Chapter - 5 Arrays, Qualifiers and Reading Numbers

Arrays

Simple variables allow user to declare one item, such as a single width:

```
int width;   // Width of the rectangle in inches
```

If we have a number of similar items, we can use *an array* to declare them. For example, if we want declare a variable to hold the widths of 1000 rectangles.

```
int width_list[1000];   // Width of each rectangle
```

The width of the first rectangle is `width[0]` the width of the second rectangle is `width[1]` and so on until `width[999]`.

Warning:

> Common sense tells you that the last element of the width array is `width [1000]`. Common sense has nothing to do with programming and the last element of the array is `width[999]`.

Computing the average of 6 numbers

```
#include <iostream>


float data[5];  // data to average and total
float total;    // the total of the data items
float average;  // average of the items


int main()
{
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 46.5;
    data[3] = 82.0;
    data[4] = 22.0;


    total = data[0] + data[1] + data[2] + data[3] + data[4];
    average =  total / 5.0;
    std::cout << "Total " << total <<
                 " Average " << average << '\n';
    return (0);
}
```

C++ Strings

Bring in the string package using the statement:

```
#include <string>
```

Declaring a string

```
std::string my_name;   // The name of the user
```

Assigning the string a value:

```
my_name = "Oualline";
```

Using the “+” operator to concatenate strings:

```
first_name = "Steve"; last_name = "Oualline";
full_name = first_name + " " + last_name;
```

More on Strings

Extract a substring:

```
result = str.substr(first, last);
//     01234567890123
str = "This is a test";
sub = str.substr(5,6);


// sub == “123”
```

Finding the length of a string

```
string.length()
```

Wide strings contain wide characters. Example:

```
std::wstring funny_name;
// If you see nothing between the "" below then you
// don't have Chinese fonts installed
funny_name = L"瓣男桂";
```

Accessing characters in a string

You can treat strings like arrays, but this is not safe:

```
// Gets the sixth character
ch = str[5];
// Will not check to see if
// the string has 6 characters
```

Better (and much safer)

```
// Gets the sixth character
// Aborts program if
// there is no such character
ch = str.at(5);
```

Reading Data

The standard class `std::cout` is used with `<<` for writing data.

The standard class `std::cin` is used with `>>` for reading data.

```
std::cin >> price >> number_on_hand;
```

Numbers are separated by whitespace (spaces, tabs, or newlines).

For example, if our input is:

```
32 6
```

Then `price` gets 32 and `number_on_hand` gets 6.

Doubling a number

```
#include <iostream>
int   value;       // a value to double


int main()
{
    std::cout << "Enter a value: ";
    std::cin >> value;


    std::cout << "Twice " << value <<
           " is " << value * 2 << '\n';


    return (0);
}
```

Sample run

```
Enter a value: 12
Twice 12 is 24
```

Question: Why is `width` undefined?

```
#include <iostream>


int  height;  /* the height of the triangle
int  width;   /* the width of the triangle */
int  area;    /* area of the triangle (computed) */


main()
{
    std::cout << "Enter width height? ";
    std::cin >> width >> height;


    area = (width * height) / 2;
    std::cout << "The area is " << area << '\n';


    return (0);
}
```

Reading Strings

The combination of `std::cin` and `>>` works fine for integers, floating point numbers and characters. It does not work well for strings.

To read a string use the `getline` function.

```
std::getline(std::cin, string);
```

For example:

```
std::string name;    // The name of a person
std::getline(std::cin, name);
```

Initializing Variables

The new C++ style initialization:

```
int counter(0);    // number cases counted so far
```

The older C style syntax.

```
int counter = 0;    // number cases counted so far
```

Array initialization:

```
// Product numbers for the parts we are making
int product_codes[3] = {10, 972, 45};
```

Implied dimensioning of arrays:

```
// Product numbers for the parts we are making
int product_codes[] = {10, 972, 45};
```

Bounds Errors

Example:

```
int data[5];

result = data[99];      // Bad
```

Example of a bigger problem:

```
int data[5];

data[99] = 55;    // Very Bad
```

Modifies random memory.

C++ will not check for this!!

"assert" is your friend

The `assert` function checks to see if a condition is true. If it is not, the program is aborted.

Example:

```
#include <assert.h>

int main()
{
    int i = 2;
    assert(i == 3);
    return (0);
}
```

Protecting arrays with `assert`

Example:

```
#include <assert.h>
int data[5];
int index;


int main()
{
    index = 5;


    assert(index >= 0);
    assert(index < 5);   // Not the best way of doing it


    index = data[index];
```

Using `sizeof` to automatically compute the array limit.

The `sizeof` function returns the number of bytes allocated to a variable.

Definitions:

`sizeof(array)` Number of bytes in an array

`sizeof(array[0]);` Number of bytes in an element of the arrays

Therefore

```
number_of_elements =
   sizeof_array_in_bytes / sizeof_element_in_bytes
```

In C++:

```
assert(index >= 0);
assert(index < (sizeof(data) / sizeof(data[0])));


index = data[index];
```

Multiple Dimensional Arrays

```
type variable[size1][size2]; // comment
```

Example:

```
// a typical matrix
int matrix[2][4];
```

Notice that C++ does **not** follow the notation used in other languages:

```
matrix[10,12] // Not C++
```

To access an element of the `matrix` we use the notation:

```
matrix[1][2] = 10;
```

More than two dimensions can be used:

```
four_dimensions[10][12][9][5];
```

Initializing Matrices

```
// a typical matrix
int matrix[2][4] =
    {
        {1, 2, 3, 4},
        {10, 20, 30, 40}
    };
```

This is shorthand for:

```
matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[0][2] = 3;
matrix[0][3] = 4;


matrix[1][0] = 10;
matrix[1][1] = 20;
matrix[1][2] = 30;
matrix[1][3] = 40;
```

Question: Why does this program produce funny answers?

```
#include <iostream>


int array[3][5] = {     // Two dimensional array
   { 0,  1,  2,  3,  4 },
   {10, 11, 12, 13, 14 },
   {20, 21, 22, 23, 24 }
};


int main()
{
    std::cout << "Last element is " <<
                 array[2,4] << '\n';
    return (0);
}
```

When run on a Sun 3/50 this program generates:

```
Last element is 0x201e8
```

C Style Strings

C Style Strings are constructed from arrays of characters.

```
// A string of up to 99 characters
char a_string[100];
```

Strings end in the special character `'\0'` (NUL).

```
a_string[0] = 'S';
a_string[1] = 'a';
a_string[2] = 'm';
a_string[3] = '\0'; // End the string
```

The variable `a_string` contains the string `"Sam"`.

Note: `a_string` now holds a string of length 3. It can hold any length string up to 99 characters long. (One character must be reserved for the end-of-string marker `'\0'`.)

Question

Are all “strings” “arrays of characters”?

Are all “character arrays” “strings”?

Using C Style Strings

String constants are enclosed in double quotes. Example: `"Sam"`.

Strings can not be directly assigned.

```
a_string = "Sam";    // Illegal
```

The standard function `std::strcpy` can be used to copy a string.

```
#include <cstring>
// ....
std::strcpy(a_string, "Sam"); // Legal.
                              // But dangerous
```

Note: `#include <cstring>` tells C++ that we are using the standard string package.

Standard C Style String Functions

Function	Description
`std::strcpy(string1, string2)`	Copies `string2` into `string1`. (Unsafe)
`std::strncpy(string1, string2, length)`	Copies `string2` into `string1` but limit the number of characters copied (including the end of string) to `length`. (Safer)
`std::strcat(string1, string2)`	Concatenates `string2` onto the end of `string1`. (Unsafe)
`std::strncat(string1, string2, length)`	Concatenates `string2` onto the end of `string1`. Limit the number of characters added to `length`. Does not guarantee that an end of string will be copied. `(Safer)`
`length = std::strlen(string)`	Gets the length of a string. (Safe)
`std::strcmp(string1, string2)`	0 if `string1` **equals** `strings2`, otherwise non-zero. (Safe)

Using `std::strcpy`

```
#include <iostream>
#include <cstring>

char name[30];  // First name of someone

main()
{
   std::strcpy(name, "Sam");
   std::cout << "The name is " <<
           name << '\n';
   return (0);
}
```

Combining Two Names

```
#include <cstring>
#include <iostream>
char first[100];        // first name
char last[100];         // last name
char full_name[100];    // full version of first and last name
int main()
{
    std::strcpy(first, "Steve");    // Initialize first name
    std::strcpy(last, "Oualline");  // Initialize last name


    std::strcpy(full_name, first);  // full = "Steve"
    // Note: strcat not strcpy
    std::strcat(full_name, " ");    // full = "Steve "
    std::strcat(full_name, last);   // full = "Steve Oualline"


    std::cout << "The full name is " << full_name << '\n';
    return (0);
}
```

Outputs:

```
    The full name is Steve Oualline
```

Initializing Strings

```
char name[] = {'S', 'a', 'm', '\0'};
```

C++ has a special shorthand for initializing strings, by using double quotes (") to simplify the initialization.

```
char name[] = "Sam";
```

The dimension of `name` is 4, because C++ allocates a place for the '\0' character that ends the string.

Note:

```
char string[50] = "Sam";
```

Declares a string variable that can hold strings that are 0 to 49 characters long, but initializes the string to a 4 character string.

The statement initializes only 4 of the 50 values in `string`. The other 46 elements are not initialized and may contain random data.

Finding the length of C Style string

```
#include <cstring>
#include <iostream>


char line[100]; // A line of data


int main()
{
    std::cout << "Enter a line:";
    std::cin.getline(line, sizeof(line));


    std::cout << "The length of the line is: " <<
                 std::strlen(line) << '\n';
    return (0);
}
```

When we run this program we get:

```
Enter a line:test
The length of the line is: 4
```

Question: What is the size of `line` and what is the length of `line`? What's the difference?

Safe C Style Strings

Safe copy

```
assert(sizeof(name) >= sizeof(“Oualline”));
std::strcpy(name, “Oualline”);


assert(sizeof(name) > std::strlen(first_name));
std::strcpy(name, first_name);


std::strncpy(name, last_name, sizeof(name)-1);
```

Safe concatenation:

```
std::strncat(name, last_name,
   sizeof(name) – strlen(name) -1);
name[sizeof(name)-1] = '\0';
```

Reading C Style Strings

```
char name[50];

// ....

std::getline(std::cin,
             name, sizeof(name);
```

Converting between string types

```
char c_string[100];
std::string cpp_string;
```

C Style string => C++ String -- Just assign

```
cpp_string = c_string;
```

C++ String => C String – use the `c_str` function call

```
strncpy(c_string, cpp_string.c_str(),
   sizeof(c_string));
```

String differences

	C++ Strings	*C Strings*
Memory Allocation	Automatic	Manual
Length	Variable	Limited
Safety	Good	Bad
Efficency / Speed	Medium	Fast

Types of Integers

Integers come in various flavors:

int Normal storage used

long int Extra storage may be used

Long Integer constants are specified with "L" at the en**d**

```
// Amount in account (in cents)
long int amount = 12345L;
```

short int

Reduced storage may be used

signed Numbers can be positive or negative (the default)

unsigned

Only positive numbers allowed.

Very Short Integers

Character variables can be used to store very short integers (in the range from -128 to 127 (signed) or 0 to 255 (unsigned)).
Example:

```
// If set, pre-process the input
unsigned char flag = 1;
```

Question: Is the following character variable signed or unsigned?

```
char foo;
```

Answers:

a. It's signed.
b. It's unsigned.
c. If we always specify **signed** or **unsigned** we don't have to worry about problems like this.

Reading and Writing Very Short Integers

Writing very short integers can be done by using the `static_cast<int>` operation.

```
  unsigned char flag = 1;

std::cout << "Flag is " <<
  static_cast<int>(flag) << "\n";
```

Reading of very short integers can not be done directly. You must read an integer and assign it to a very short integer.

Types of Floating Point numbers

float

Normal floating point number. (Default range and precision.)

double

Double precision (and double range) floating point number.

long double

(Not-standard. Available only on a few compilers.)

Extended precision and range.

Constant Declarations

```
// The classic circle constant
const float PI = 3.1415926;
```

Note:
By convention variable names use lower case only names while constants use upper case only. However there is nothing in the language that requires this and several programming systems use a different convention.

A constant can not be changed:

```
PI = 3.0;       // Illegal
```

Integer constants can be used as a size parameter when declaring an array.

```
// Max. num. of elements in total list.
const int TOTAL_MAX = 50;
// Total values for each category
float total_list[TOTAL_MAX];
```

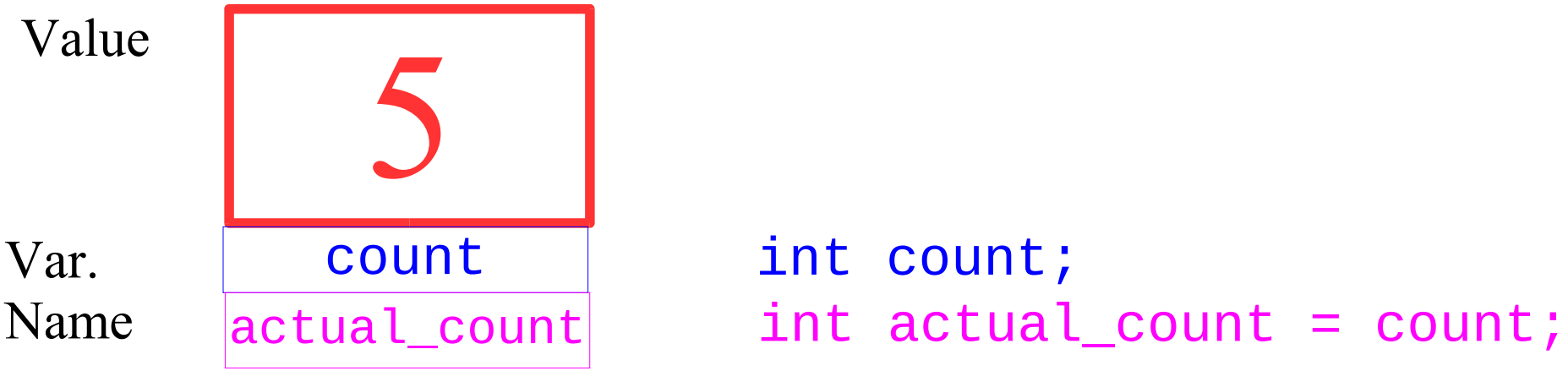
Reference Declarations

Reference declarations allow you to give another name to an existing variable (an alias.)

Example:

```
int count;          // Number of items so far
int &actual_count = count;
// Another name for count
```

From now on `count` and `actual_count` are the *same* variable. Anything done to `count` is reflected in `actual_count`.



Qualifiers

The complete list of qualifiers

Special	Class	Size	Sign	Type
volatile	**register**	**long**	**signed**	**int**
<blank>	**static**	**short**	**unsigned**	**float**
	extern	**double**	<blank>	**char**
	auto	<blank>		<blank>
	<blank>			

Special

violate	Indicates a special variable who's value may change at any time. (Used in specialized programming not covered by this course.)
<blank>	Normal variable.

Variable Class

register	This indicates a frequently used variable that should be kept in a machine register.
static	The meaning of this word depends on the context.
extern	The variable is defined in another file.
auto	A variable allocated from the stack. This keyword is hardly ever used.
<blank>	Indicates that the default class (**auto**) is selected.

Size

long	Indicates a larger than normal integer. (Some non-standard compilers use long double to indicate a very large floating point variable).
short	A smaller than normal integer.
double	A double size floating point number.
<blank>	Indicates a normal size number.

Sign

signed Values range from the negative to the positive. Always true for floating point numbers.

unsigned

Positive numbers only allowed.

<blank>

For integers defaults to signed.

Character variables may be **signed**, **unsigned** or <blank>. These are three differ-ent types of variables and may not be mixed. The <blank> indicator should be use for character variables which will hold only characters instead of very short integers. For very short integers, you should always specify **signed** or **unsigned**.

Type

int	Integer.
float	Floating point numbers.
char	Single characters, but can also be used for very short integers.

Hexadecimal and Octal Constants

Hexadecimal constants (base 16) begin with 0x. (0x12)

Octal constants (base 8) begin with a leading 0. (012)

Base 10	Base 8	Base 16
6	06	0x6
9	011	0x9
15	017	0xF

Question: Why does the following program fail to print the correct zip code? What does it print instead?

```
long int zip;        // Zip code


int main()
{
   zip = 02137L;    // Use the Zip Code for Cambridge MA


   std::cout <<"New York's Zip code is: "<< zip << '\n';
   return(0).
}
```

Shortcut operators

The code:

```
total_entries = total_entries + 1;
```

Can be replaced by:

```
++total_entries;
```

Similarly:

```
total_entries = total_entries - 1;
```

Can be replaced by:

```
--total_entries;
```

Also

```
total_entries = total_entries + 2;
```

is the same as

```
total_entries += 2;
```

Shorthand Operators

Operator	Shorthand	Equivalent Statement
+=	`x += 2;`	`x = x + 2;`
-=	`x -= 2;`	`x = x - 2;`
*=	`x *= 2;`	`x = x * 2;`
/=	`x /= 2;`	`x = x / 2;`
%=	`x %= 2;`	`x = x % 2;`

Side Effects

A side effect occurs when you have a statement that performs a main operation *and* also another operation:
Example:

```
size = 5;
result = ++size;
```

The first statement assigns `size` the value of 6. The second statement:

1. Increments `size`. (side effect).
2. Assigns `result` the value of `size` (main operation).

Do not use side effects. They confuse the code, add risk to your program and in general, cause a lot of trouble. We are after clear code, not clever compact code.

Problems with side effects

```
value = 1;
result = (value++ * 5) + (value++ * 3);
```

This expression tells C++ to perform the steps:

a. Multiply `value` by 5, add 1 to `value`.

b. Multiply `value` by 3, add 1 to `value`.

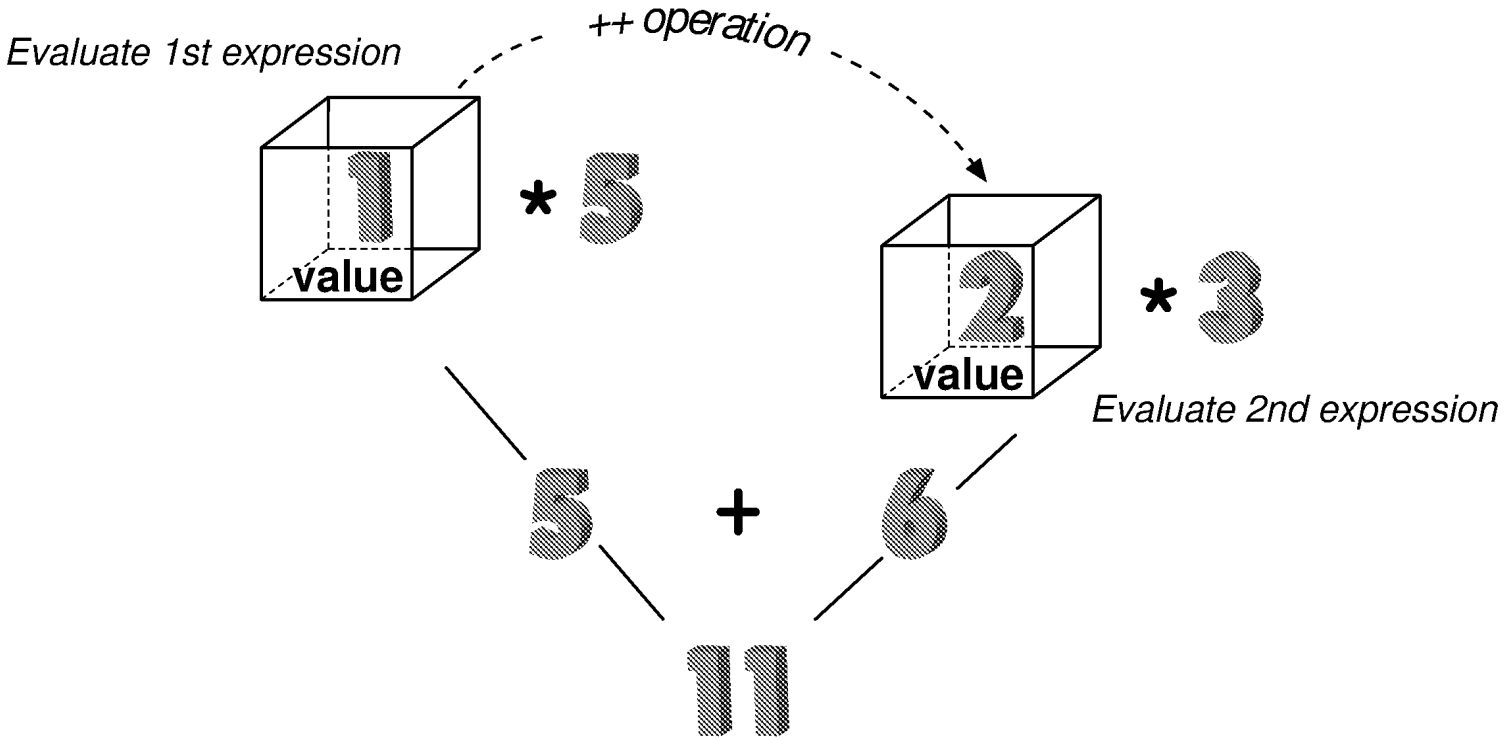
c. Add the results of the two multiplies together.

But in what order?

Steps a. and b. are of equal priority so the compiler can execute them in any order it wants to.

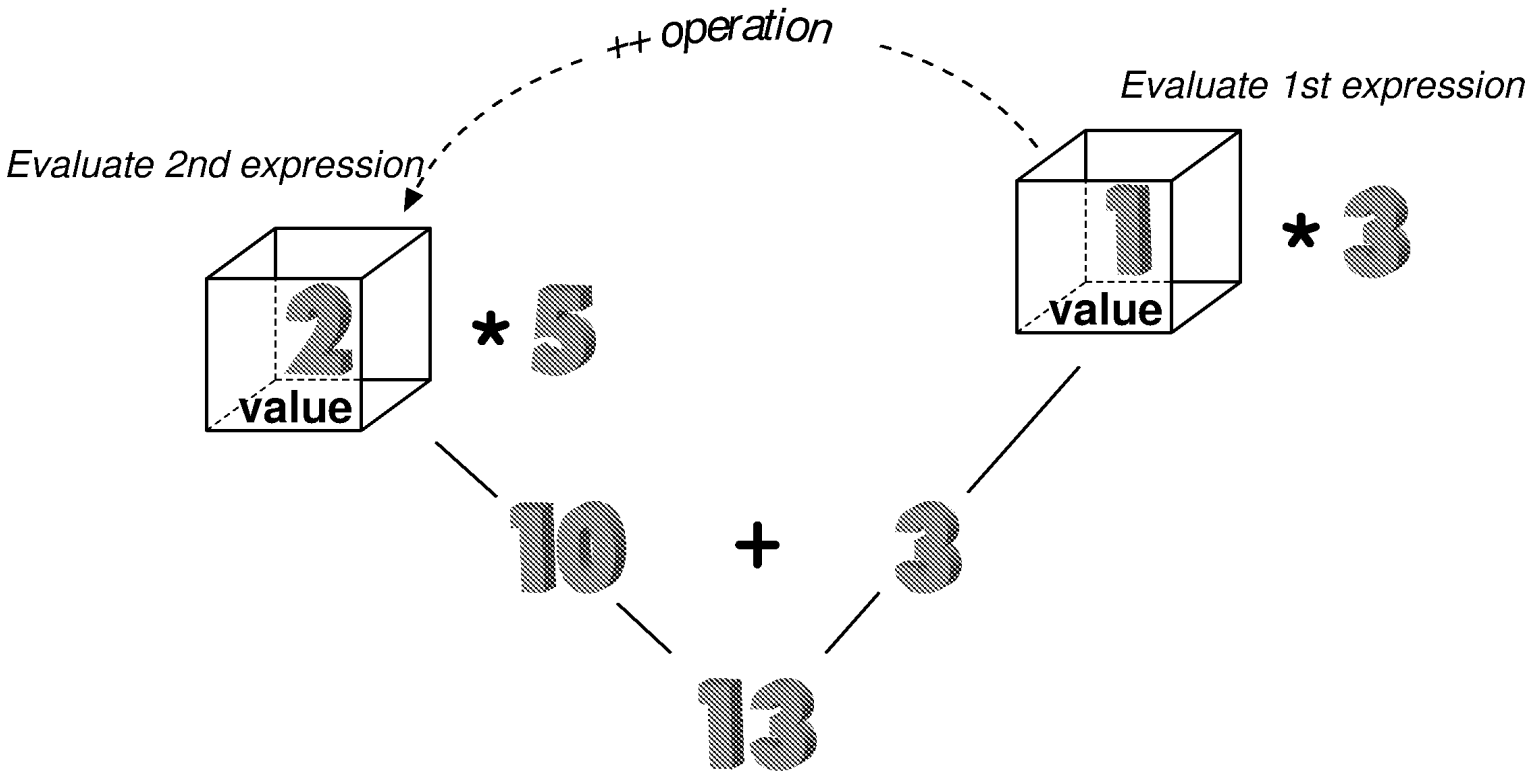
"a" first

```
result  =  (value++ * 5)  +  (value++ * 3);
```



"b" First

```
result  =  (value++ * 5)  +  (value++ * 3);
```



Final Warning

We've not discussed all of the problems that side effects can cause. We'll see how side effects can cause havoc when we study the pre-processor. The simple rule is:

Put ++ and -- on lines by themselves.

This avoids a tremendous amount of risk. Your programs have enough problems without your playing with fire.