each function eventually must *return control* to the function that called it and *resume* execution of the following instructions in the calling function
  - Keeping track of the *address of the next instruction to run* when the function returns
    - One common place in a program to hold these return addresses for different called functions
    - The return address of the latest function call is used before other return addresses
    - Last in, first out (LIFO)
    - A stack data structure



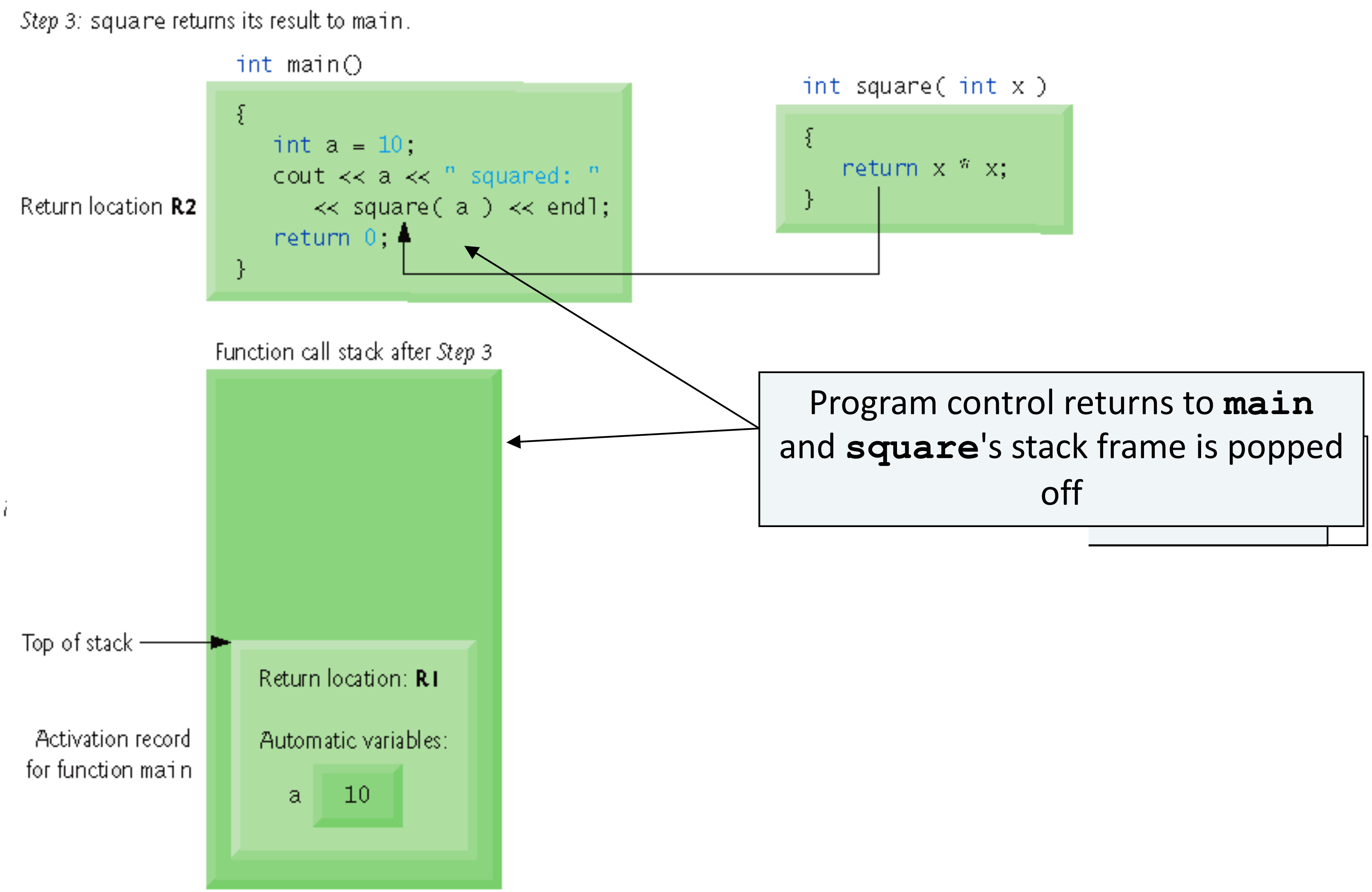


Function Call Stack



- Call stack
  - A stack data structure that stores information about the active subroutines (functions) of a program
    - All functions in a program share the same call stack
  - Each time a function calls another function, an entry (activation record / stack frame) for the called function is pushed onto the stack
    - The entry includes the return address in the caller function
    - If the function returns, the entry is popped, and control transfers back to the return address in the popped entry
  - The stack frame is also often used for storing other information for the called function including
    - Local variables of the function
    - Parameter values to be passed into the function

Call Stack Illustration



Yet Another Example

```
int main( )
{
    int x = 72, y = 73;
    cout << x << endl;
    func(72, 73);        // func(x, y);
    cout << y << endl;
}
```

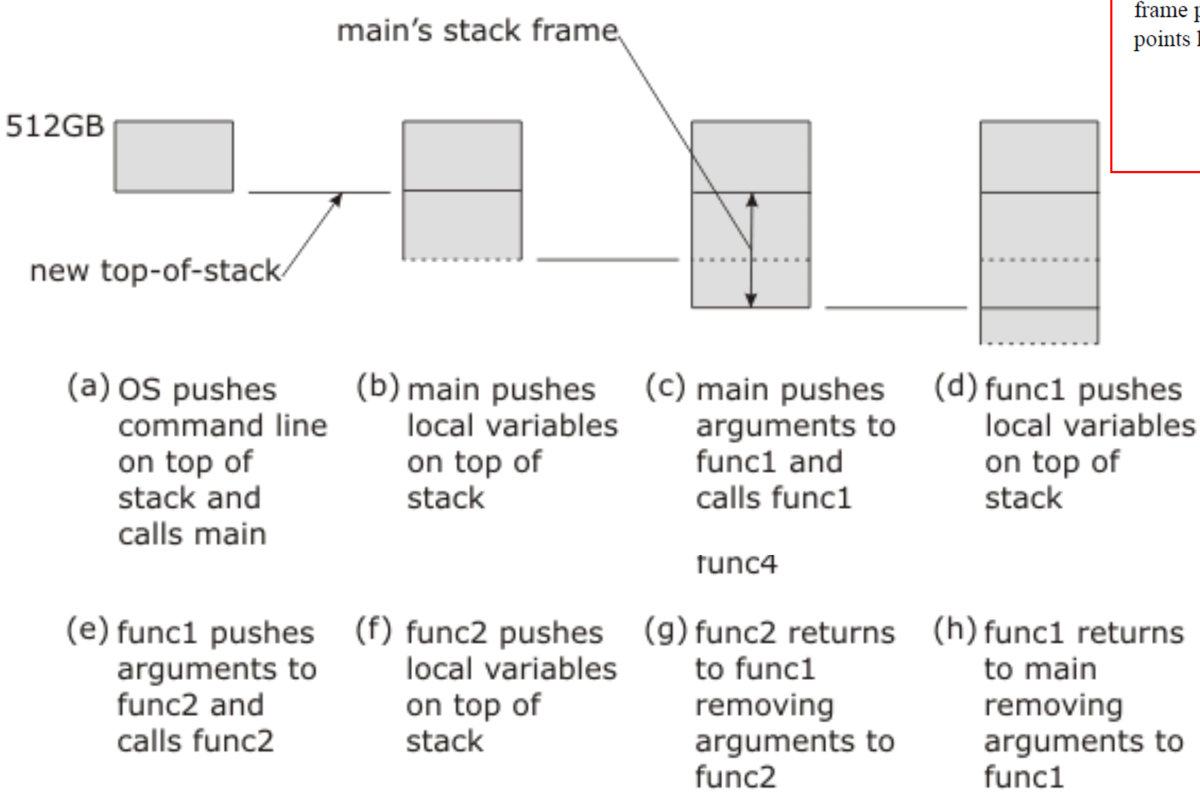
- Call graph

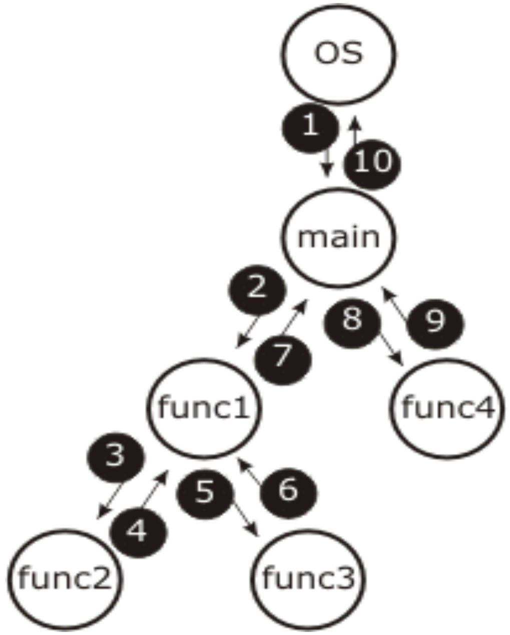
func(72,73) called from main()

```
func(int x, int y) {
  int a;
  int b[3];
/* no other auto variables */
```

Stack

Offset from current frame pointer (for func())		Contents
	main() auto variables	
+12	73	y
+8	72	x
+4	ra	return address
frame pointer points here → 0	mfp	caller's frame pointer
−4	garbage	a
−8	garbage	b[2]
−12	garbage	b[1]
−16	garbage	b[0]





Inline Function

Definition of the inline function is typically placed before it is used (e.g. in the header file) so the compiler knows how to expand the function when it sees it

- Inline function
  - ☞ Use the `inline` qualifier

```
inline double myfunction(double x) {…}
```

  - ☞ "Advise" the compiler to generate a copy of the function's code in place to avoid a function call

- Trade-off of inline functions
  - Reduce function call overhead—especially for small and frequently used functions
  - Multiple copies of the function code are inserted in the program—often making the program larger

Example on Inline Function

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

inline int sum(int n)
{
    int val = 0;
    for (int i=1;i<=n;i++)
    val += i;
    return val;
}

int main()
{
    int num, total;
    cin >> num;
    total = sum(num);
    cout << total << '\n';
    total = sum(num*2);
    cout << total << '\n';
}
```

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

int main()
{
    int num, total;
    cin >> num;
    {
        int n = num;
        int val = 0;
        for (int i=1;i<=n;i++) val += i;
        total = val;
    }
    cout << total << '\n';
    {
        int n = 2*num;
        int val = 0;
        for (int i=1;i<=n;i++) val += i;
        total = val;
    }
    cout << total << '\n';
}
```

Recursive Function

- Recursive function
  - A function that calls itself, either directly, or indirectly (through another function)
- Divide and conquer
  - Repeatedly performing a smaller task by itself

☞ Computation of the factorial

- $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n \cdot (n-1)!$
- Example: $5! = 5 \cdot (4!) = 5 \cdot 4 \cdot (3!) = \ldots$
  $= 5 \cdot 4 \cdot 3 \cdot 2 \cdot (1!)$

Factorial function: `factorial(n)` ↔ `n!`

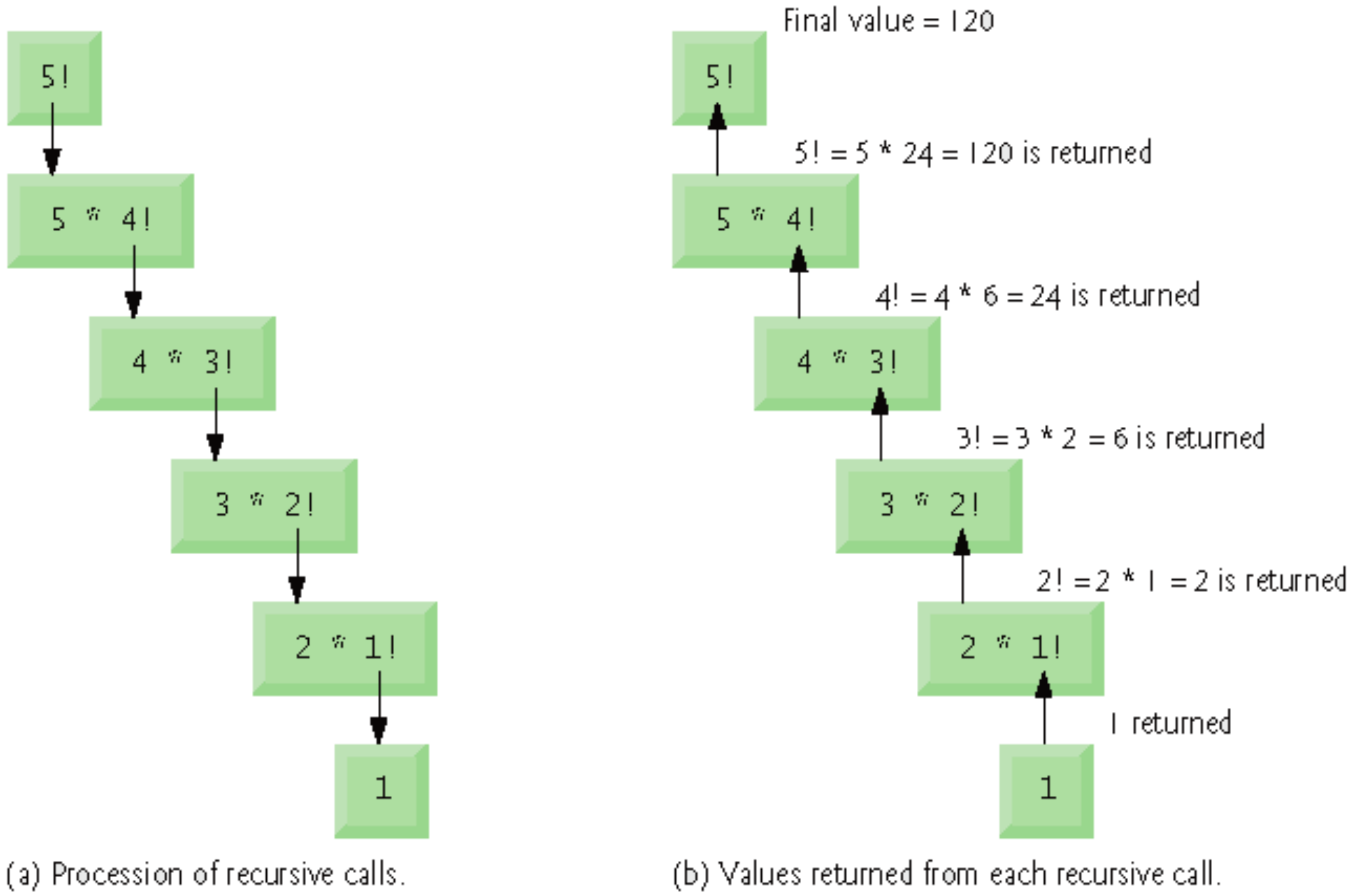
Using Recursive Function

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

unsigned long factorial(int);

int main()
{
    for (int counter = 0; counter <= 10; counter++)
    cout << counter << "! = " << factorial(counter) << endl;
}

unsigned long factorial(int number)
{
    if (number <= 1)
        return 1;
    else
        return number * factorial(number - 1);
}
```

Test for the base cases:
0! = 1 and 1! = 1

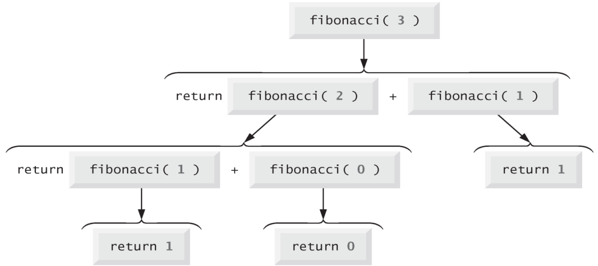
Recursive Call

```
unsigned long factorial(unsigned long number)
{
    if ( number <= 1 )
        return 1;
    else
        return number * factorial(number - 1);
}
```



More on Recursion

- Recursion
  - The function divides the problem into two pieces
    - A piece that the function knows how to do
    - A piece that it does not know how to do
  - ① Base case
    - The simplest case, which the function knows how to handle
  - ② Recursive call (recursion step)
    - The function launches (calls) a fresh copy of itself to work on the smaller problem
    - Can result in many more recursive calls, as the function keeps dividing each new problem into two pieces
    - This sequence of smaller and smaller problems must eventually converge on the base case; otherwise the recursion will continue forever



Yet Another Example

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

unsigned long fibonacci(int);

int main()
{
    for (int c = 0; c <= 10; c++)
    cout << "fibonacci( " << c << " ) = "
         << fibonacci(c) << endl;
}

unsigned long fibonacci(int number)
{
    if (number <= 1)
        return number;
    else
        return fibonacci(number-1) + fibonacci(number-2);
}
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2$$

Note that there is no guarantee that fibonacci(2) will be executed *before* fibonacci(1)

It could be problematic if the order matters (e.g. operations with side effects)

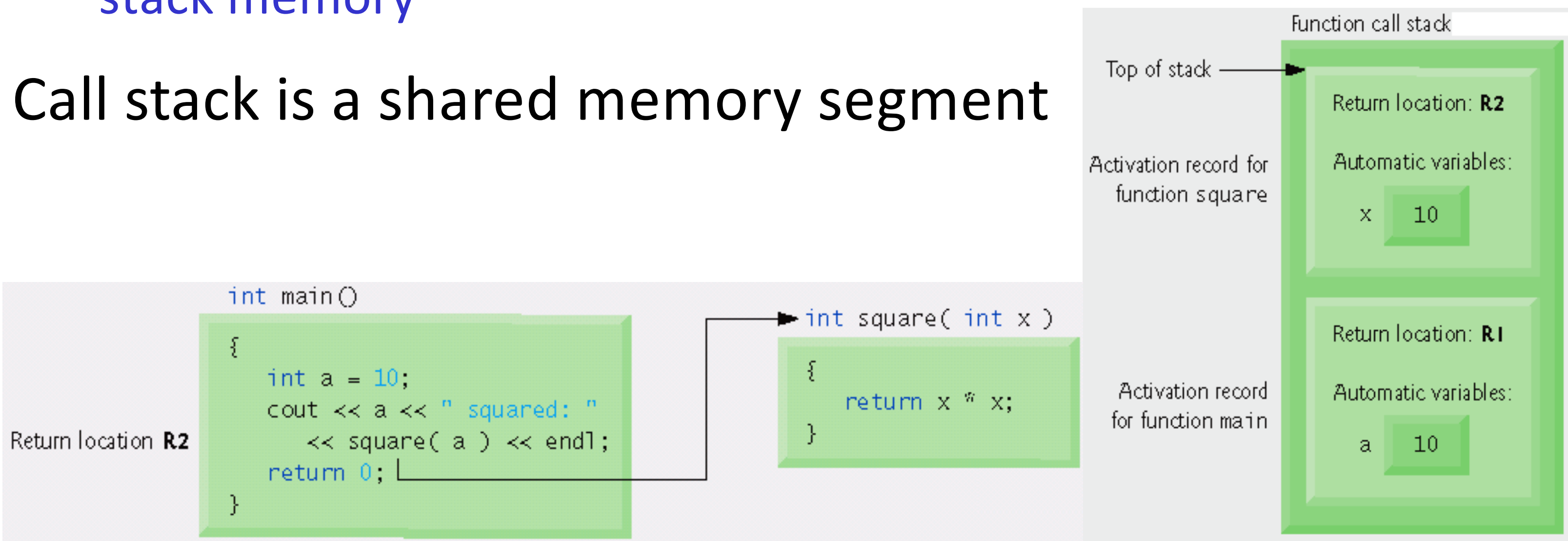
Order of evaluation is specified only for && || ?: , (sequence points)

Number of recursive calls to calculate the n^{th} Fibonacci number is $O(2^n)$

Be Careful of Using Recursion

- Negatives of recursion
  - Overhead of repeated function calls
    - Can be expensive in both processor time and memory space
  - Each recursive call causes another copy of the function data (e.g. the function's variables) to be created
    - Can consume considerable memory and cause overflow of the stack memory

☞ Call stack is a shared memory segment



Recursion vs. Iteration

- Both involve repetition
  - Iteration – explicitly uses repetition structure
  - Recursion – repeated function calls
- Both involve a termination test
  - Iteration – loop-termination test
  - Recursion – base case
- Both gradually approach termination
  - Iteration modifies counter until loop-termination test fails
  - Recursion produces progressively simpler versions of problem

Factorial Function Revisited

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

unsigned long factorial(int);

int main()
{
    for (int counter = 0; counter <= 10; counter++)
    cout << counter << "! = " << factorial(counter) << endl;
}

unsigned long factorial(int number)
{
    if (number <= 1) return 1;

    unsigned long prod=1;
    for (int i=2; i<=number; i++) prod *= i;
    return prod;
}
```

Fibonacci Function Revisited

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2$$

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

unsigned long fibonacci(int number)
{
    if (number == 0) return 0;
    unsigned long u=0, v=1, t;
    for (int i=2; i<=number; i++)
    {
        t = u + v;
        u = v;
        v = t;
    }
    return v;
}
int main()
{
    for (int c = 0; c <= 10; c++)
    cout << "fibonacci( " << c << " ) = " << fibonacci(c) << endl;
}
```