# Chapter 8

# Arithmetic in C++

# 8.1 The C++ Vocabulary

| asm | do | if | reinterpret_cast | try |
|---|---|---|---|---|
| auto | double | inline | return | typedef |
| bool | dynamic_cast | int | short | typeid |
| break | else | long | signed | typename |
| case | enum | main | sizeof | union |
| catch | explicit | mutable | static | unsigned |
| char | export | namespace | static_cast | using |
| class | extern | new | struct | virtual |
| const | false | operator | switch | void |
| const_cast | float | private | template | volatile |
| continue | for | protected | this | wchar_t |
| default | friend | public | throw | while |
| delete | goto | register | true | |

**types — type qualifiers — execution control structures**

# 8.2 Variables and Types

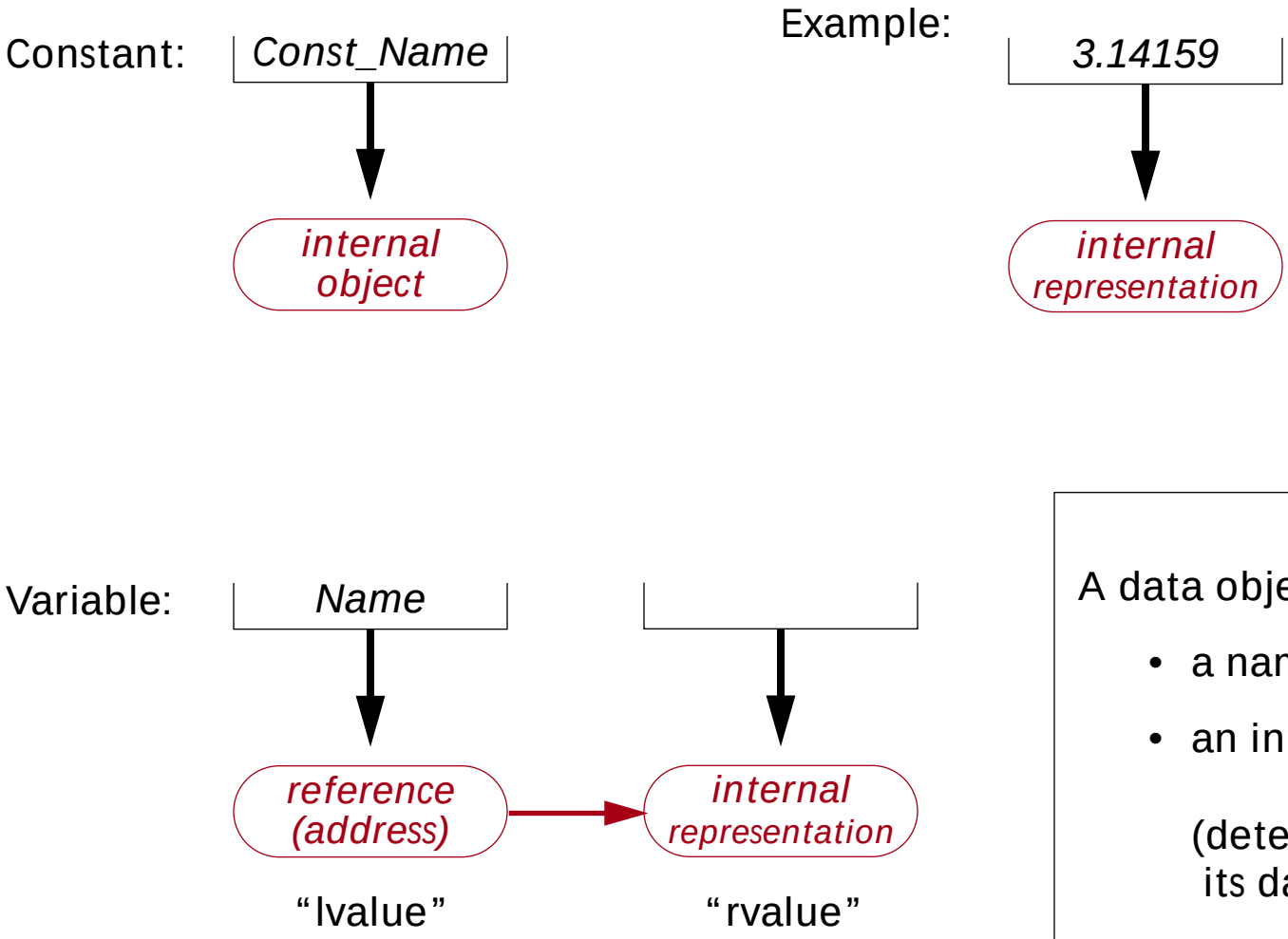
## 8.2.1 Data Objects

The **operational model** of computing / programming considers a programs as

a sequence of instructions for **manipulating data objects.**

Data objects can be distinguished by their **reference levels**:

| | | |
|---|---|---|
| 0 | Constant | an actual value (a number, a character, ...) |
| 1 | Variable | contains a value (= constant) |
| 2 | Level-1 pointer | refers to a variable, points to a variable |
| 3 | Level-2 pointer | refers to a level-1 pointer |
| 4 | ... | ... |




A data objects has …

- a name
- an internal object

(determined by its data type)

## 8.2.2 Variables

Variables are data objects with the following properties:

- **Identifier**, **name**
  - Defined in its declaration (example: int size = 12)
- **Type**
  - Defined in its declaration (example: int size = 12)
- **Internal object** = address of the manipulable memory section (**lvalue**)
- **Value** = alterable contents of the addressed memory section
  - Its interpretation depends on its type.
  - Its value can be changed by assignments (=) or specific operators (++).
- **Lifespan**:
  - Dynamically defined by the program structure (see: functions).

## 8.2.3 Declaration of Variables

*basic_variable_declaration* ::= *type* {*name*,}* *name*;

```
bool correct_input;  // true or false
int number_of_items; // number of items
float width, height; // width and height
```

### Initializing Variables

*basic_variable_declaration* ::= *type* {*name_init*,}* *name_init*;

*name_init* ::= { *name* | *name* ( *init* ) | *name* = *init* }

```
int counter(0);  // C++ initialization
int counter = 0; // older C style syntax
```

# 8.3 Elementary C++ Data Types

- **Integers**

```
int normal_integer;
long int long_integer;
short int short_integer;
unsigned short int unsigned_short_int;
```

- **Floating Point Numbers**

```
float normal_float;
double double_float;
long double high_precision_float;
```

- **Characters**

```
char single_character;
```

## 8.3.1 Integers

Integers (= whole numbers) have no fractional part or decimal point.

### Range of values

| 32 bit = 4 bytes (UNIX, Mac) | 16 bit = 2 bytes (PC DOS) |
|---|---|
| $-2^{31}$ to $2^{31}-1$ | $-2^{15}$ to $2^{15}-1$ |
| -2,147,483,648 to 2,147,483,647 | -32,768 to 32,767 |

### Types of Integers

- `int`: the most efficient size for the machine is chosen (2 or 4 bytes)
- `long int`: extra storage for integer (4 bytes guaranteed)
- `short int`: reduced storage (2 bytes)
- `unsigned`: uses all bits for the number (for 1 byte: range 0 to 65,535)

## 8.3.2 Floating Point Numbers

Floating point numbers have a decimal point or mantissa and exponent specification.

- `3.1419`
- `0.5`
- `1.2e34` ( = $1.2 * 10^{34}$)

### Range of values

The range of floating point numbers (which are always signed) and their accuracy varies widely among different computer and operating systems.[1]

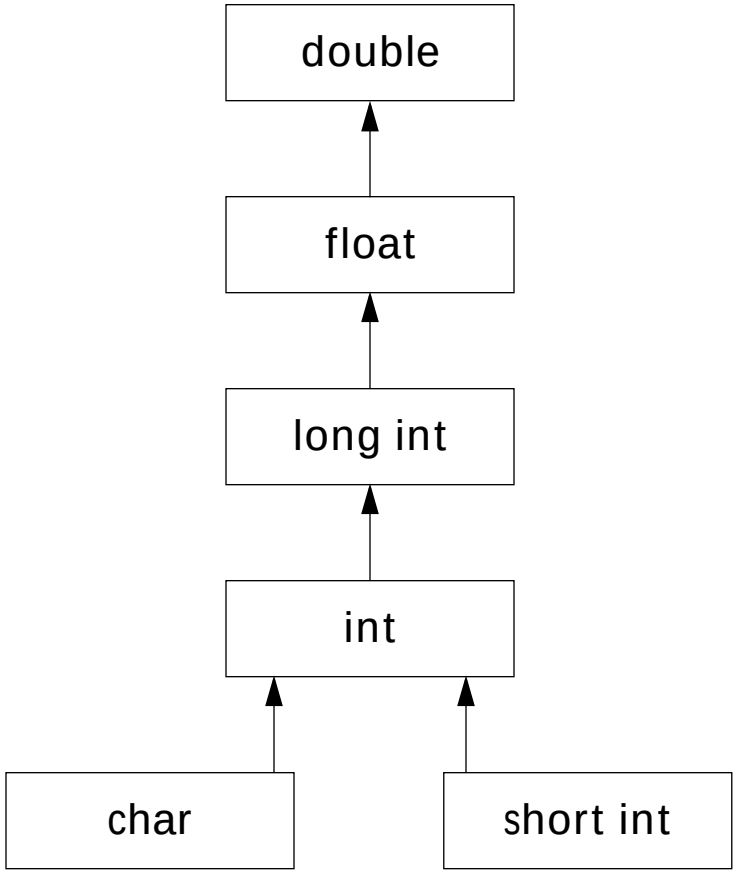
1. For further details consult: S. Qualline, *Practical C++ Programming*, O´Reilly, 1997, Chapter 19.

## Types of Floats

- `float`: normal precision (usually 4 bytes = 32 bits)
- `double`: twice the range and precision of `float` (usually 8 bytes = 64 bits)
- `long double`: extended precision (at least 8 bytes)

## 8.3.3 Type Conversions in Expressions

If an expression (see chapter 8.4) contains perands of different types, an **automatic type conversion** (to the type which is highest in the following hierarchy) is performed.




*This diagram is a simplified version of all possible type conversions.*

## Examples of binary operators and implicit type conversions

| Operator | Meaning | Integer example | Floating-point example |
|---|---|---|---|
| + | Addition | 5 + 2 ⇒ 7 | 5.5 + 2.2 ⇒ 7.7 |
| - | Subtraction | 5 -2 ⇒ 3 | 5.5 - 2.2 ⇒ 3.3 |
| * | Multiplication | 5 * 2 ⇒ 10 | 5.5 * 2.2 ⇒ 12.1 |
| / | Division | 5 / 2 ⇒ 2 | 5.5 / 2.2 ⇒ 1.375 |
| % | Modulus, or remainder | 5 % 2 ⇒ 1 | 5.5 % 2.2 ⇒ *type error* |

| Operator | Meaning | Integer / floating-point example |
|---|---|---|
| + | Addition | 5 + 2.2 ⇒ 7.7 |
| - | Subtraction | 5.0 - 2 ⇒ 3.0 |
| * | Multiplication | 4 * 2.2 ⇒ 8.8 |
| / | Division | 6.0 / 2 ⇒ 3.0 |
| % | Modulus, or remainder | 5.0 % 2.0 ⇒ *type error* |

## 8.3.4 Qualifiers for Variable Declarations[1]

*variable_declaration* ::= [ *special* ] [ *class* ] [ *size* ] [ *sign* ] *type*
{*name_init*,}* *name_init*;

| Special | Class | Size | Sign | Type |
|---|---|---|---|---|
| `volatile` | `register` | `short` | `signed` | `int` |
| <blank> | `static` | `long` | `unsigned` | `float` |
| | `extern` | <blank> | <blank> | `double` |
| | `auto` | | | |
| | <blank> | | | `(bool)` |

1. S. Qualline, *Practical C++ Programming*, O´Reilly, 1997, p. 76.

## Special:

- `volatile`: special variable whose value may change at any time
- <blank>: normal variable

## Class:

- `register`: frequently used variable, kept in a machine register
- `static`: meaning depends on context
- `extern`: defined in another file
- `auto`: variable allocated from the stack
- <blank>: `auto` default class is selected

## Size:

- `long`: larger than normal integer or very large float
- `short`: smaller than normal integer
- <blank>: normal size number (integer or float)

## Sign:

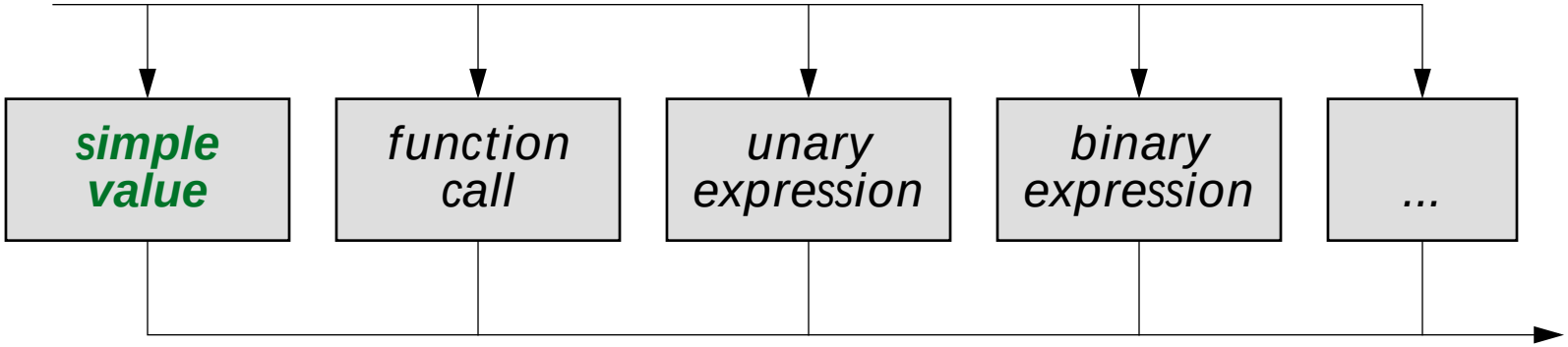
- `signed`: signed integer or character
- `unsigned`: use all integer or character bits (no sign bit)

## Type:

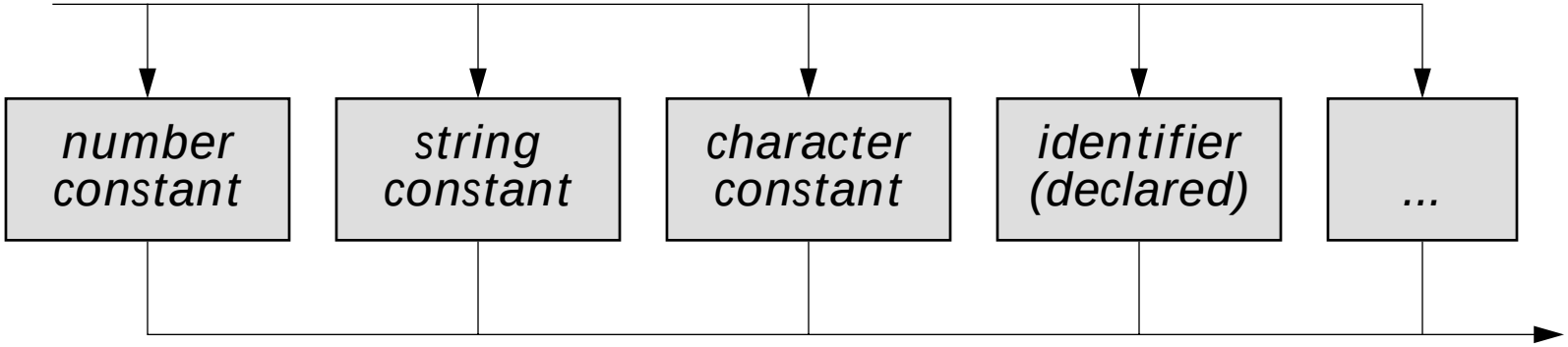
- `int`: integer
- `float`: floating point number
- `double`: double-size float
- `char`: single character, also used for very short integer
- `bool`: Boolean value, `true` or `false`

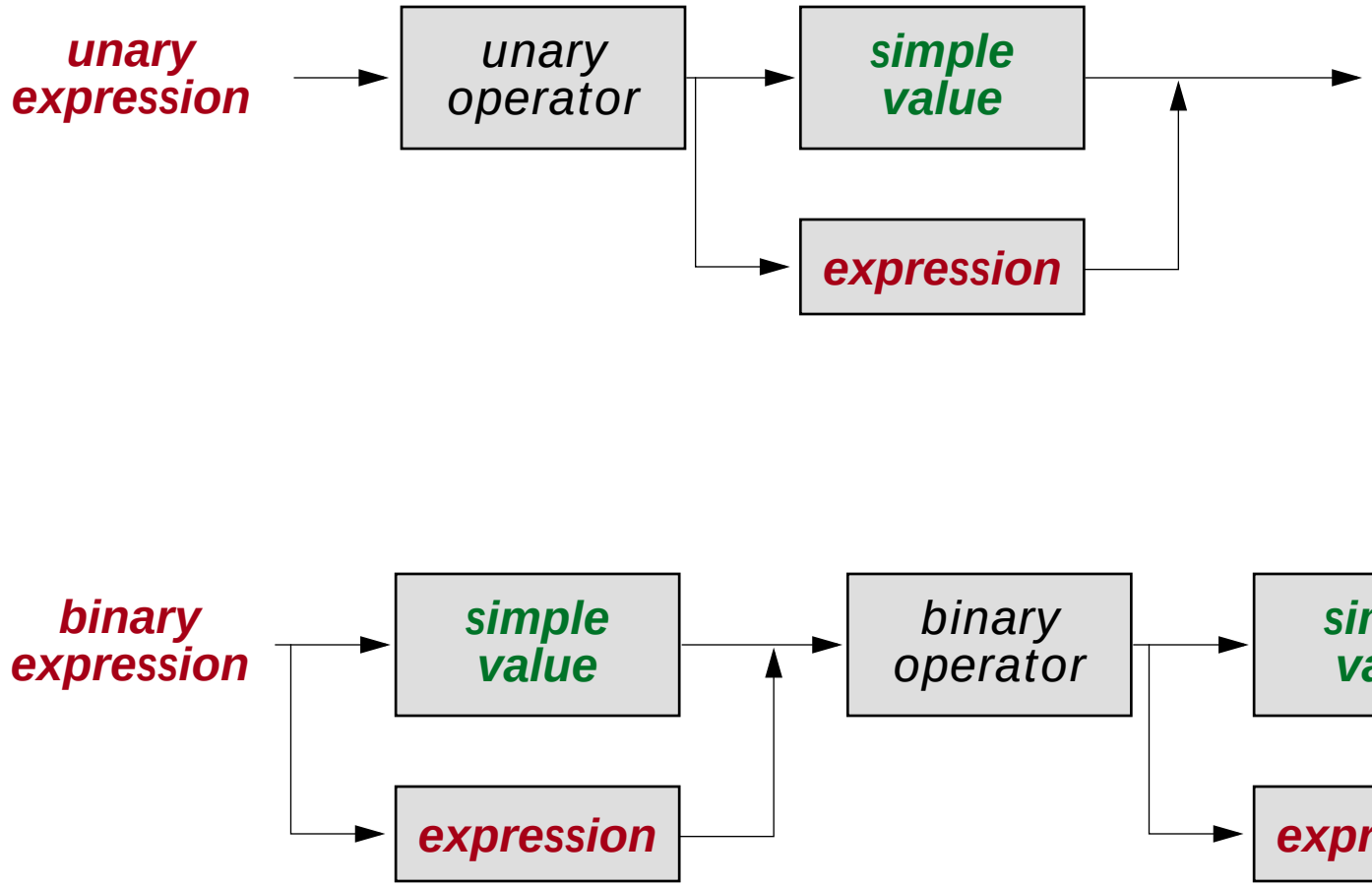
# 8.4 Arithmetic Expressions

***expression***

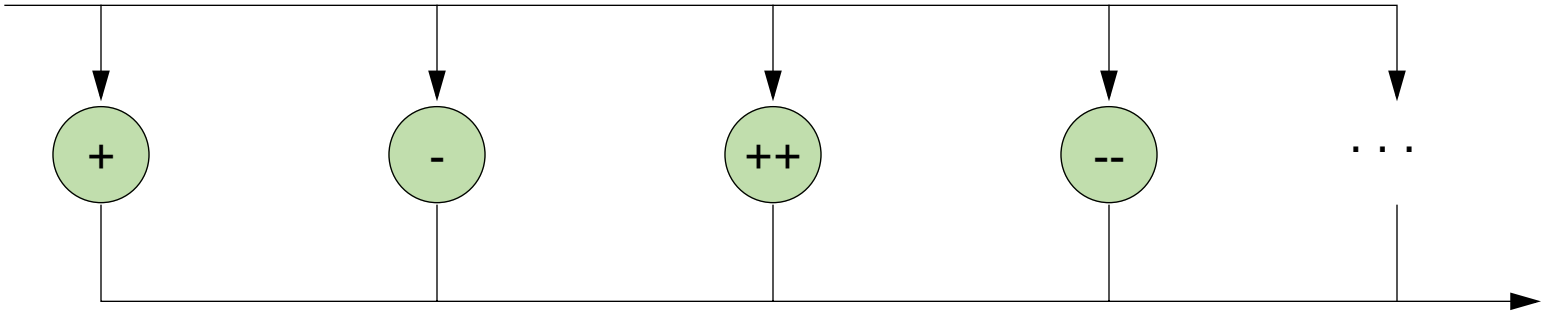



***simple value***

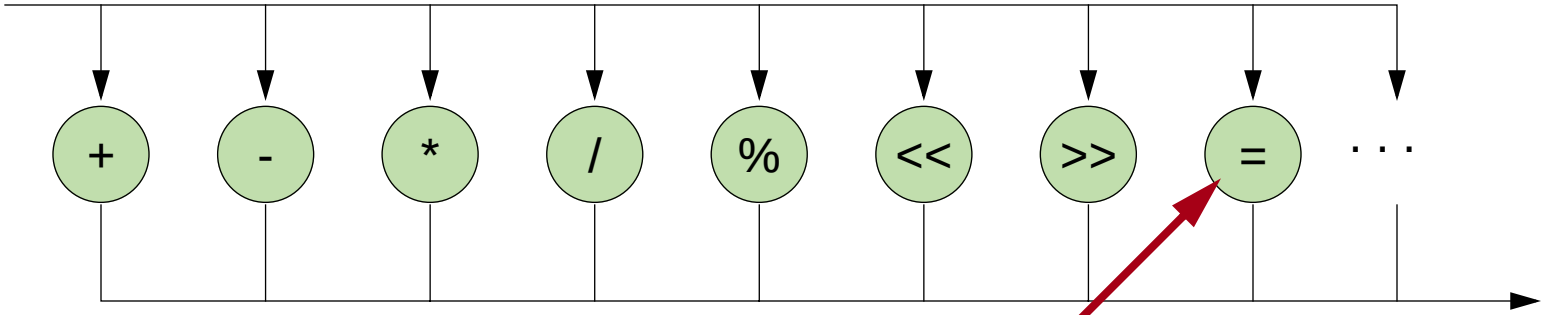



unary expression
unary operator
simple value
expression
binary expression
simple value
binary operator
simple value
expression
expression

***unary operator***




***binary operator***




For assignments the *simple value* must be an identifier!

# 8.4.1 Standard Arithmetic Operators

## Arithmetics

| | | | | | |
|---|---|---|---|---|---|
| **op** | *int* | ⇒ | *int* | | +, - |
| **op** | *float* | ⇒ | *float* | | +, - |
| *int* | **op** | *int* | ⇒ | *int* | +, -, *, /, % |
| *float* | **op** | *float* | ⇒ | *float* | +, -, *, / |

## Comparisons

| | | | | | |
|---|---|---|---|---|---|
| *int* | **op** | *int* | ⇒ | *int* | ==, !=, <=, <, >, >= |
| *float* | **op** | *float* | ⇒ | *float* | ==, !=, <=, <, >, >= |

## Logical Operators

**op** *int* ⇒ *int* ! (*not*)

*int* **op** *int* ⇒ *int* && (*and*), || (*or*)

## Bitwise Logical Operators

**op** *int* ⇒ *int* ~ (*not*)

*int* **op** *int* ⇒ *int* & (*and*), | (*or*), ^ (*xor*),
>> (*right shift*), << (*left shift*)

| **x** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| **~x** | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

| **x** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|
| **x << 2** | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

## The ? Operator

*condition* **?** *true_expression* **:** *false_expression*

Example:

```
int x = 15;
cout << (x < 7 ? 4 : 2); // Output: 2
```

## The Comma Operator: Concatenation of Two Expressions

*expression*$_1$ , *expression*$_2$

Evaluates *expression*$_1$ and then *expression*$_2$, and finally returns the value of *expression*$_2$ as the result.

Example:

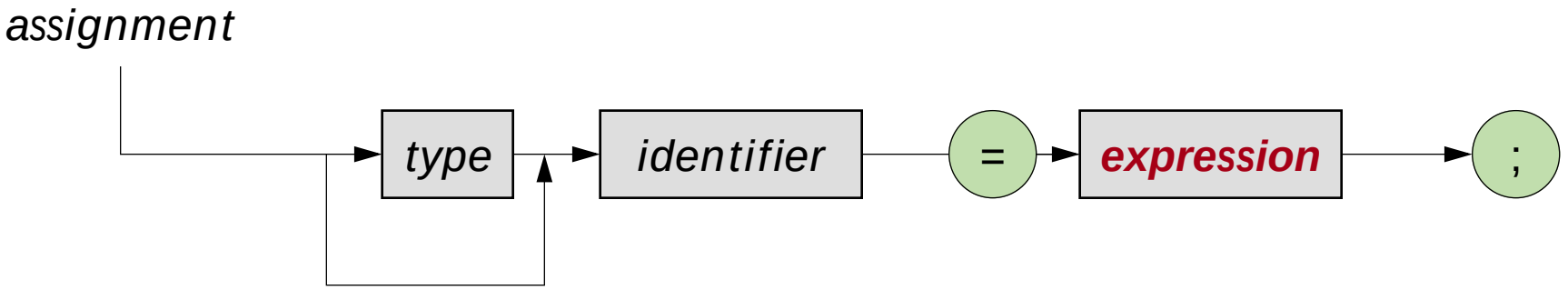
```
int x = 0, y = 1;
cout << (x = x + y, y = y * 2);
```

More useful in loops (see later):

```
for (i=1, j=9; ...; i = i+2, j = j-3)
{ ... // loop body }
```

## 8.4.2 Assignments and Compound Assignments

### Syntax Diagram for Assignments




```
int pounds = 0;
float kilos = pounds / 2.2;

int x, y, z; int f(int n);
...
x = -y * (z++ - f(4));
```

## Compound Assignments

A common operation in programming is to apply an operator to a variable, and then store the result in the same variable.

For example, the following assignment doubles the value of j:

```
j = j * 2;
```

This can be rewritten as:

```
j *= 2;
```

This works for the following operators in C++ :

```
+=, -=, *=, /*, %=, <<=, >>=, &=, ^=, |=
```

## Special Case: Increments and Decrements by One

For increasing or decreasing a variable value by 1, there is even a shorter notation in C++.

Instead of writing

```
k = k + 1;
```

one can write

```
k++; // return value of k, then increment
```

or

```
++k; // increment, then return value of k
```

This also works for the decrement operator '--':

```
k--; // or --k; equivalent to k = k -1;
```

Examples:

```
a = 1;              // a = 1
b = ++a;            // a = 2, b = 2
c = a++;            // a = 3, c = 2

a = 5; b = 3;
n = ++a + b--; // a = 6, b = 2, n = 9

a = 5; b = 3;
n = ++a * ++b; // a = 6, b = 4, n = 24
n = a++ * b++; // a = 6, b = 4, n = 15

a = 5; b = 3;
n = a++ * --b; // a = 6, b = 2, n = 10
```

## 8.4.3 Operator Priorities

| Priority | Operators | Ass. | Associativity |
|---|---|---|---|
| high | **! ~ ++ -- + -** | ⇐ | **right** to left |
| | * / % | ⇒ | left to right |
| | + - | ⇒ | left to right |
| | << >> | ⇒ | left to right |
| | < <= > >= | ⇒ | left to right |
| | == != | ⇒ | left to right |
| | & | ⇒ | left to right |
| | ^ | ⇒ | left to right |
| | \| | ⇒ | left to right |
| | && | ⇒ | left to right |
| | \|\| | ⇒ | left to right |
| | **? :** | ⇐ | **right** to left |
| | **= += -= *= /= %= &= ^= \|= <<= >>=** | ⇐ | **right** to left |
| low | , | ⇒ | left to right |

# 8.5 Symbolic Constants

## 8.5.1 Declaration of Symbolic Constants

*constant_declaration* ::= const *type* {*name_init*,}* *name_init*;

```
const float PI = 3.1415926; // magic Pi
```

The values of constants can not be changed by further assignments:

```
PI = 3.0  // Doesn't work!
```

## 8.5.2 Hexadecimal and Octal Constants

The C++ language has conventions for representing octal and hexadecimal values:

- **Octal** number: leading zeros
- **Hexadecimal** number: leading “0x” (zero + ‘x’):

**Examples**:

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

# 8.6 The char Data Type

The type char represents a single character, enclosed in single quotation marks (‘A’, ‘a’, ‘!’, ‘\b’).

```
char capitalA = ‘A’;
char smallB = ‘b’;
char lookOutChar = ‘!’;
char capitalB = 66; cout << capitalB;
```

The backslash character (\) is called the *escape character*, signalling that a special character follows.

```
char backspace = ‘\b’;
char newline = ‘\n’;
char backslash = ‘\\’;
char quote = ‘\’’
char doubleQuote = ‘\”’
```

## A small selection of special characters:

| Character | Name | Meaning |
|---|---|---|
| `\b` | Backspace | Move the cursor one character to the left |
| `\f` | Form feed | Go to the top of a new page |
| `\n` | New line | Go to the next line |
| `\r` | Return | Go to the beginning of the current line |
| `\t` | Tab | Advance to the next tab stop |
| `\` | Single quote | The character ‘ |
| `\` | Double quote | The character “ |
| `\nnn` | The character with ASCII code *nnn* | The ASCII character with number *nnn* (octal) |
| `\NN` | The character with ASCII code *NN* | The ASCII character with number *NN* (hexadecimal) |

# 8.7 References


- G. Blank and R. Barnes, *The Universal Machine*, Boston, MA: WCB/ McGraw-Hill, 1998. Chapter 3.2.