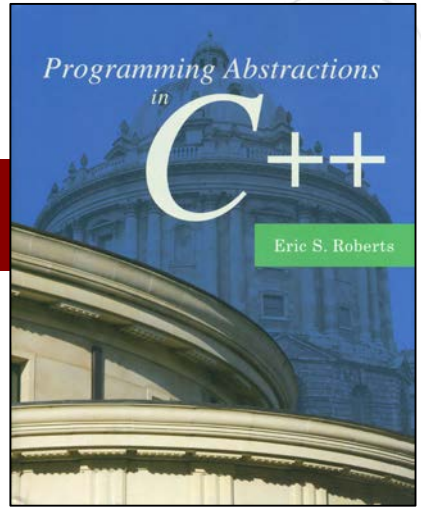
CHAPTER 1

An Overview of C++

Out of these various experiments come programs. This is our experience: programs do not come out of the minds of one person or two people such as ourselves, but out of day-to-day work.

—Stokely Carmichael and Charles V. Hamilton, *Black Power*, 1967



The "Hello World" Program

touching lives

The C++ programming language is an extension of the language C, which was developed at Bell Labs in the early 1970s. The primary reference manual for C was written by Brian Kernighan and Dennis Ritchie.

On the first page of their book, the authors suggest that the best way to begin learning a new programming language is to write a simple program that prints the message "hello, world" on the display. That advice remains sound today.

1.1 Getting Started

The only way to learn a new programming language is to write programs in it. The first program to write is the same for all languages:

Print the words

```
hello, world
```

This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print "**`hello, world`**" is

```
#include <stdio.h>

main() {
    printf("hello, world \n");
}
```

The "Hello World" Program in C++

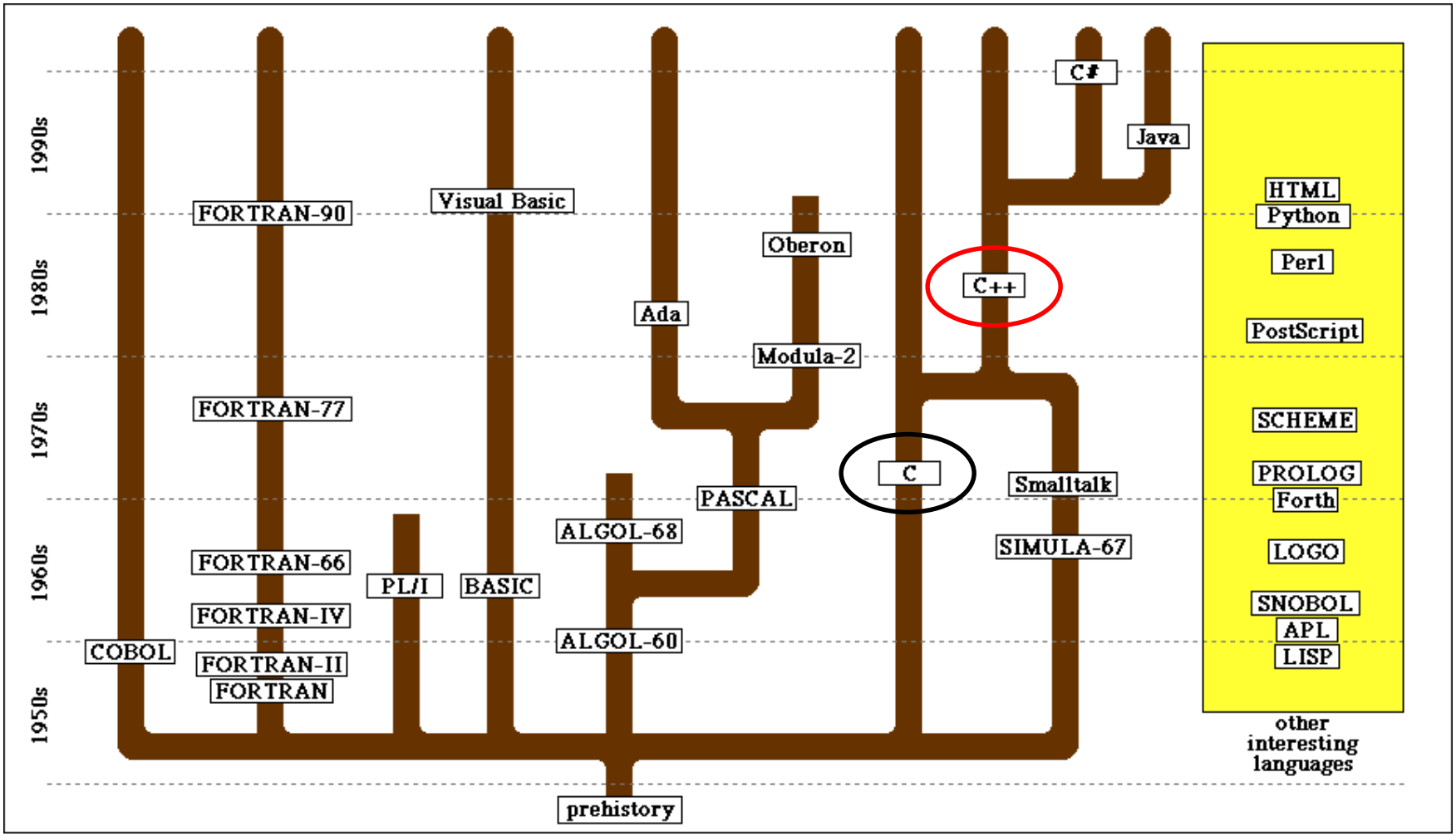

```
/*
 * File: HelloWorld.cpp
 * --------------------
 * This file is adapted from the example
 * on page 1 of Kernighan and Ritchie's
 * book The C Programming Language.
 */

#include <iostream>
using namespace std;

int main() {
   cout << "hello, world" << endl;
   return 0;
}
```

comments

library inclusions

main program

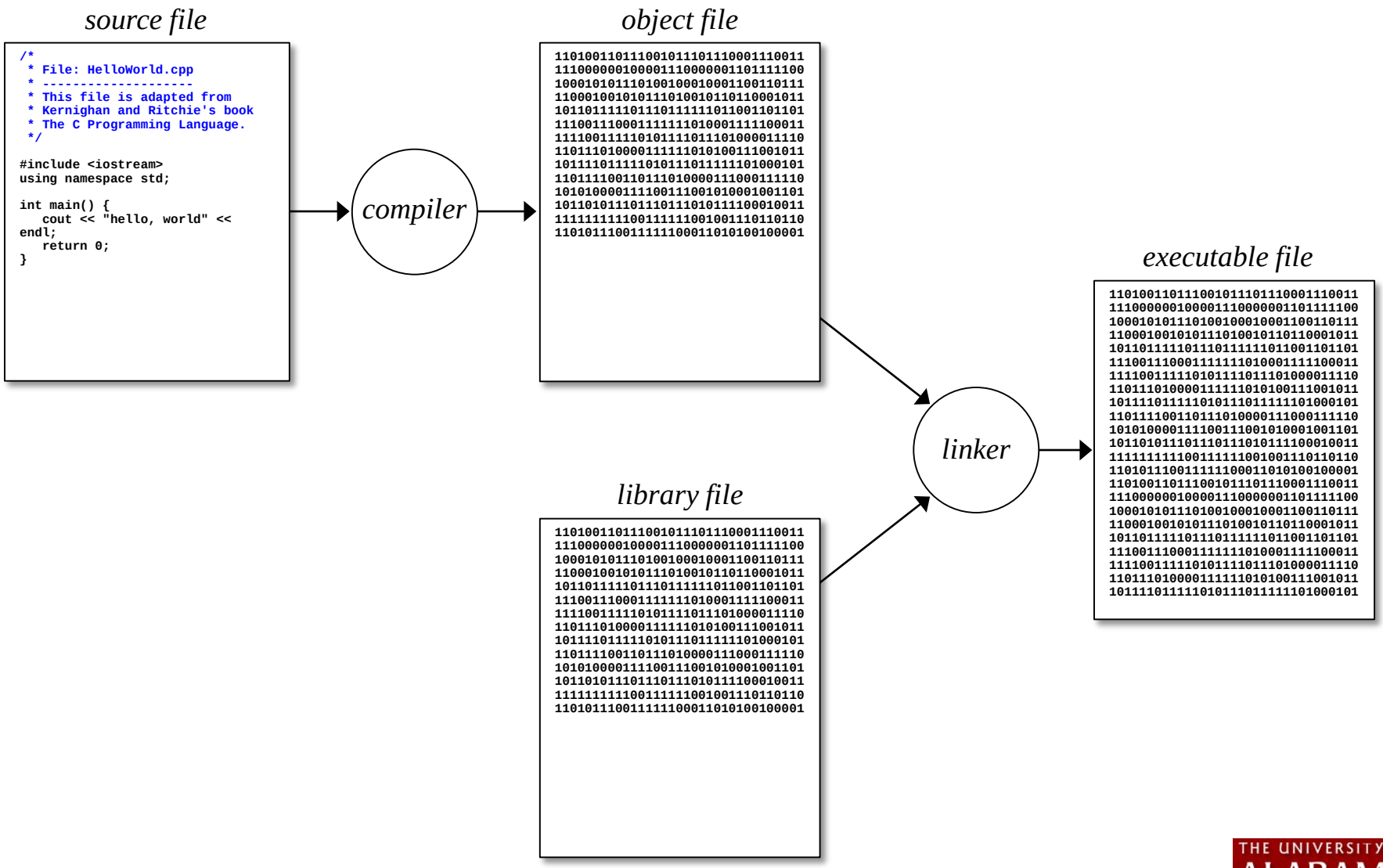
The History of C++



touching lives

THE UNIVERSITY OF ALABAMA
DEPARTMENT OF COMPUTER SCIENCE

The Compilation Process



The Structure of a C++ Program

```
/*
 * File: PowersOfTwo.cpp
 * ---------------------
 * This program generates a list of the powers of
 * two up to an exponent limit entered by the user.
 */

#include <iostream>
using namespace std;

/* Function prototypes */

int raiseToPower(int n, int k);

/* Main program */

int main() {
   int limit;
   cout << "This program lists powers of two." << endl;
   cout << "Enter exponent limit: ";
   cin >> limit;
   for (int i = 0; i <= limit; i++) {
      cout << "2 to the " << i << " = "
           << raiseToPower(2, i) << endl;
   }
   return 0;
}
```

Variables

- A ***variable*** is a named location for storing a value.
- In C++, you must ***declare*** a variable before you can use it. The declaration establishes the name and type of the variable and, in most cases, specifies the initial value as well.
- The most common form of a variable declaration is

```
type name = value;
```

  where *type* is the name of a C++ primitive type or class, *name* is an identifier that indicates the name of the variable, and *value* is an expression specifying the initial value.

- Most declarations appear as statements in the body of a method definition. Variables declared in this way are called ***local variables*** and are accessible only inside that method.
- Variables may also be declared as part of a class. These are called ***instance variables*** and are covered in Chapter 6.

Data Types

- Although many data types are represented using objects or other compound structures, C++ defines a set of ***primitive types*** to represent simple data.
- Of the eight primitive types available in C++, the programs in this text typically use only the following four:

`int`	This type is used to represent integers, which are whole numbers such as 17 or −53.
`double`	This type is used to represent numbers that include a decimal fraction, such as 3.14159265.
`bool`	This type represents a logical value (**`true`** or **`false`**).
`char`	This type represents a single ASCII character.

Expressions in C++

- The heart of the **`Add2Integers`** program from Chapter 2 is the line

```
int total = n1 + n2;
```

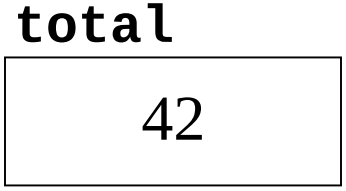
  that performs the actual addition.

- The **`n1 + n2`** that appears to the right of the equal sign is an example of an ***expression***, which specifies the operations involved in the computation.
- An expression in C++ consists of ***terms*** joined together by ***operators***.
- Each term must be one of the following:
  - A constant (such as **`3.14159265`** or **`"hello, world"`**)
  - A variable name (such as **`n1`**, **`n2`**, or **`total`**)
  - A method call that returns a value (such as **`readInt`**)
  - An expression enclosed in parentheses

Terms in an Expression

touching lives

- The simplest terms that appear in expressions are ***constants*** and ***variables***. The value of a constant does not change during the course of a program. A variable is a placeholder for a value that can be updated as the program runs.
- A variable in C++ is most easily envisioned as a box capable of storing a value.



(contains an **`int`**)

- Each variable has the following attributes:
  - A ***name***, which enables you to differentiate one variable from another.
  - A ***type***, which specifies what type of value the variable can contain.
  - A ***value***, which represents the current contents of the variable.
- The name and type of a variable are fixed. The value changes whenever you ***assign*** a new value to the variable.

Operators and Operands

- As in most languages, C++ programs specify computation in the form of ***arithmetic expressions*** that closely resemble expressions in mathematics.
- The most common operators in C++ are the ones that specify arithmetic computation:

`+`	Addition	`*`	Multiplication
`-`	Subtraction	`/`	Division
		`%`	Remainder

- Operators in C++ usually appear between two subexpressions, which are called its ***operands***. Operators that take two operands are called ***binary operators***.
- The `-` operator can also appear as a ***unary operator***, as in the expression `-x`, which denotes the negative of `x`.

Division and Type Casts

- In C++, whenever you apply a binary operator to numeric values, the result will be of type **`int`** if both operands are of type **`int`**, but will be a **`double`** if either operand is a **`double`**.
- This rule has important consequences in the case of division. For example, the expression

```
14 / 5
```

  seems as if it should have the value 2.8, but because both operands are of type **`int`**, C++ computes an integer result by throwing away the fractional part. The result is therefore 2.
- If you want to obtain the mathematically correct result, you need to convert at least one operand to a **`double`**, as in

```
double(14) / 5
```

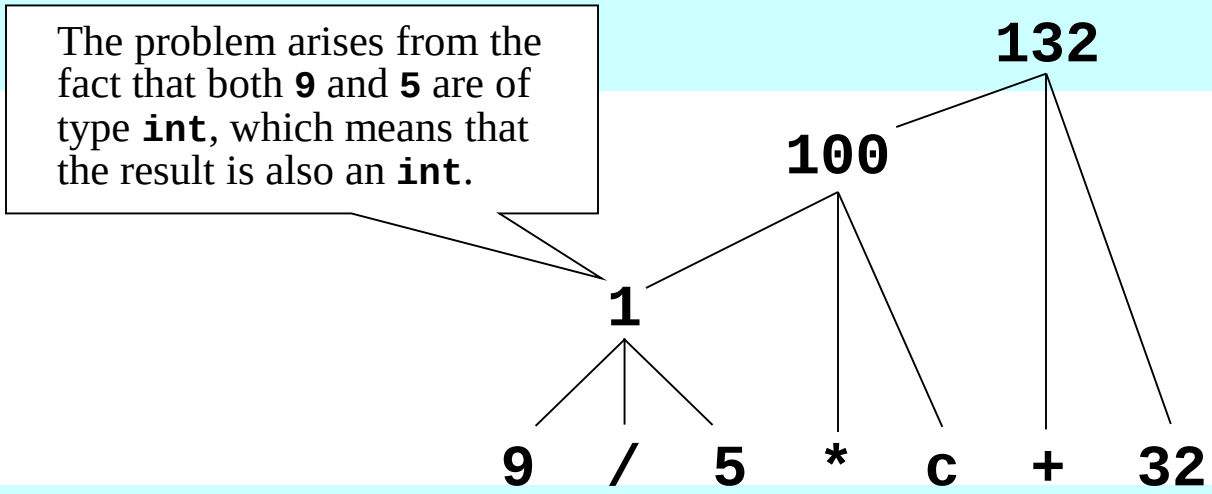
  The conversion is accomplished by means of a ***type cast***, which you specify by using the type name as a function call.

The Pitfalls of Integer Division

Consider the following C++ statements, which are intended to convert 100˚ Celsius temperature to its Fahrenheit equivalent:

```
double c = 100;
double f = 9 / 5 * c + 32;
```

The computation consists of evaluating the following expression:

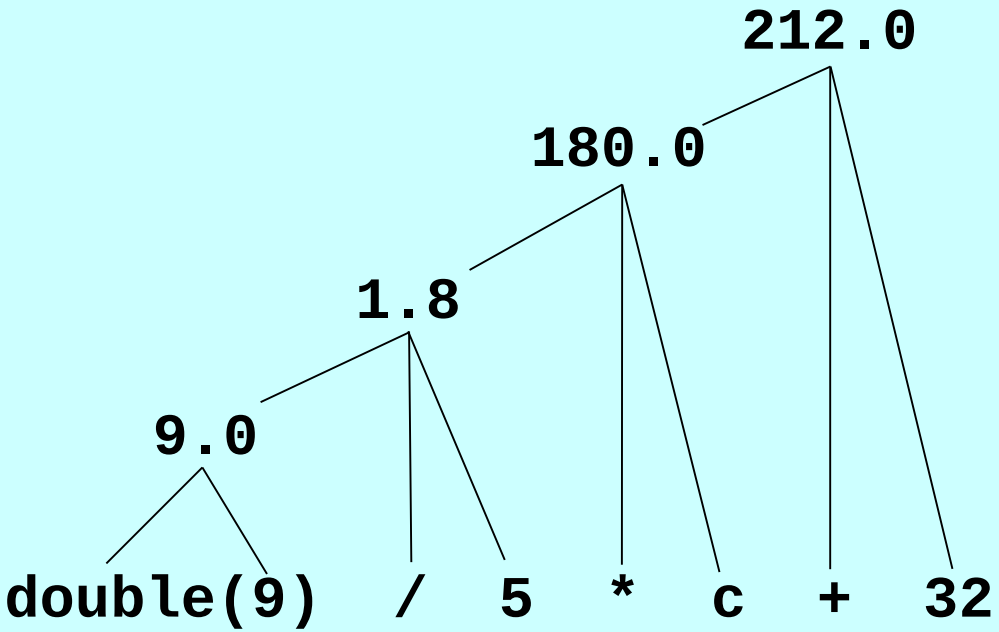


The Pitfalls of Integer Division

You can fix this problem by converting the fraction to a **double**, either by inserting decimal points or by using a type cast:

```
double c = 100;
double f = double(9) / 5 * c + 32;
```

The computation now looks like this:



The Remainder Operator

- The only arithmetic operator that has no direct mathematical counterpart is `%`, which applies only to integer operands and computes the remainder when the first divided by the second:

```
14 % 5   returns   4
14 % 7   returns   0
7 % 14   returns   7
```

- The result of the `%` operator make intuitive sense only if both operands are positive. The examples in the book do not depend on knowing how `%` works with negative numbers.
- The remainder operator turns out to be useful in a surprising number of programming applications and is well worth a bit of study.

Precedence

- If an expression contains more than one operator, C++ uses ***precedence rules*** to determine the order of evaluation. The arithmetic operators have the following relative precedence:

unary -
* / %
+ -

highest ↕ *lowest*

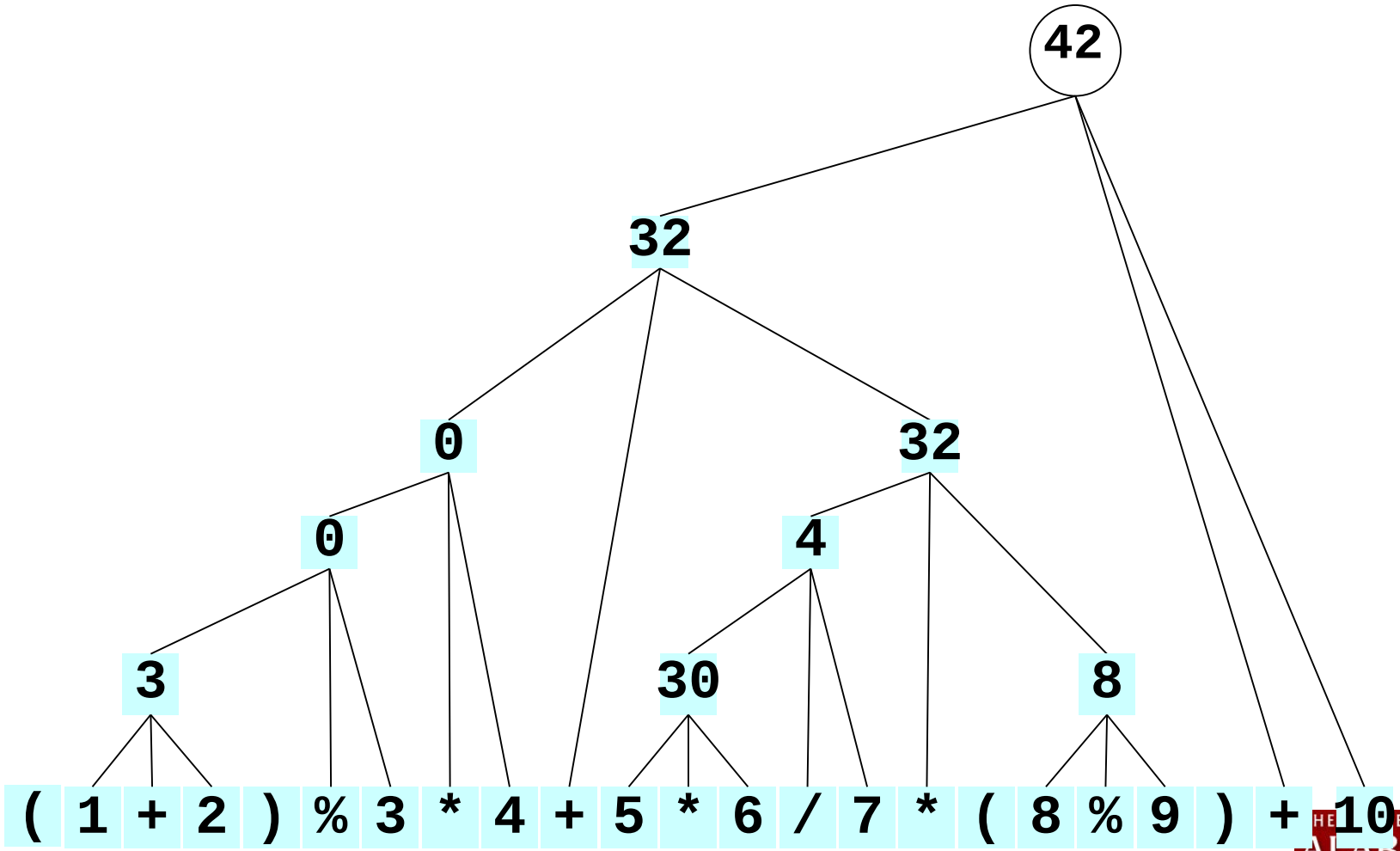
Thus, C++ evaluates any unary **-** operators first, followed by the operators *****, **/**, and **%**, and finally the operators **+** and **-**.

- Precedence applies only when two operands compete for the same operator. If the operators are independent, C++ evaluates expressions from left to right.
- Parentheses may be used to change the order of operations.

Precedence Evaluation

touching lives

What is the value of the expression at the bottom of the screen?



Assignment Statements

- You can change the value of a variable in your program by using an ***assignment statement***, which has the general form:

variable **=** *expression***;**

- The effect of an assignment statement is to compute the value of the expression on the right side of the equal sign and assign that value to the variable that appears on the left. Thus, the assignment statement

```
total = total + value;
```

adds together the current values of the variables **`total`** and **`value`** and then stores that sum back in the variable **`total`**.

- When you assign a new value to a variable, the old value of that variable is lost.

Shorthand Assignments

- Statements such as

```
total = total + value;
```

  are so common that C++ allows the following shorthand form:

```
total += value;
```

- The general form of a ***shorthand assignment*** is

variable *op*`=` *expression*`;`

  where *op* is any of C++'s binary operators. The effect of this statement is the same as

variable `=` *variable* *op* `(`*expression*`);`

  For example, the following statement multiplies **`salary`** by 2.

```
salary *= 2;
```

Increment and Decrement Operators

- Another important shorthand form that appears frequently in C++ programs is the ***increment operator***, which is most commonly written immediately after a variable, like this:

```
x++;
```

  The effect of this statement is to add one to the value of **x**, which means that this statement is equivalent to

```
x += 1;
```

  or in an even longer form

```
x = x + 1;
```

- The **--** operator (which is called the ***decrement operator***) is similar but subtracts one instead of adding one.
- The **++** and **--** operators are more complicated than shown here, but it makes sense to defer the details until Chapter 11.

Statement Types in C++

- Statements in C++ fall into three basic types:
  - Simple statements
  - Compound statements
  - Control statements
- ***Simple statements*** are formed by adding a semicolon to the end of a C++ expression.
- ***Compound statements*** (also called ***blocks***) are sequences of statements enclosed in curly braces.
- ***Control statements*** fall into two categories:
  - ***Conditional statements*** that specify some kind of test
  - ***Iterative statements*** that specify repetition

Boolean Expressions

- The operators used with the **`boolean`** data type fall into two categories: ***relational operators*** and ***logical operators***.
- There are six relational operators that compare values of other types and produce a **`boolean`** result:

`==`	Equals	**`!=`**	Not equals
`<`	Less than	**`<=`**	Less than or equal to
`>`	Greater than	**`>=`**	Greater than or equal to

For example, the expression **`n <= 10`** has the value **`true`** if **`n`** is less than or equal to 10 and the value **`false`** otherwise.

- There are also three logical operators:

`&&`	Logical AND	**`p && q`** means both **`p`** and **`q`**
`||`	Logical OR	**`p || q`** means either **`p`** or **`q`** (or both)
`!`	Logical NOT	**`!p`** means the opposite of **`p`**

Notes on the Boolean Operators

- Remember that C++ uses `=` to denote assignment. To test whether two values are equal, you must use the `==` operator.
- It is not legal in C++ to use more than one relational operator in a single comparison as is often done in mathematics. To express the idea embodied in the mathematical expression

$$0 \leq x \leq 9$$

  you need to make both comparisons explicit, as in

```
0 <= x && x <= 9
```

- The `||` operator means *either or both*, which is not always clear in the English interpretation of *or.*
- Be careful when you combine the `!` operator with `&&` and `||` because the interpretation often differs from informal English.

Short-Circuit Evaluation

- C++ evaluates the **`&&`** and **`||`** operators using a strategy called ***short-circuit mode*** in which it evaluates the right operand only if it needs to do so.
- For example, if **`n`** is 0, the right hand operand of **`&&`** in

  `n != 0 && x % n == 0`

  is not evaluated at all because **`n != 0`** is **`false`**. Because the expression

  `false &&` *anything*

  is always **`false`**, the rest of the expression no longer matters.
- One of the advantages of short-circuit evaluation is that you can use **`&&`** and **`||`** to prevent execution errors. If **`n`** were 0 in the earlier example, evaluating **`x % n`** would cause a "division by zero" error.

The `if` Statement

The simplest of the control statements is the **`if`** statement, which occurs in two forms. You use the first form whenever you need to perform an operation only if a particular condition is true:

```
if (condition) {
    statements to be executed if the condition is true
}
```

You use the second form whenever you want to choose between two alternative paths, one for cases in which a condition is true and a second for cases in which that condition is false:

```
if (condition) {
    statements to be executed if the condition is true
} else {
    statements to be executed if the condition is false
}
```

Common Forms of the `if` Statement

The examples in the book use only the following forms of the **`if`** statement:

Single line **`if`** statement

```
if (condition) statement
```

Multiline **`if`** statement with curly braces

```
if (condition) {
    statement
    . . . more statements . . .
}
```

`if`/**`else`** statement with curly braces

```
if (condition) {
    statements_true
} else {
    statements_false
}
```

Cascading **`if`** statement

```
if (condition_1) {
    statements_1
} else if (condition_2) {
    statements_2
. . . more else/if conditions . . .
} else {
    statements_else
}
```

The ?: Operator

- In addition to the **if** statement, C++ provides a more compact way to express conditional execution that can be extremely useful in certain situations. This feature is called the **?:** operator (pronounced *question-mark-colon*) and is part of the expression structure. The **?:** operator has the following form:

condition **?** $expression_1$ **:** $expression_2$

- When C++ evaluates the **?:** operator, it first determines the value of *condition*. If *condition* is **true**, C++ evaluates $expression_1$ and uses that as the value; if *condition* is **false**, C++ evaluates $expression_2$ instead.
- You can use the **?:** operator to assign the larger of **x** and **y** to the variable **max** like this:

```
max = (x > y) ? x : y;
```

The `switch` Statement

The **`switch`** statement provides a convenient syntax for choosing among a set of possible paths:

```
switch ( expression ) {
  case v1:
     statements to be executed if expression = v1
     break;
  case v2:
     statements to be executed if expression = v2
     break;
  . . . more case clauses if needed . . .
  default:
     statements to be executed if no values match
     break;
}
```

Example of the switch Statement

```
int main() {
   int month;
   cout << "Enter numeric month (Jan=1): ";
   cin >> month;
   switch (month) {
     case 2:
       cout << "28 days (29 in leap years)" << endl;
       break;
     case 4: case 6: case 9: case 11:
       cout << "30 days" << endl;
       break;
     case 1: case 3: case 5: case 7: case 8: case 12:
       cout << "31 days" << endl;
       break;
     default:
       cout << "Illegal month number" << endl;
       break;
   }
   return 0;
}
```

The End