C++ Basics

Arithmetic

Arithmetic

y = a + b * c – d -> **expression**

a + b -> **addition operation**

+ -> **addition operator**

a, b - > **operands**

a++ -> **postfix increment operation**

++ -> **postfix operator**

a -> **operand**

Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise order determined by the following **rules of operator precedence:**

1. Operators in expressions contained within pairs of parentheses are applied first. Parentheses are said to be at the "*highest level of precedence*."

In cases of nested, or embedded, parentheses, such as

(a * (b + c))

the operators in the ***innermost*** pair of parentheses (b+c) *are applied first.*

Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise order determined by the following **rules of operator precedence:**

2. *Multiplication(*), division(/) and remainder(%)* operations are evaluated next. If an expression contains several multiplication, division and remainder operations, operators are applied from *left to right*.

Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise order determined by the following **rules of operator precedence:**

3. *Addition(+)* and *subtraction(-)* operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from *left to right*. Addition and subtraction also have the *same level* of precedence.

So, **the rules of operator precedence define the order in which C++ applies operators.**

Rules of Operator Precedence

```
y = a * x * x + b * x + c;
  6   1   2   4   3   5
```

Step 1. `y = 2 * 5 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`2 * 5 is 10`

Step 2. `y = 10 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`10 * 5 is 50`

Step 3. `y = 50 + 3 * 5 + 7;` *(Multiplication before addition)*

`3 * 5 is 15`

Step 4. `y = 50 + 15 + 7;` *(Leftmost addition)*

`50 + 15 is 65`

Step 5. `y = 65 + 7;` *(Last addition)*

`65 + 7 is 72`

Step 6. `y = 72` *(Low-precedence assignment—place `72` in `y`)*

Operator precedence and associativity

Operator	Type	Associativity
::	binary scope resolution	left to right
::	unary scope resolution	
()	grouping parentheses *[See caution in Fig. 2.10 regarding grouping parentheses.]*	
()	function call	left to right
[]	array subscript	
.	member selection via object	
->	member selection via pointer	
++	unary postfix increment	
--	unary postfix decrement	
typeid	runtime type information	
dynamic_cast<*type*>	runtime type-checked cast	
static_cast<*type*>	compile-time type-checked cast	
reinterpret_cast<*type*>	cast for nonstandard conversions	
const_cast<*type*>	cast away const-ness	
++	unary prefix increment	right to left
--	unary prefix decrement	
+	unary plus	
-	unary minus	
!	unary logical negation	
~	unary bitwise complement	
sizeof	determine size in bytes	
&	address	
*	dereference	
new	dynamic memory allocation	
new[]	dynamic array allocation	
delete	dynamic memory deallocation	
delete[]	dynamic array deallocation	
(*type*)	C-style unary cast	right to left

Operator precedence and associativity

<table>
<tr><th>Operator</th><th>Type</th><th>Associativity</th></tr>
<tr><td>.*
->*</td><td>pointer to member via object
pointer to member via pointer</td><td>left to right</td></tr>
<tr><td>*
/
%</td><td>multiplication
division
remainder</td><td>left to right</td></tr>
<tr><td>+
-</td><td>addition
subtraction</td><td>left to right</td></tr>
<tr><td><<
>></td><td>bitwise left shift
bitwise right shift</td><td>left to right</td></tr>
<tr><td><
<=
>
>=</td><td>relational less than
relational less than or equal to
relational greater than
relational greater than or equal to</td><td>left to right</td></tr>
<tr><td>==
!=</td><td>relational is equal to
relational is not equal to</td><td>left to right</td></tr>
<tr><td>&</td><td>bitwise AND</td><td>left to right</td></tr>
<tr><td>^</td><td>bitwise exclusive OR</td><td>left to right</td></tr>
<tr><td>|</td><td>bitwise inclusive OR</td><td>left to right</td></tr>
<tr><td>&&</td><td>logical AND</td><td>left to right</td></tr>
<tr><td>||</td><td>logical OR</td><td>left to right</td></tr>
<tr><td>?:</td><td>ternary conditional</td><td>right to left</td></tr>
<tr><td>=
+=
-=
*=
/=
%=
&=
^=
|=
<<=
>>=</td><td>assignment
addition assignment
subtraction assignment
multiplication assignment
division assignment
remainder assignment
bitwise AND assignment
bitwise exclusive OR assignment
bitwise inclusive OR assignment
bitwise left-shift assignment
bitwise right-shift assignment</td><td>right to left</td></tr>
<tr><td>,</td><td>comma</td><td>left to right</td></tr>
</table>

Associativity

When we say that certain operators are applied from *left to right* or *right to left*, we are referring to the **associativity** of the operators.

For example, the addition operators (+) in the expression

$$a + b + c + d$$

associate from *left to right* and is parsed as

$$((a + b) + c) + d$$

Associativity

The associativity of **assignment(=)** operator is *right to left*:

a = b = c = d

is parsed as

a = (b = (c = d))

not

((a = b) = c) = d

Order of evaluation

- Order of evaluation of any part of any expression, including order of evaluation of ***function arguments*** is **unspecified** (with some exceptions).
- The compiler can evaluate operands and other subexpressions in any order and may choose another order when the same expression is evaluated again.

➢ There is **NO** concept of *left-to-right or right-to-left evaluation* in C++. This is not to be confused with left-to-right and right-to-left associativity of operators.

The expression

```
a() + b() + c()
```

is parsed as

```
(a() + b()) + c()
```

due to left-to-right associativity of operator+, **but c() may be evaluated first, last, or between a() or b() at run time**:

Order of evaluation

```cpp
#include <cstdio>

int a() { return std::puts("a"); }
int b() { return std::puts("b"); }
int c() { return std::puts("c"); }

void z(int, int, int) {}

int main()
{
    z(a(), b(), c());       // all 6 permutations of output are allowed
    return a() + b() + c(); // all 6 permutations of output are allowed
}
```

Possible output:

```
b
c
a
c
a
b
```

Please refer to this for more details:
https://en.cppreference.com/w/cpp/language/eval_order

Order of evaluation

- Order of evaluation of any part of any expression, including order of evaluation of *function arguments* is **unspecified** (<u>with some exceptions</u>).

Example of some exceptions:

- **Logical AND operator** (**&&**) <u>guarantees</u> *left-to-right* evaluation: the second operand is not evaluated if the first operand is **false.**

  a() && b() // b() will be evaluated if a() is <u>true</u>

- **Logical OR operator (||)** <u>guarantees</u> *left-to-right* evaluation; moreover, the second operand is not evaluated if the first operand evaluates to **true**

  a() || b() // b() will NOT be evaluated if a() is <u>true</u>

Examples

```
int x[2] = {0,10};
int* xPtr = x;
```

- `cout << ++*xPtr;`
- `cout << *++xPtr;`
- `cout << *xPtr++;`
- `cout << (x[0] ? 5 : x[1] ? 555 : 10);`
- `cout << x[0] ? 10 : 1000;`

Examples

```
int x[2] = {0,10};
int* xPtr = x;
```

- `cout << ++*xPtr; // ++(*xPtr),  1`
- `cout << *++xPtr;`
- `cout << *xPtr++;`
- `cout << (x[0] ? 5 : x[1] ? 555 : 10);`
- `cout << x[0] ? 10 : 1000;`

Examples

```
int x[2] = {0,10};
int* xPtr = x;
```

- `cout << ++*xPtr; // ++(*xPtr),  1`
- `cout << *++xPtr; // *(++xPtr),  10`
- `cout << *xPtr++;`
- `cout << (x[0] ? 5 : x[1] ? 555 : 10);`
- `cout << x[0] ? 10 : 1000;`

Examples

```
int x[2] = {0,10};
int* xPtr = x;
```

- `cout << ++*xPtr; // ++(*xPtr),  1`
- `cout << *++xPtr; // *(++xPtr),  10`
- `cout << *xPtr++; // *(xPtr++),  0`
- `cout << (x[0] ? 5 : x[1] ? 555 : 10);`
- `cout << x[0] ? 10 : 1000;`

Examples

```
int x[2] = {0,10};
int* xPtr = x;
```

- `cout << ++*xPtr; // ++(*xPtr),  1`
- `cout << *++xPtr; // *(++xPtr),  10`
- `cout << *xPtr++; // *(xPtr++),  0`
- `cout << (x[0] ? 5 : (x[1] ? 555 : 10)); // 555`
- `cout << x[0] ? 10 : 1000;`

Examples

```
int x[2] = {0,10};
int* xPtr = x;
```

- `cout << ++*xPtr; // ++(*xPtr),  1`
- `cout << *++xPtr; // *(++xPtr),  10`
- `cout << *xPtr++; // *(xPtr++),  0`
- `cout << (x[0] ? 5 : (x[1] ? 555 : 10)); // 555`
- `(cout << x[0]) ? 10 : 1000; // 0`

C++ Basics

If…else statement

How this will be parsed by compiler?

```
int x = 10, y = 4;
if (x > 5)
    if (y > 5)
        cout << "x and y are > 5";
else
    cout << "x is <= 5";
```

If…else statement

How this will be parsed by compiler?

```
int x = 10, y = 4;
if (x > 5)
    if (y > 5)
        cout << "x and y are > 5";
else
    cout << "x is <= 5";
```

Result: x is <= 5

This is called **dangling-else** problem

If…else statement

```
int x = 10, y = 4;
if (x > 5) {
    if (y > 5) {
        cout << "x and y are > 5";
    } else {
        cout << "x is <= 5";
    }
}
```

- C++ compilers *always associate an **else** with the immediately preceding **if*** unless told to do otherwise by the placement of braces ({ and }).

If…else statement

- **Always use braces {} in control statements, even for single-statement bodies**

```
if (condition) {
    single-statement or multi-statement body
}
```

- **Indent** both body statements (or groups of statements) of an if…else statement.

```
if (a > 0) {
    cout << "a > 0";
} else if (a < 0) {
    cout << "a < 0";
} else {
    if (b > 0) {
        cout << "b > 0";
        if (c > 0) {
            cout << "c > 0";
        }
    }
}
```

for statement

```
for (initialization; loopContinuationCondition; increment)
{
    statement
}
```

1. *Initialization*
2. *loopContinuationCondition*
3. *Statement*
4. *Increment*

while statement

```
initialization;
while (loopContinuationCondition) {
    statement
    increment;
}
```

1. *Initialization*
2. *loopContinuationCondition*
3. *Statement*
4. *Increment*

for & while statements

Can you convert every **for** statement to **while**?

```
int n = 10;
for (int i = 0; i < n; ++i) {
    if (5 == i) {
        cout << "5";
    }
}
```

for & while statements

Can you convert every **for** statement to **while**?

```
int n = 10;
for (int i = 0; i < n; ++i) {
    if (5 == i) {
        cout << "5";
    }
}
```

```
----------------------------------------------------------------
```

```
int n = 10;
{
    int i = 0;
    while (i < n) {
        if (5 == i) {
            cout << "5";
        }
        ++i;
    }
}
```

for & while statements

```
int n = 10;
for (int i = 0; i < n; ++i) {
    if (5 == i) {
        continue;
    }
    ...
}
```

```
int n = 10;
{
    int i = 0;
    while (i < n) {
        if (5 == i) {
            continue;
        }
        ...
        ++i;
    }
}
```

for & while statements

```
int n = 10;
for (int i = 0; i < n; ++i) {
    if (5 == i) {
        continue;
    }
    ...
}
```

```
int n = 10;
{
    int i = 0;
    while (i < n) {
        if (5 == i) {
            ++i;
            continue;
        }
        ...
        ++i;
    }
}
```

Confusing the == and = operators

What is the difference between these two statements?

```
if (payCode == 4) { // good
    cout << "You get a bonus!" << endl;
}

if (payCode = 4) { // bad
    cout << "You get a bonus!" << endl;
}
```

if (paycode = 4) will be parsed as

1. paycode =4
2. If (paycode)

How to avoid from this kind of errors?

lvalues and rvalues

- Variable names are said to be **lvalues** (for "*left values*") because they can be used on an *assignment operator's left side*.

- **Literals** are said to be **rvalues** (for "*right values*") because they can be used on **only** *an assignment operator's right side*.

  x = 25; // x -> lvalue, 25 -> rvalue

- **Lvalues** can also be used as **rvalues** on the *right side* of an assignment, **but not vice versa**:

  y = x; // OK, y -> lvalue, x -> lvalue (*used as rvalue*)
  25 = x; // ERROR!

Confusing the == and = operators

- ✓ Programmers normally write conditions such as **x == 7** with the variable name (an lvalue) on the left and the literal (an rvalue) on the right. Placing the literal on the left, as in **7 == x**, enables the compiler to issue an error if you accidentally replace the == operator with = .

```
if (4 == payCode) { // good
    cout << "You get a bonus!" << endl;
}

if (4 = payCode) { // ERROR!
    cout << "You get a bonus!" << endl;
}
```

string literals

What is the difference between these two declarations?

```
    const char* literal = "Hello";
    literal[1] = 'a';
-----------------------------------------
    char str[] = {"hello"};
    str[1] = 'a';
```

string literals

What is the difference between these two declarations?

```
    const char* literal = "Hello";
    literal[1] = 'a'; // compile ERROR!
------------------------------------------
    char str[] = {"hello"};
    str[1] = 'a'; // OK
```

string literals

What is the difference between these two declarations?

```
    const char* literal = "Hello";
    char* p = const_cast<char*>(literal);
    p[1] = 'a';
-------------------------------------------
    char str[] = {"hello"};
    str[1] = 'a'; // OK
```

string literals

What is the difference between these two declarations?

```
    const char* literal = "Hello"; // stored in read-only memory
    char* p = const_cast<char*>(literal);
    p[1] = 'a'; // undefined behavior
------------------------------------------
    char str[] = {"hello"}; // stored in stack
    str[1] = 'a'; // OK
```

- Attempting to modify a string literal results in **undefined behavior**: they may be stored in **read-only storage** or combined with other string literals:

Scope Rules

The portion of a program where an identifier can be used is known as its **scope.**

- Block Scope

```
{
    int a;
}
```

- Global Namespace Scope - An identifier declared **outside** any *function* or *class* has *global namespace scope.*

- Function Scope

- Namespace Scope

```
namespace MyNamespace {
    int a;
}
...
MyNamespace::a = 5;
```

Scope Rules

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

void useLocal(); // function prototype
void useStaticLocal(); // function prototype
void useGlobal(); // function prototype

int x{1}; // global variable

int main() {
   cout << "global x in main is " << x << endl;

   int x{5}; // local variable to main

   cout << "local x in main's outer scope is " << x << endl;

   { // block starts a new scope
      int x{7}; // hides both x in outer scope and global x

      cout << "local x in main's inner scope is " << x << endl;
      cout << "global x in global namespace scope is " << ::x << endl;
   }

   cout << "local x in main's outer scope is " << x << endl;

   useLocal(); // useLocal has local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal(); // useGlobal uses global x
   useLocal(); // useLocal reinitializes its local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal(); // global x also retains its prior value

   cout << "\nlocal x in main is " << x << endl;
}
```

Scope Rules

```cpp
// useLocal reinitializes local variable x during each call
void useLocal() {
   int x{25}; // initialized each time useLocal is called

   cout << "\nlocal x is " << x << " on entering useLocal" << endl;
   ++x;
   cout << "local x is " << x << " on exiting useLocal" << endl;
}

// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function
void useStaticLocal() {
   static int x{50}; // initialized first time useStaticLocal is called

   cout << "\nlocal static x is " << x << " on entering useStaticLocal"
      << endl;
   ++x;
   cout << "local static x is " << x << " on exiting useStaticLocal"
      << endl;
}

// useGlobal modifies global variable x during each call
void useGlobal() {
   cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting useGlobal" << endl;
}
```

Scope Rules

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
global x in global namespace scope is 1
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Function Signature

The portion of a function prototype *that includes the* ***name of the function*** *and the* ***types of its arguments*** is called the **function signature** or simply the **signature**.

- The function's *return type* is **NOT part of the function signature**.
  Signature includes the following underlined members:

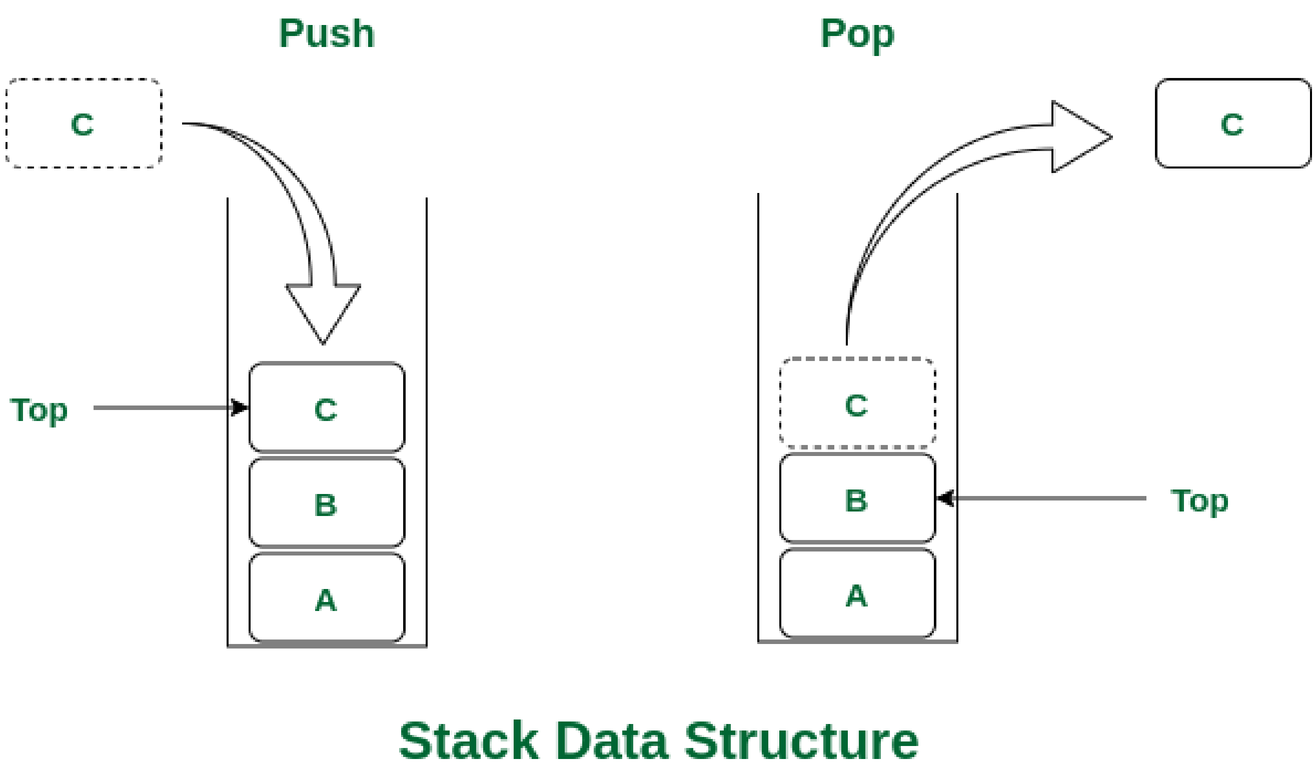
retType **function**(**ParamType1** ParamName1, **ParamType1** ParamName1)

```
...
void maximum(int, int, int);
int maximum(int, int, int);
...
```

Compile Error!!! Functions that differ only in their return type cannot be overloaded.

Function Call Stack

Stacks are known as **last-in, first-out (LIFO)** data structures—the last item **pushed (inserted)** on the stack is the *first* item **popped (removed)** from the stack.



Function Call Stack

Each time a function calls another function, an *entry* is **pushed** onto the **stack**. This entry is called a **stack frame** or an *activation record*.

Stack frame contains

- Return address to the calling function
- Non-static Local variables.

➢ If a function is called, the stack frame for the function call is simply ***pushed*** onto the call stack.

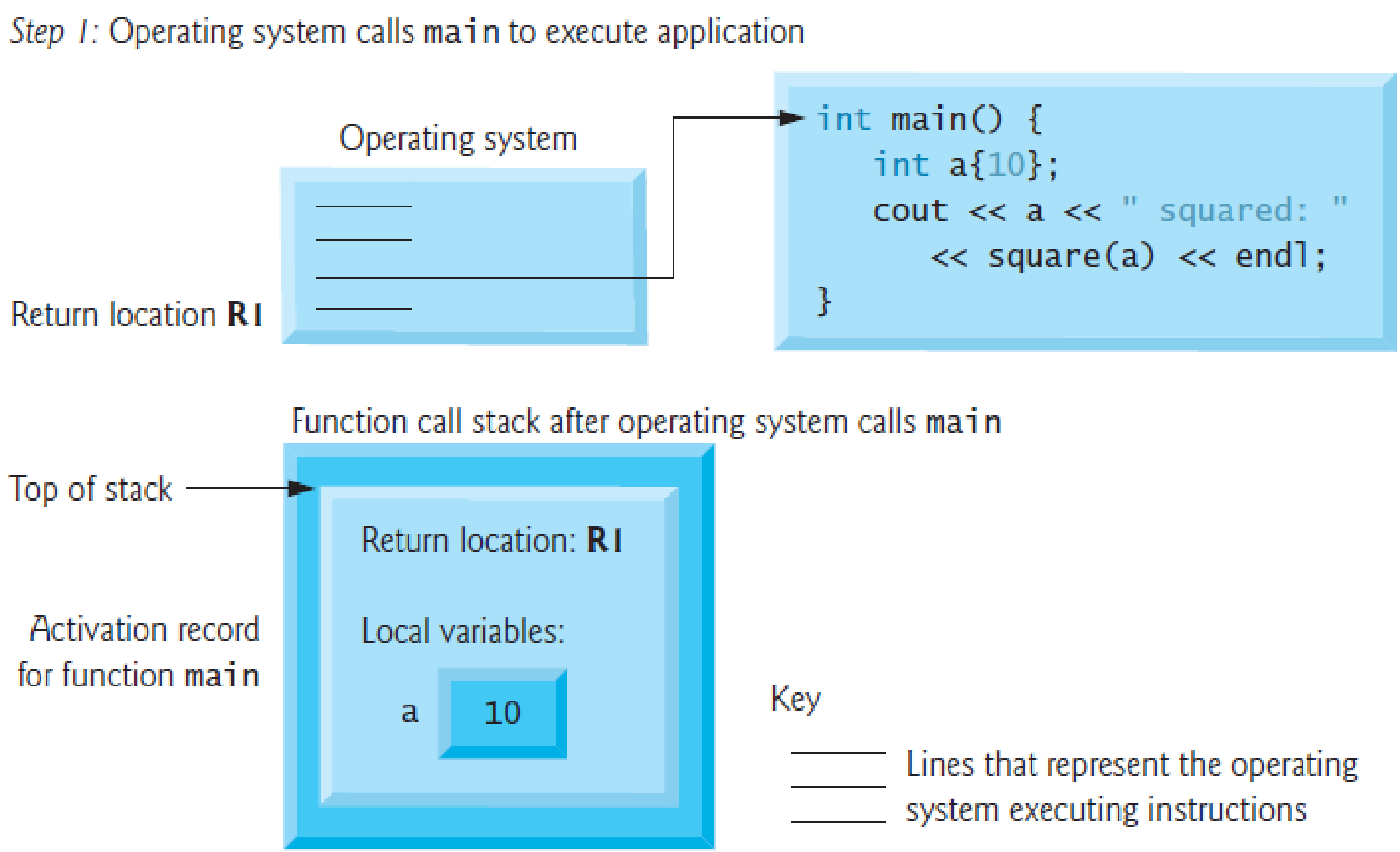
➢ If the called function returns instead of calling another function before returning, the stack frame for the function call is ***popped***, and *control transfers to the return address in the popped stack frame*.

Function Call Stack

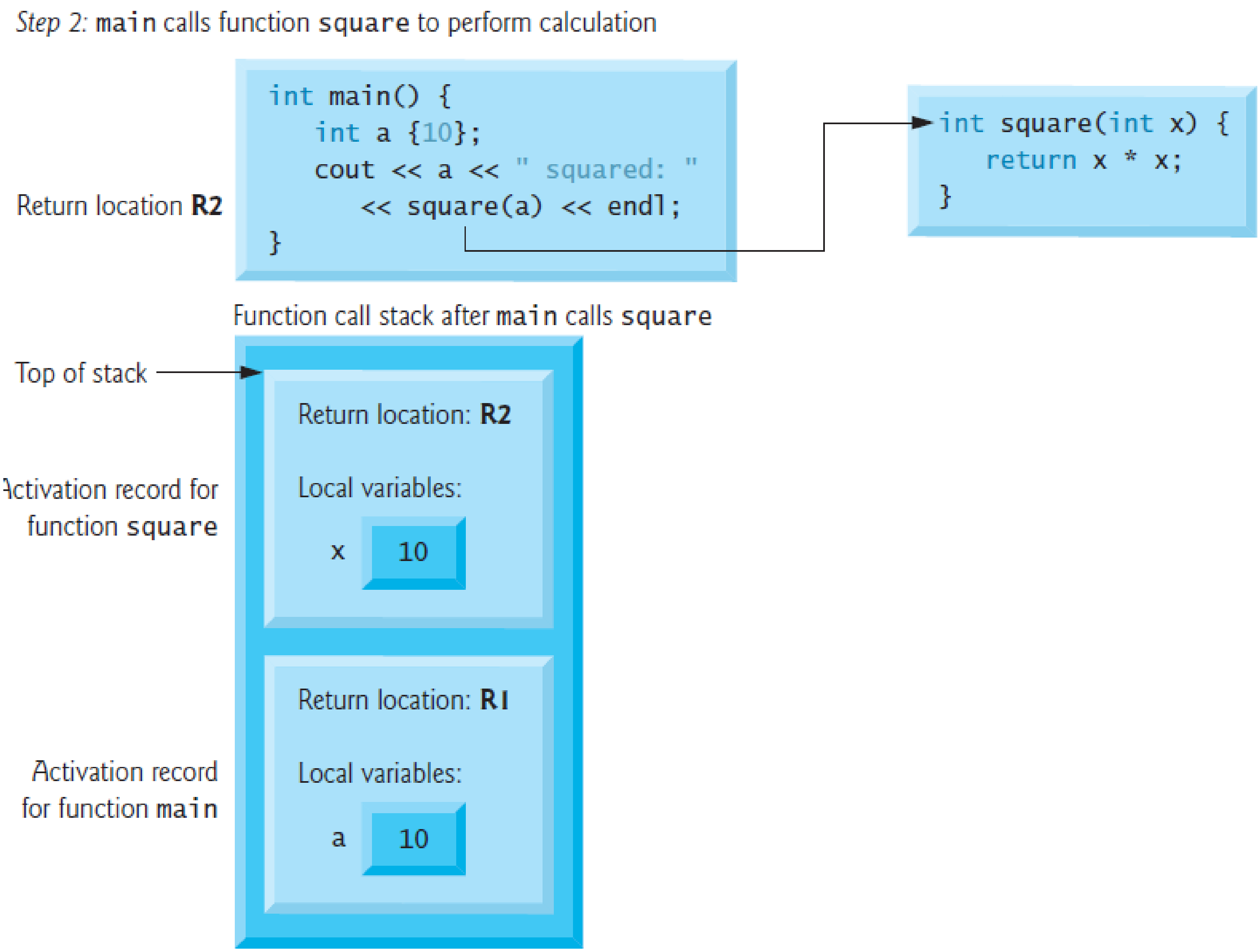
```
#include <iostream>
using namespace std;

int square(int); // prototype for function square

int main() {
   int a{10}; // value to square (local variable in main)

   cout << a << " squared: " << square(a) << endl; // display a squared
}

// returns the square of an integer
int square(int x) { // x is a local variable
   return x * x; // calculate square and return result
}
```

```
10 squared: 100
```

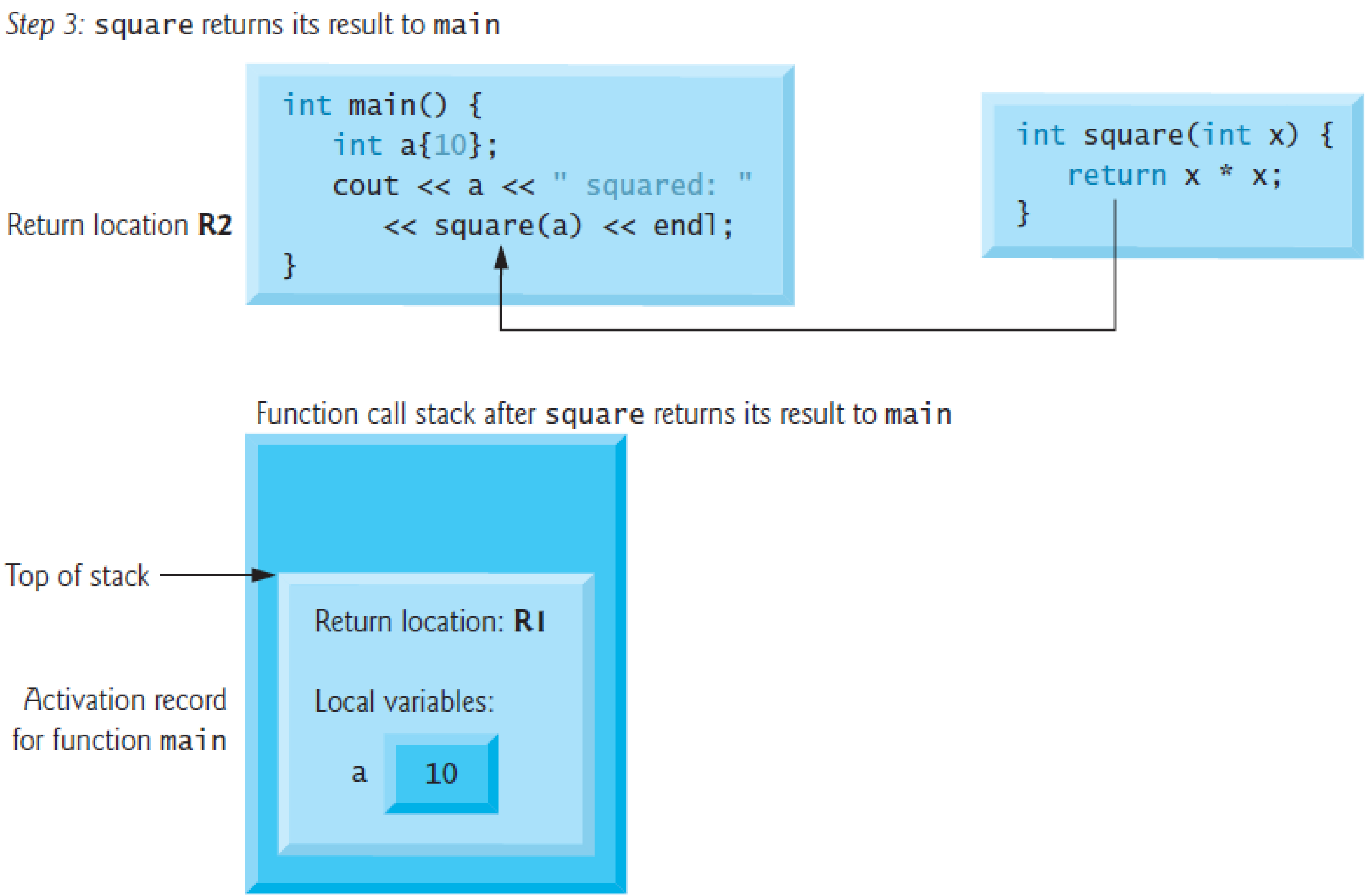
Function Call Stack



Function Call Stack



Function Call Stack



Function Call Stack

What is **stack overflow**?

- *The amount of memory* in a computer is **finite**, so only a certain amount of memory can be used to store activation records on the function-call stack.
- If more function calls occur than can have their activation records stored on the function-call stack, a fatal error known as **stack overflow** occurs.

- This is how the website **stackoverflow.com** got its name. This is a great website for getting answers to your programming questions.

Inline Functions

- Function calls involve *execution-time overhead*.
- 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 can ignore the inline qualifier and generally does so for all but the *smallest* functions.
- Reusable inline functions are typically placed in *headers*, so that their definitions can be included in each source file that uses them.

- *If you **change** the definition of an inline function, you must **recompile** all of that function’s clients.*
- *Compilers can inline code for which you have not explicitly used the inline keyword. Today’s optimizing compilers are so sophisticated that it’s best **to leave inlining decisions to the compiler.***

Inline Functions

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

// Definition of inline function cube. Definition of function appears
// before function is called, so a function prototype is not required.
// First line of function definition acts as the prototype.
inline double cube(const double side) {
   return side * side * side; // calculate cube
}

int main() {
   double sideValue; // stores value entered by user
   cout << "Enter the side length of your cube: ";
   cin >> sideValue; // read value from user

   // calculate cube of sideValue and display result
   cout << "Volume of cube with side "
      << sideValue << " is " << cube(sideValue) << endl;
}
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Inline Functions

```
inline double cube(const double side) {
   return side * side * side; // calculate cube
}
```

Why `double side` is const?

- This tells the compiler that the function does not modify variable *side.*
- This ensures that side's value is *not* changed by the function during the calculation.

❖ *The **const** qualifier should be used to enforce the principle of **least privilege**. Using this principle to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.*

References and Reference Parameters

Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**.

- When an argument is **passed by value**, a <u>**copy**</u> of the argument's value is made and passed (on the function-call stack) to the called function.
- With **pass-by-reference**, the caller gives the called function the ability **to access the caller's data directly, and to modify that data.**

- *One <u>disadvantage</u> of pass-by-value is that, if a <u>large data item</u> is being passed, <u>copying</u> that data can take a considerable amount of <u>execution time</u> and memory space.*

- *Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.*

- *Pass-by-reference can <u>weaken security</u>; the called function can corrupt the caller's data.*

References and Reference Parameters

A **reference parameter** is an alias for its corresponding argument in a function call.

```
int a = 1;
int& aRef = a; // ‘aRef’ is a reference to ‘a’
aRef++; // a=2, aRef=2
```

Passing Arguments by Value and by Reference (example)

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

int squareByValue(int); // function prototype (value pass)
void squareByReference(int&); // function prototype (reference pass)

int main() {
   int x{2}; // value to square using squareByValue
   int z{4}; // value to square using squareByReference
```

References and Reference Parameters

```

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue(x) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference(z);
   cout << "z = " << z << " after squareByReference" << endl;
}

// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue(int number) {
   return number *= number; // caller's argument not modified
}

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference(int& numberRef) {
   numberRef *= numberRef; // caller's argument modified
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

References and Reference Parameters

Const References

- Const reference specifies that the reference is not be allowed to modify the corresponding argument.

What will you modify here (class set/get methods)?

```
void setName(std::string accountName);
std::string getName() const;
```

References and Reference Parameters

Const References

- Const reference specifies that the reference is not be allowed to modify the corresponding argument.

What will you modify here?

```
void setName(std::string accountName);
std::string getName() const;
```

```
void setName(const std::string& accountName);
const std::string& getName() const;
```

References and Reference Parameters

❖ *When returning a reference to a* ***local variable****—unless that variable is declared* ***static****—the reference refers to a variable that's discarded when the function terminates. An attempt to access such a variable yields undefined behavior. References to undefined variables are called* ***dangling references****.*

```
int& getLocalVarReference() {
    int x = 5;
    return x;
}

int& danglingRef = getLocalVarReference();
danglingRef++;
std::cout << danglingRef << std::endl;
```

`Result:` Application crashed!

❖ *Returning a reference to a local variable in a called function is a logic error for which compilers typically issue a* ***warning****. Compilation warnings indicate potential problems, so most software-engineering teams have policies* ***requiring code to compile without warnings****.*

Default Arguments

- It's common for a program to invoke a function repeatedly with the same argument value for a particular parameter.

- In such cases, you can specify that such a parameter has a **default argument**, i.e., a default value to be passed to that parameter.

- When a program **omits** an argument for a *parameter with a default argument* in a function call, **the compiler rewrites the function call and inserts the default value of that argument**.

- Default arguments *must* be the rightmost (trailing) arguments in a function's parameter list.

- *Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.*

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

// function prototype that specifies default arguments
unsigned int boxVolume(unsigned int length = 1, unsigned int width = 1,
   unsigned int height = 1);

int main() {
   // no arguments--use default values for all dimensions
   cout << "The default box volume is: " << boxVolume();

   // specify length; default width and height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 1 and height 1 is: " << boxVolume(10);

   // specify length and width; default height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 1 is: " << boxVolume(10, 5);

   // specify all arguments
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 2 is: " << boxVolume(10, 5, 2)
      << endl;
}

// function boxVolume calculates the volume of a box
unsigned int boxVolume(unsigned int length, unsigned int width,
   unsigned int height) {
   return length * width * height;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Unary Scope Resolution Operator

C++ provides the **unary scope resolution operator** (::) to access a global variable when a local variable of the same name is in scope.

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

int number{7}; // global variable named number

int main() {
   double number{10.5}; // local variable named number

   // display values of local and global variables
   cout << "Local double value of number = " << number
      << "\nGlobal int value of number = " << ::number << endl;
}
```

```
Local double value of number = 10.5
Global int value of number = 7
```

❖ ***Always using*** *the unary scope resolution operator (::) to refer to global variables (even if there is no collision with a local-variable name) makes it clear that you're intending to access a global variable rather than a local variable.*