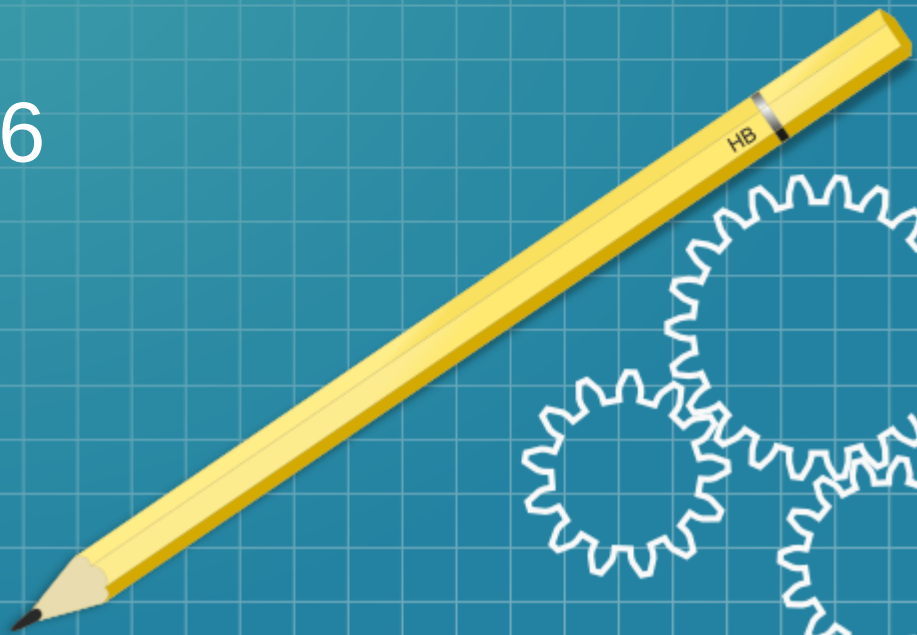
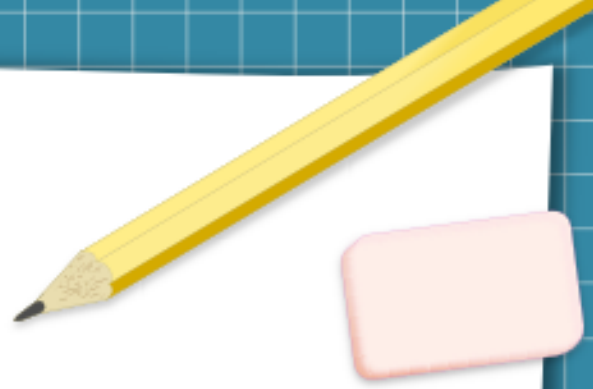
Functions and an Introduction to Recursion

Chapter 6



We will discuss:



- global functions
- function prototypes, function definitions
- Standard Library headers
- random numbers
- scopes
- function call stacks, activation records
- inline functions

Global Functions

Some functions, such as `main`, are *not* members of a class.

These functions are called *global functions*.

Example:

```
#include <cmath>

sqrt(3.4)
pow(2,6)
tan(1.2)
```

see *Section 6.3* for a list of the `cmath` library global functions

Function prototypes and definitions

- we can define a function and later on use it in our code

or

- we can *declare* a function, by giving its *prototype*, and then define it later in the program.

Function prototypes and definitions

- we can define a function and later on use it in our code

or

- we can *declare* a function, by giving its *prototype*, and then define it later in the program.

Example: consider the following code fragment:

```
int maximum(int a, int b);

void main(){

    int x = 10, y = 9;
    cout << maximum(x,y) << endl;
}

int maximum(int a, int b){
    if (a > b) { return a; }
     else {return b; }
}
```

function prototype

function signature

function call

function definition

Function prototypes and definitions

- we can define a function and later on use it in our code

or

- we can *declare* a function, by giving its *prototype*, and then define it later in the program.

Example: consider the following code fragment (without function prototype):

```
int maximum(int a, int b){
    if (a > b) { return a; }
     else {return b; }
}

void main(){

    int x = 10, y = 9;
    cout << maximum(x,y) << endl;
}
```

function definition

function call

Function prototypes and definitions

When compiling the program, the computer uses the prototype to:

- check that the *signature* in the prototype matches the *signature* in the definition
- check that the call of the function contains *the correct # and types of arguments*, types are in correct order.
- ensure the *return type* can be used correctly in the expression that called the function

Few more comments

The *scope* of the function is the region of a program in which the function is known and is accessible.

Functions in the same scope *must have unique signatures*.

Argument coercion is a feature of function prototypes: forcing arguments to the appropriate types, specified by the prototype, as long as it is not a "narrowing conversion."
Example: if we define our maximum function on doubles, but supply integers in the function call, the function would still work.

Argument-promotion and *implicit conversions*:
C++ has a list of *promotion rules*, which indicate the *implicit conversions* allowed between fundamental types.
`int` → `double`
`double` → `int` (truncates the decimal part)
...

Promotion hierarchy for arithmetic data types

highest type

long double	
double	
float	
unsigned long long int	(syn: unsigned long long)
long long int	(syn: long long)
unsigned long int	(syn: unsigned long)
long int	(syn: long)
unsigned int	(syn: unsigned)
int	
unsigned short int	(syn: unsigned short)
short int	(syn: short)
unsigned car	
char and signed char	
bool	

lowest type

C++ Standard Library Headers

C++ Standard Library is divided into many potions, each with its own header.

If we want to use, say standard input/output functions, we need to include appropriate header, `<iostream>` in this case.

See *Section 6.6* of the textbook that lists some common headers with brief explanations of what they are used for.

Random Numbers

`<random>` library provides us with *non deterministic random numbers* – a set of random numbers that can't be predicted.

See the example in RandomNumbersExample.cpp

Here you can find more information on *uniform discrete distribution*:
http://www.cplusplus.com/reference/random/uniform_int_distribution/

Scope Rules

Scope is a portion of a program where an identifier can be used.

Let's discuss:
- *block scope*, and
- *global namespace scope*

Later on we will see:
- *class scope* (Chapter 9)
- *function scope* (Chapter 23)
- *function-prototype scope* (Chapter 23)
- *namespace scope* (Chapter 23)

Block Scope

Identifiers declared *inside a block* have *block scope* .

It *begins* at identifier's declaration and *ends* at the terminating closing brace(})is a portion of a program where an identifier can be used.

Local variables have block scope.

Function parameters have block scope (even though they are declared outside the block's braces).

When blocks are nested and an identifier in the outside block has the same name as an identifier in the inner block, the outer identifier is "hidden" until the inner block terminates.

Block Scope: static local variables

A local variable can be declared static:

```
static int index;
```

All `static` local variables of numeric types are initialized to zero by default.

These variables also have block scope, but they retain its value when the function returns to its caller. The next time the function is called, the `static` local variable contains the value it had after the function's last execution.

See example in scopesExample.cpp

Global Namespace Scope

Identifiers declared *outside any function or class* have *global namespace scope* .

Such identifier is known to all functions from the point of declaration until the end of the file.

Function definitions and *prototypes* (placed outside a function), *class definitions, global variables* all have *global namespace scope*.

Variables declared outside any class or function definition are called *global variables*. They retain their values throughout a program's execution.

A general practice: *avoid use of global variables unless they are truly needed*. See example in scopesExample2.cpp

Function Call Stack and Activation Records

Stack is *last-in last-out* (LIFO) data structure.

The access is restricted to one end:

- we can add an element to the end/top (*push* an item onto a stack),
- check the element at the end (operation *top*), and
- retrieve the element from the end/top (operation pop).

Function Call Stack and Activation Records

function call stack / *program execution stack* is a data structure working behind the scenes.

It supports the function's call/return mechanism, creation, maintenance, and destruction of each called function's local variables.

Each time a function calls another function, a *stack frame* / *activation record* is created and pushed onto a *function call stack*.

Stack frame contains the *return address*, *local non-static variables* and *their values*.

Stack overflow: when the memory used to store activation records is full and no more records can be added.

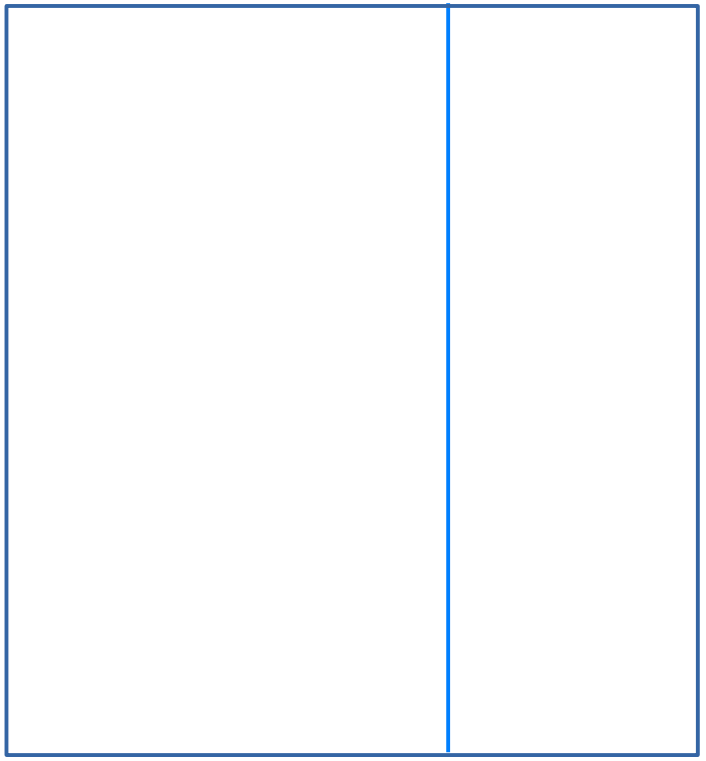
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



call stack

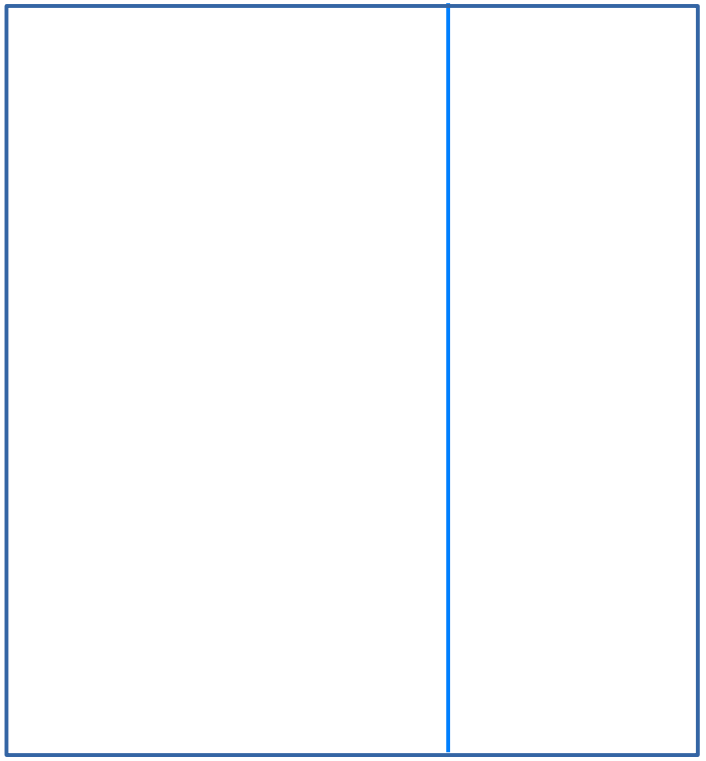
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



a = 3

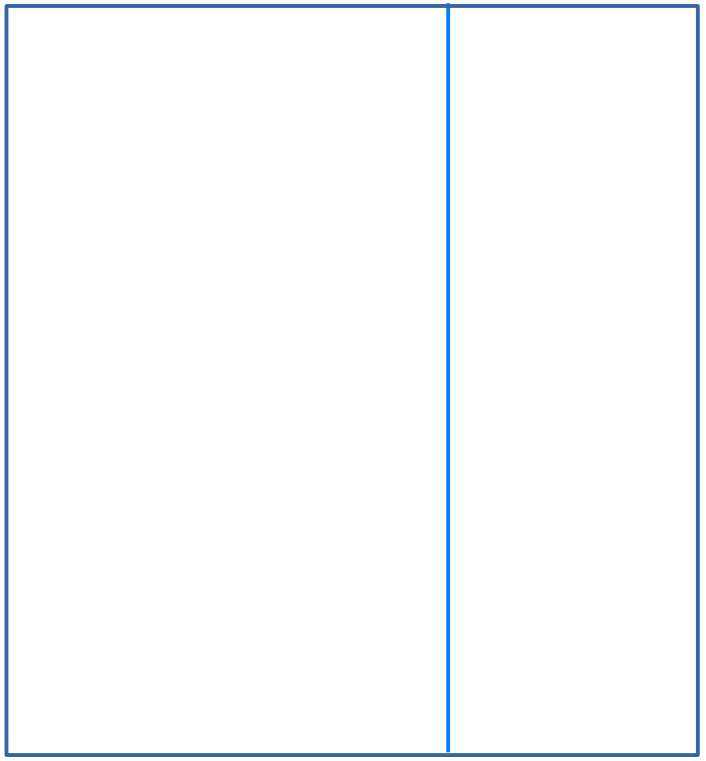
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



call stack

a = 3, b = 4

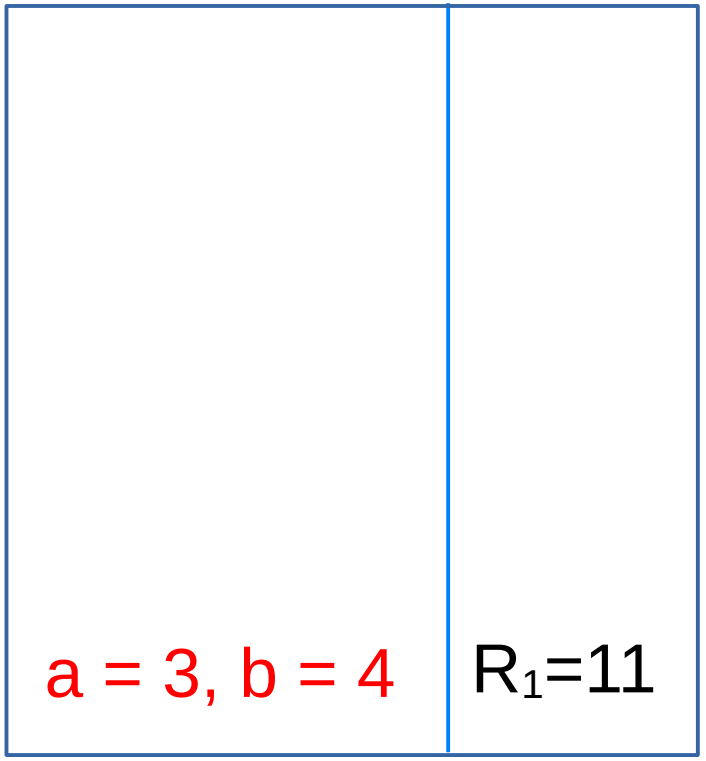
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



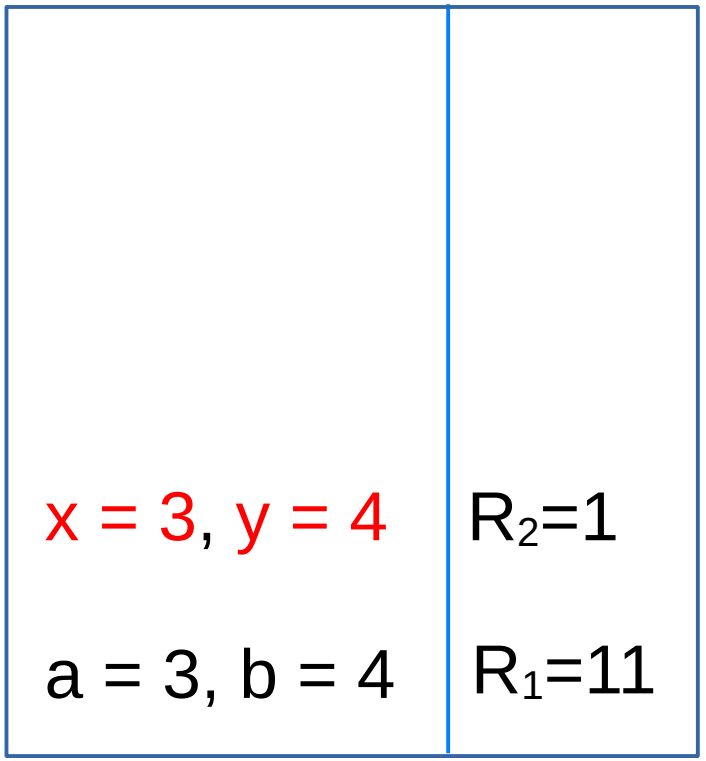
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



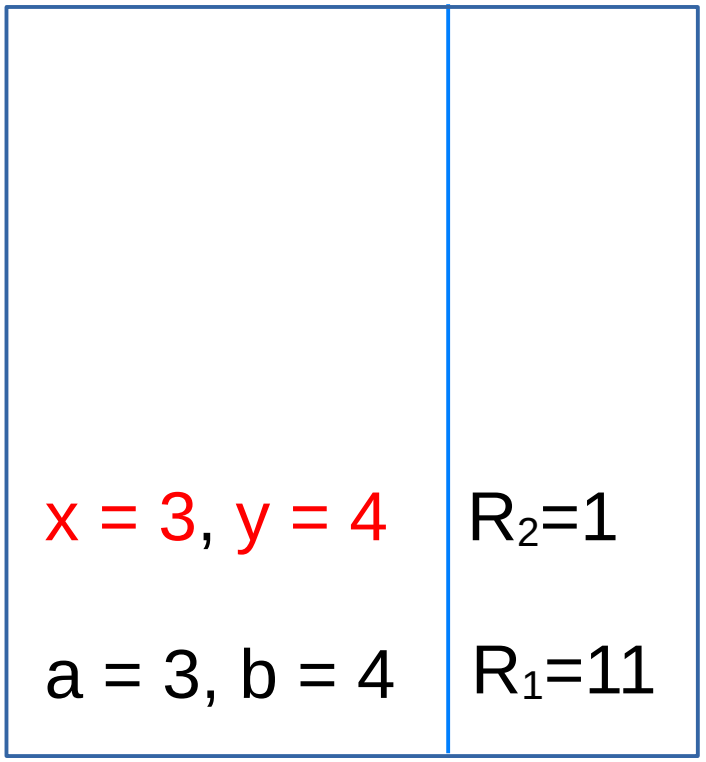
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



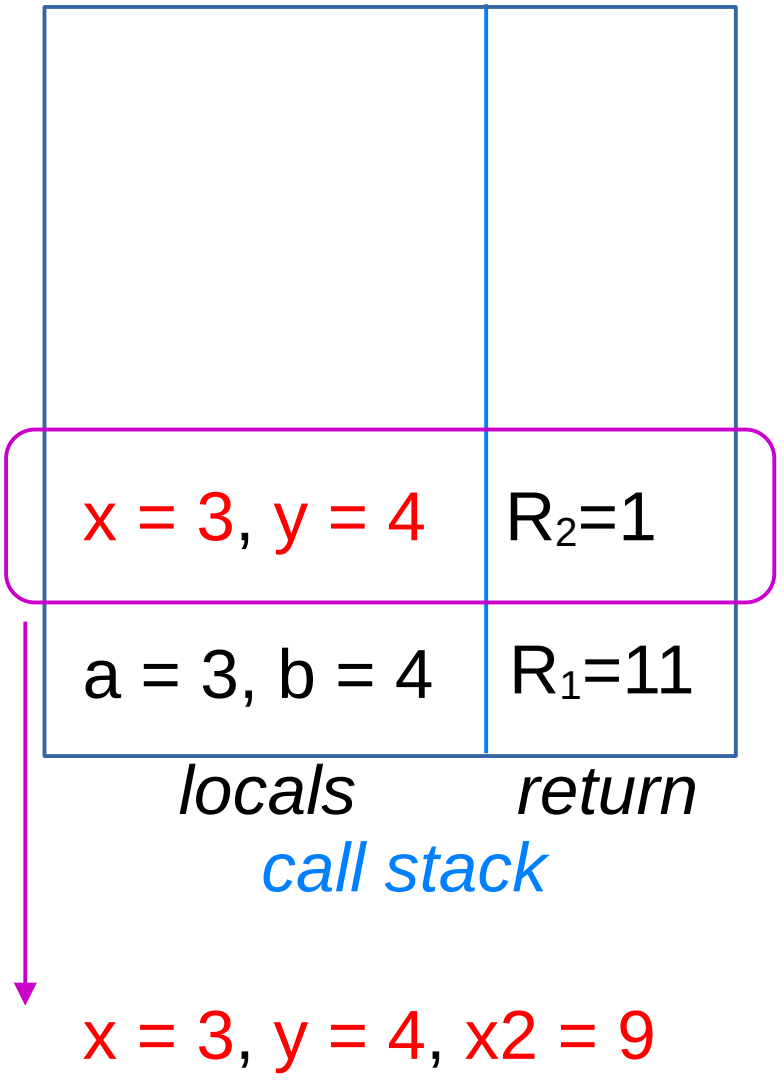
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



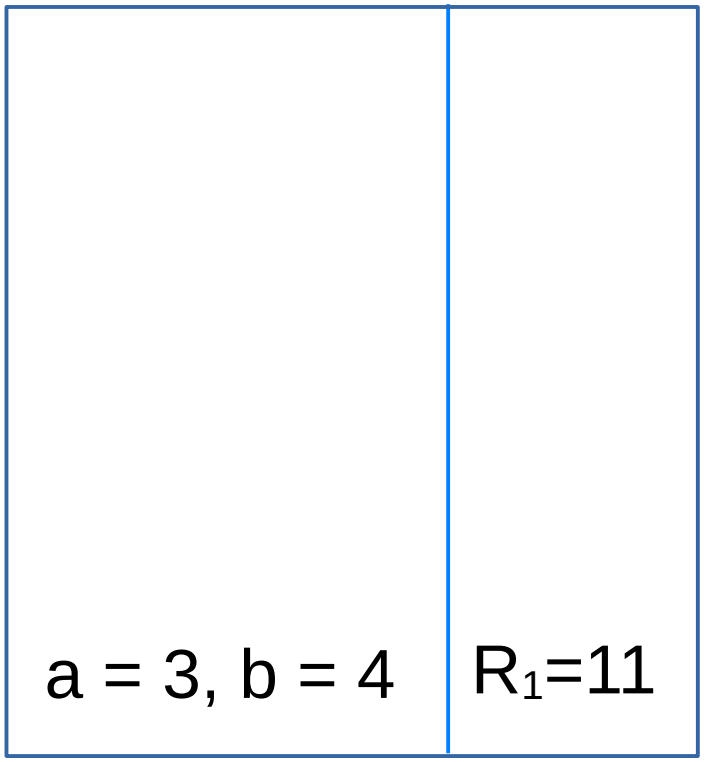
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



x = 3, y = 4, x2 = 9

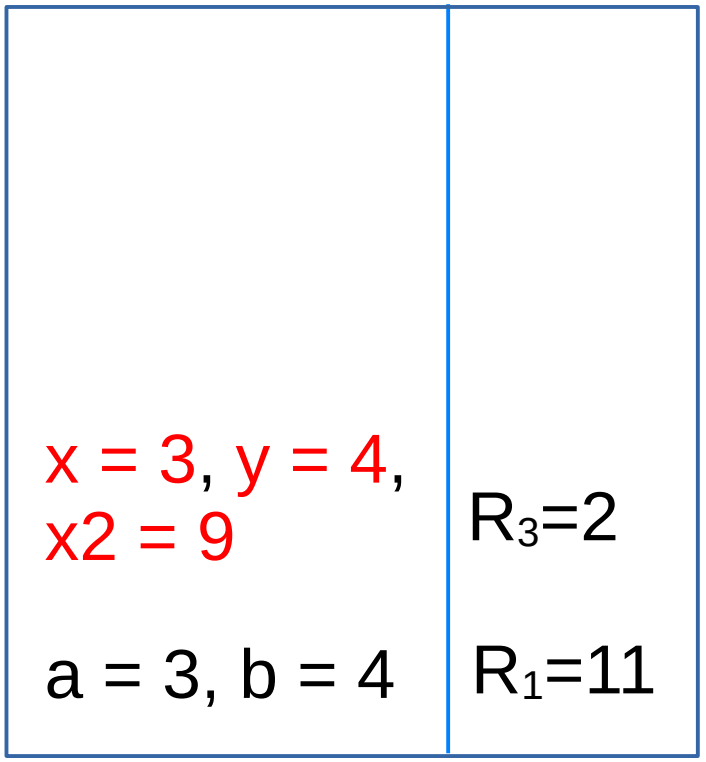
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



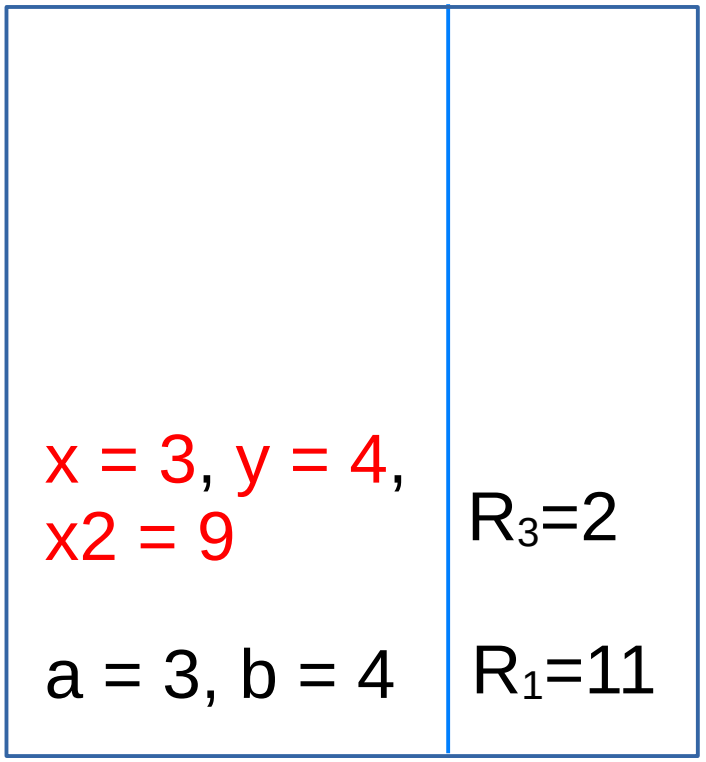
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



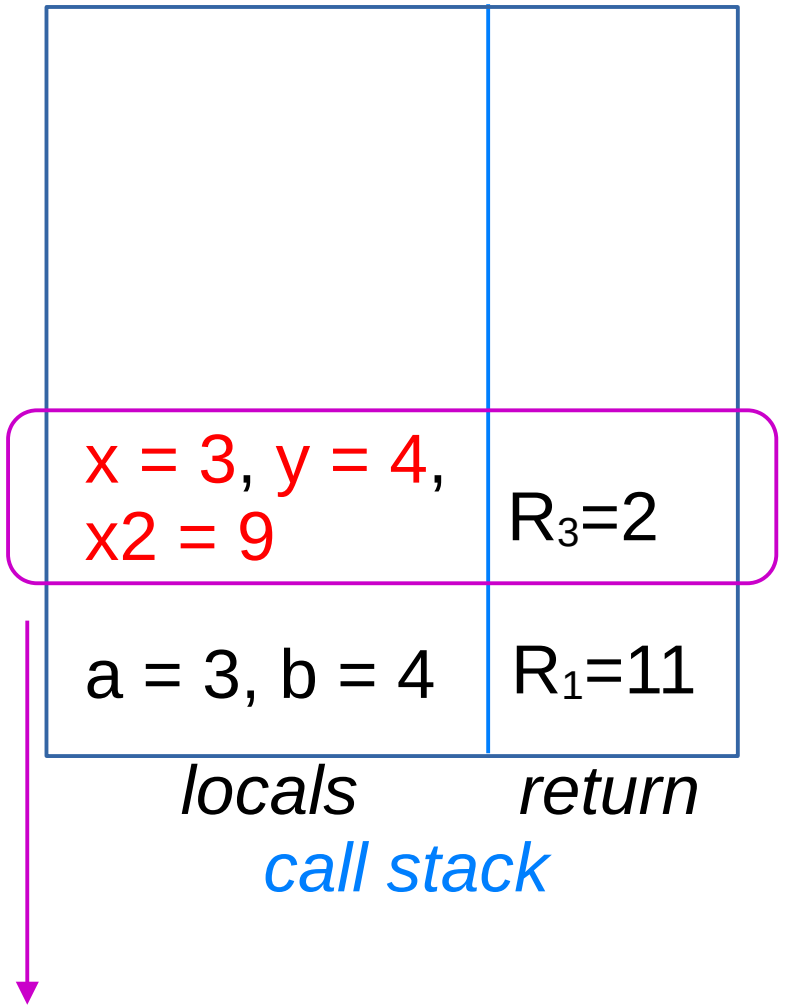
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



x = 3, y = 4, x2 = 9, y2 = 16

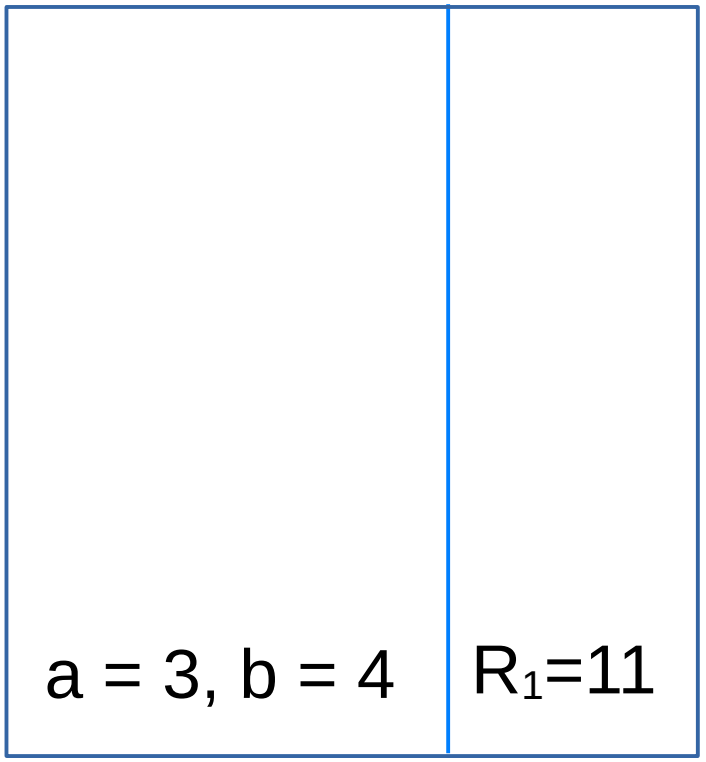
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



x = 3, y = 4, x2 = 9, y2 = 16

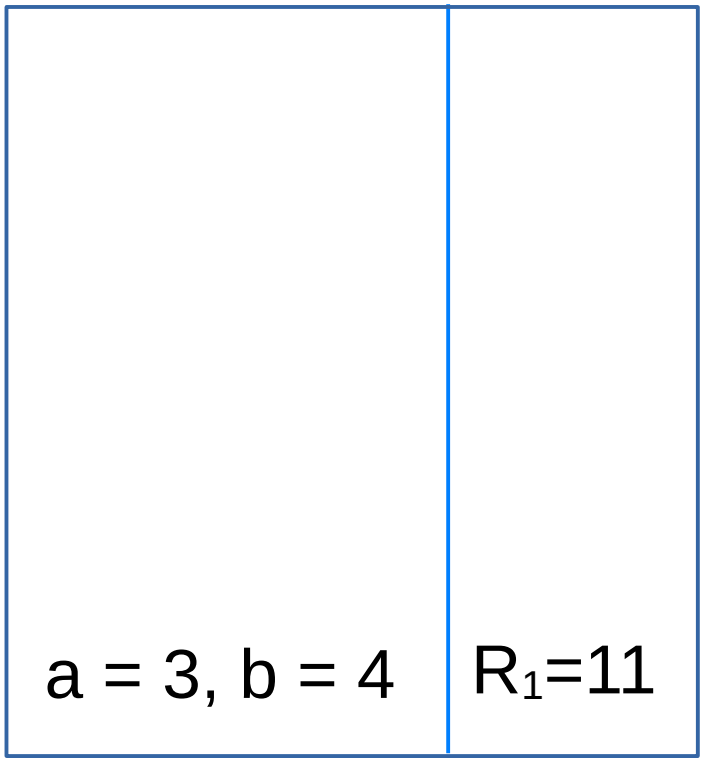
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



x = 3, y = 4, x2 = 9, y2 = 16, z = 25

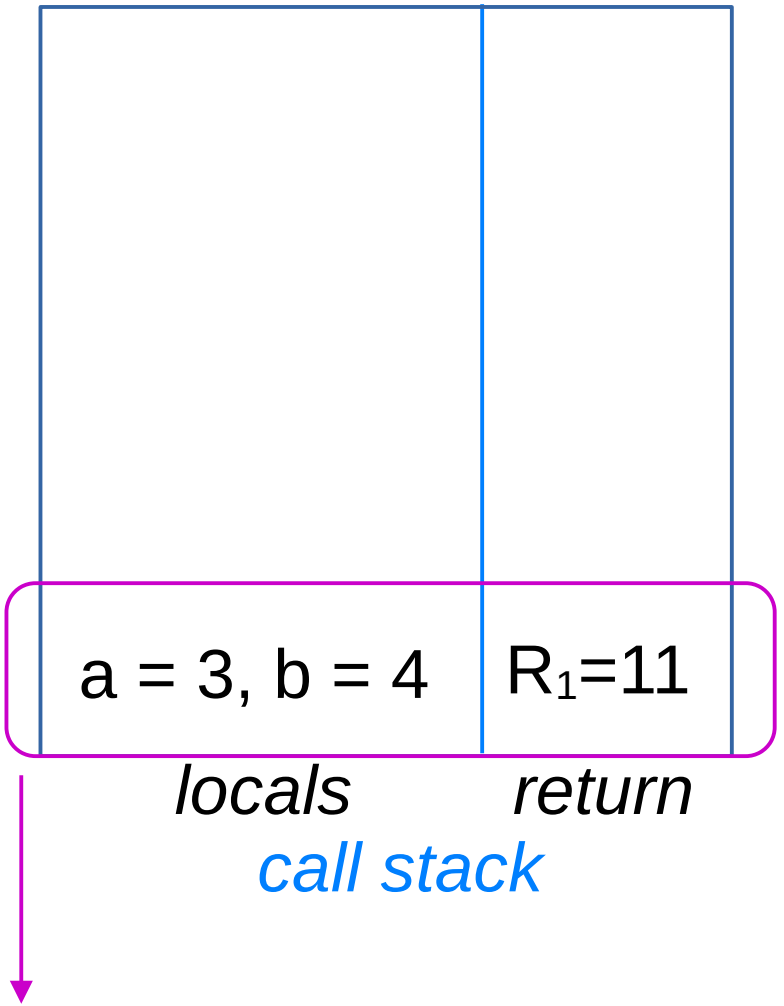
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



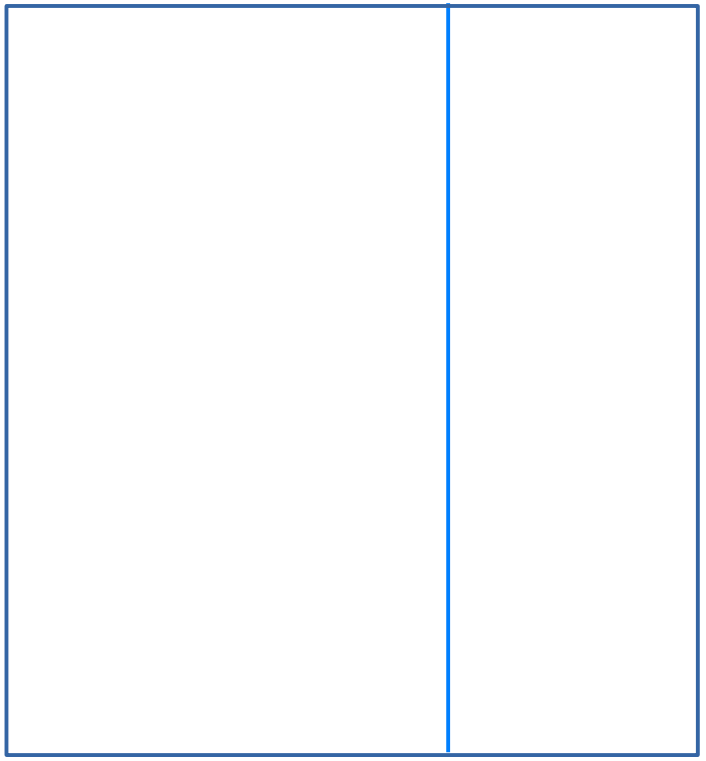
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



call stack

a = 3, b = 4, c = 25

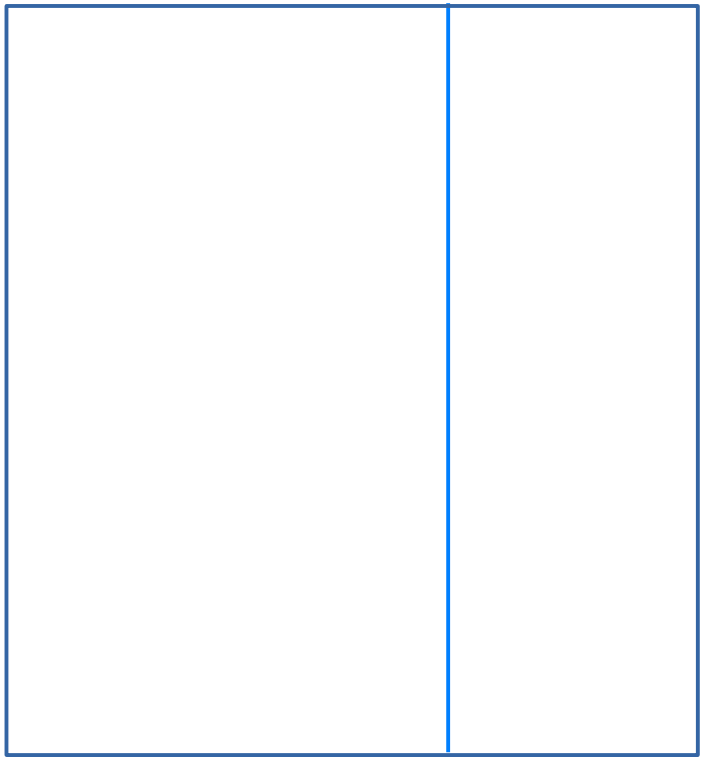
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



call stack

a = 3, b = 4, c = 25

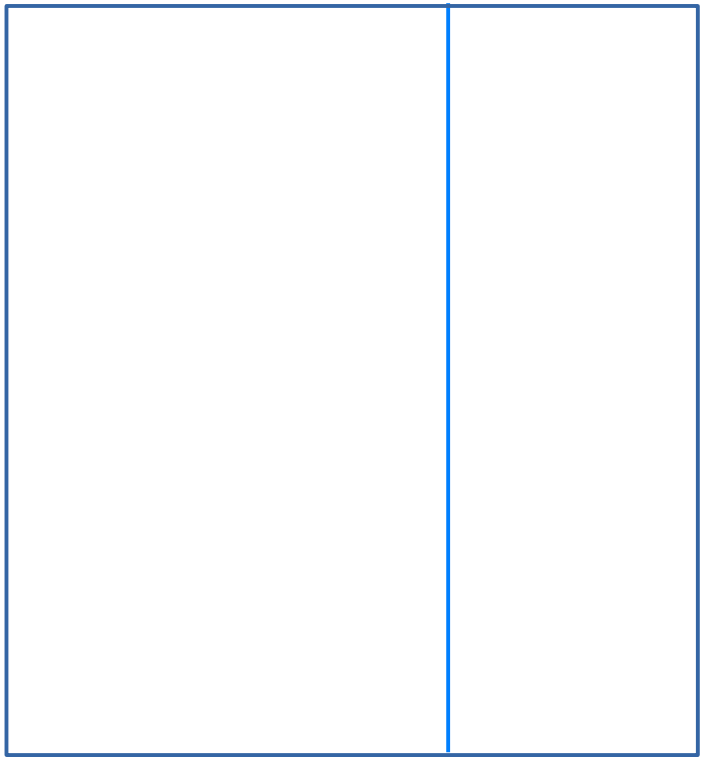
Function Call Stack and Activation Records

Consider the following code fragment:

```
0: int A(int x, int y){
1:     x2 = B(x);
2:     y2 = B(y);
3:     z = x2 + y2;
4:     return z; }

5: int B(int n) {
6:     n2 = n * n;
7:     return n2; }

8: int main() {
9:     a = 3;
10:    b = 4;
11:    c = A(a, b);
12:    cout << c;
13:    return 1;}
```



call stack

a = 3, b = 4, c = 25

Inline functions

C++ provides *inline functions* to help reduce function-call overhead.

Placing the qualifier *inline* before a function's return type in the function definition *advises the compiler to generate a copy of the function's body code* in every place where the function is called, when appropriate, to *avoid a function call*.

Inline functions

C++ provides *inline functions* to help reduce function-call overhead.

Placing the qualifier *inline* before a function's return type in the function definition *advises the compiler to generate a copy of the function's body code* in every place where the function is called, when appropriate, to *avoid a function call*.

This often makes the program larger.

The compiler may ignore the *inline* qualifier and generally does so for all but the *smallest functions*.

Inline functions

Consider this code fragment:

```
inline double area(double l, double w) {
    return l*w;
}

inline double perimeter(double l, double w) {
    return 2*l + 2*w;
}

int main(){
… // get length and width from the user

cout << "perimeter P=" << perimeter(l,w);
cout << ", area A=" << area(l,w) << endl;}
}
```

See example in inlineFunctions.cpp

In-class work

1. Let's write a small library of short-code functions! Here is a list of functions we would like to have:

1) distance between two points given by their coordinates
Distance *D* between points *P* (x_1, y_1) and *Q* (x_2,y_2) is

$$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

2) the length of a hypotenuse in a right triangle if two legs (other two sides) are given.

$$a^2+b^2=c^2$$ *Pythagorean theorem*

3) perfect number check: an integer is a *perfect number* if the sum of its divisors, including 1, but not the number itself, is equal to the number.
Example: 6 = 1 + 2 + 3 , 6 is a perfect number

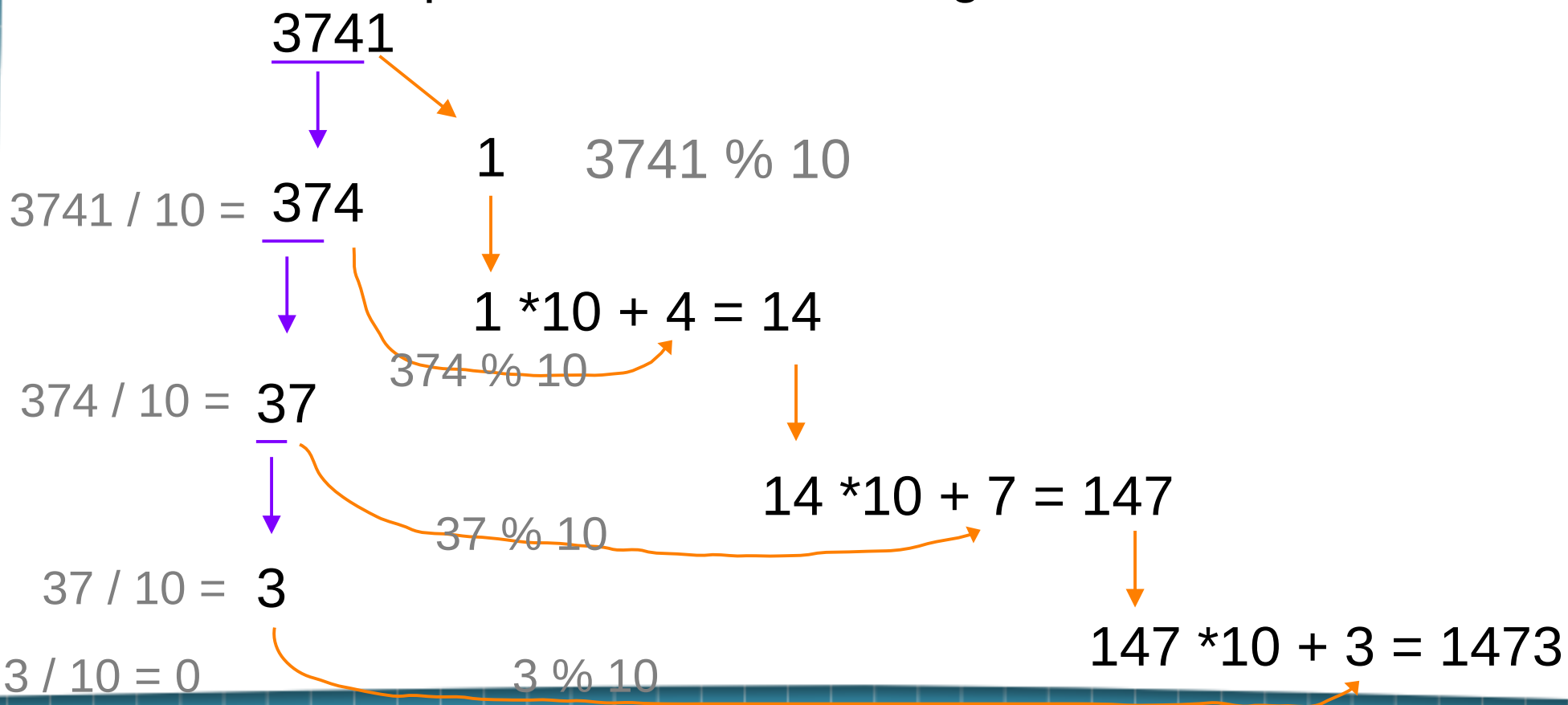
Which of these functions would qualify for *inline* ?

In-class work

2. Write a function that takes an integer value and returns the number with its digits reversed.

Example: given the number 63784 the function returns 48736

Hint: look at this procedure of reversing 3741:



HW assignment

1) Exercise 6.31
2) Exercise 6.33

Self-study:
Read Sections 6.7 – 6.8

Suggested exercises
(*not for grade, but the questions related to these will appear on a quiz or a test*):
1) Chapter 6,
   Summary and all Self-Review Exercises (pages 203-205)
2) Chapter 6, Exercises: 6.11, 6.16, 6.17, 6.18