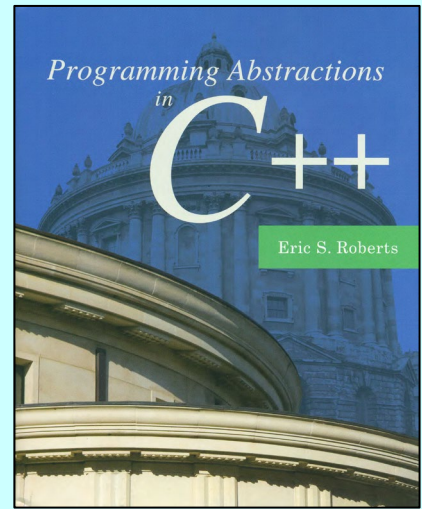


CHAPTER 1

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



1.1 The history of C++

1.2 The compilation process

1.3 Your first C++ program

1.4 The structure of a C++ program

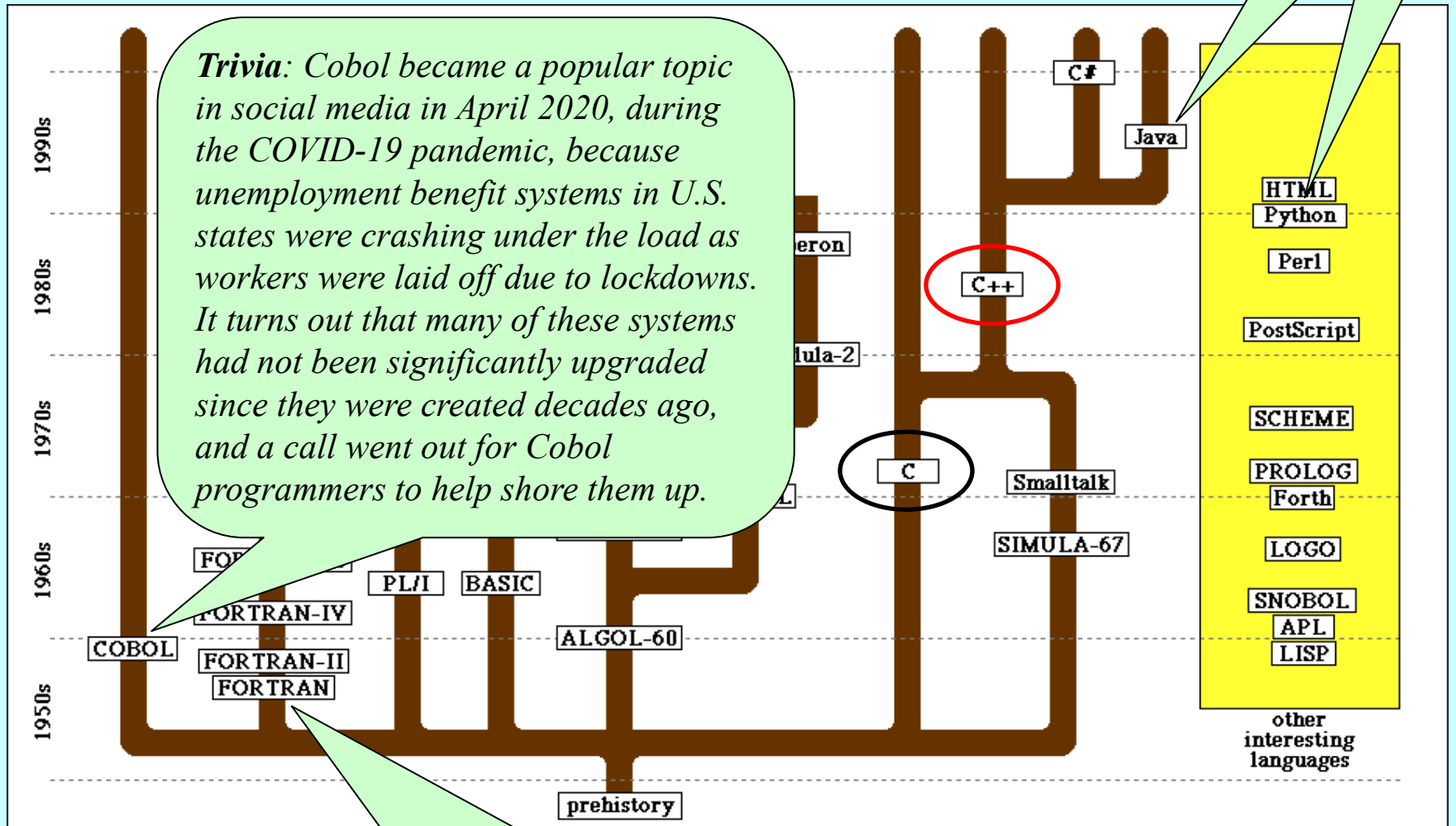
1.5 Variables

1.6 Data types

1.7 Expressions

1.8 Statements

The History of C++



The first widely used high-level general purpose programming language

C++ Versions

- The C++ programming language was developed by Bjarne Stroustrup at Bell Labs since 1979, as an extension of the C language (by Dennis Ritchie also at Bell Labs since 1972) as he wanted an efficient and flexible language similar to C, which also provided high-level features.
- C++ was initially standardized in 1998 by the International Organization for Standardization (ISO) as C++98, which was then amended by the C++03, C++11, C++14, C++17, and C++20 standards.
- The current C++23 standard supersedes these with new features and an enlarged standard library.
- C++26 is the next planned standard thereafter, keeping with the current streak of a new version every three years.
- We will use C++98 in most of the course material, and occasionally features from C++11.



C++ vs. Python

	C++	Python
Execution	Compiled and linked into an executable.	Interpreted and executed by an engine.
Types	Static typing , explicitly declared, bound to names, checked at compile time	Dynamic typing , bound to values, checked at run time, and are not so easily subverted
Memory Management	Requires much more attention to bookkeeping and storage details	Nearly no attention to be paid to memory management. Supports garbage collection .
Language Complexity	Complex and full-featured. C++ tries to give you every language feature under the sun while at the same time never (forcibly) abstracting anything away that could potentially affect performance.	Simple . Python tries to give you only one or a few ways to do things, and those ways are designed to be simple, even at the cost of some language power or running efficiency.



Compiler vs. Interpreter

- Compiler and Interpreter are two different ways to execute a program written in a programming or scripting language.
- A **compiler** takes entire program and converts it into object code which is typically stored in a file. The object code is also referred as binary code and can be directly executed by the machine after **linking**. Examples of compiled programming languages are C and **C++**.
- An **interpreter** directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. Examples of interpreted languages are **Python** and R.
- An analogy to human languages...

Compiler vs. Interpreter

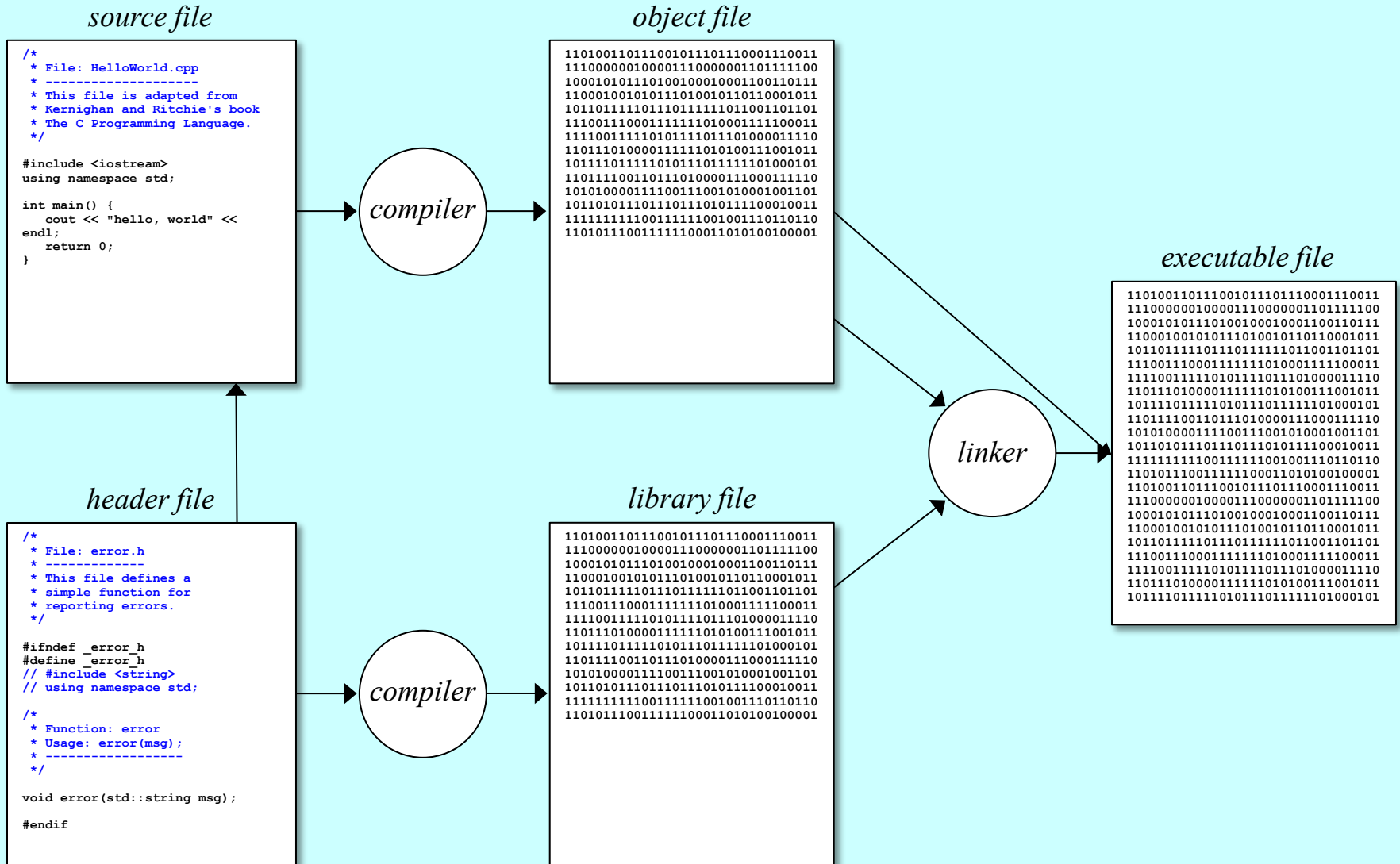
- Both compilers and interpreters convert source code (text files) into tokens. The basic difference is that a compiler system, including a (built-in or separate) linker, generates a **stand-alone machine code program**, while an interpreter system instead performs the actions described by the high-level program.
- Once a program is compiled, its **source code** is not useful for running the code. For interpreted programs, the source code is needed to run the program every time.
- In general, interpreted programs run **slower** than the compiled programs.

Compiler vs. Interpreter

- **Java** is a compiled programming language, but its source code is compiled to an intermediate binary form called *bytecode*.
- The bytecode is then executed by a Java Virtual Machine (JVM). Different JVMs support different systems. Write Once, Run Anywhere (WORA)!
- Modern JVMs use a technique called Just-In-Time (JIT) compilation to compile the bytecode to native instructions understood by the hardware CPU on the fly at runtime.
- Some implementations of JVMs may choose to interpret the bytecode, instead of JIT compiling it to machine code, and run it directly.
- An analogy to human languages...



The Compilation Process



IDE & Compiler

- **IDE** (Integrated Development Environment): A software that provides comprehensive facilities to computer programmers for software development, and normally consists of
 - **a source code editor**, which sometimes contains:
 - a class browser, an object browser, and a class hierarchy diagram, for use in **object-oriented** software development
 - **GUI** (Graphical User Interface) construction tools
 - a version control system
 - build automation tools (must at least contain a **compiler** or an **interpreter**, or both, sometimes a **linker** too)
 - a **debugger**
- All these components may be from a third party.

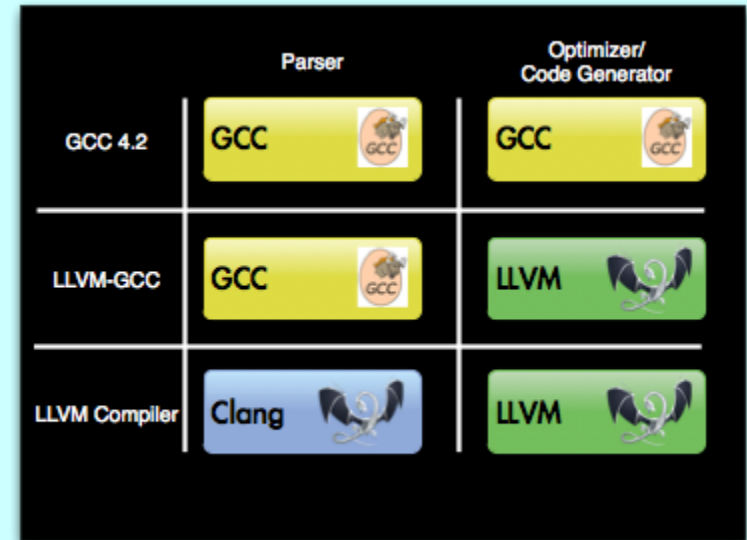
TUTORIAL

IDE & Compiler

g.,

hell (<http://cpp.sh/>)

- Code::Blocks
- Qt Creator
- Microsoft Visual Studio
- Apple Xcode
- Compilers, e.g.,
 - GCC (GNU Compiler Collection)
 - Clang + LLVM (Low-Level Virtual Machine)
 - MinGW (Minimalist GNU for Windows)
 - MSVC (Microsoft Visual C++)
- GUI and additional libraries support, e.g.,
 - Qt (Design Studio)
 - MFC (Microsoft Foundation Classes)
 - StanfordCPPLib (from the textbook)
 - OpenCV (Open source computer vision libraries)



The “Hello World” Program

The C++ programming language is an extension of the C language, 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.  
 */
```

comments

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

Don't worry about the font colors, you can customize these in most IDE editors.


library inclusions

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

main program

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

The Structure of a C++ Program

- A **comment** is text that is ignored by the compiler but which nonetheless conveys information about the source code to other programmers (actually, to yourself too).
- Multi-line comments

```
/* Comments line 1  
   Comments line 2  
   Comments line 3 */
```

- Single-line comments

```
// Comments line 1  
// Comments line 2  
// Comments line 3
```

- It is extremely important to write comments. Code tells you **how**, comments tell you **why**!

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;  
}
```

} *library inclusions*

The Structure of a C++ Program

- ***Libraries*** are collections of previously written tools that perform useful operations.
- **`#include <iostream>`** instructs the compiler to read the relevant definitions from what is called a **header (.h) file**; for you own libraries, write **`#include "myownlib.h"`**. (Remember **`import`** in Python?)
- To ensure that the names defined in different parts of a large system do not interfere with one another, C++ segments code into structures called **namespaces**, each of which keeps track of its own set of names.
- The **standard C++ libraries** use a namespace called **`std`**, which means that you cannot refer to the names defined in standard header files like *iostream.h* unless you let the compiler know to which namespace those definitions belong.

Preprocessor directives

- Preprocessor directives are not program statements but directives for the **preprocessor**, which examines the code before actual compilation.
- Each directive has the following format:
 - the number/hash/pound character (#)
 - preprocessing instruction (one of **include**, **define**, **undef**, **if**, **ifdef**, **ifndef**, **else**, **elif**, **endif**, **line**, **error**, **pragma**)
 - arguments (depending on the instruction)
 - the new line character (line break)
- No semicolon (;) is expected at the end of a preprocessor directive, and one can extend a preprocessor directive through more than one line by preceding the newline character at the end of the line by a backslash (\).

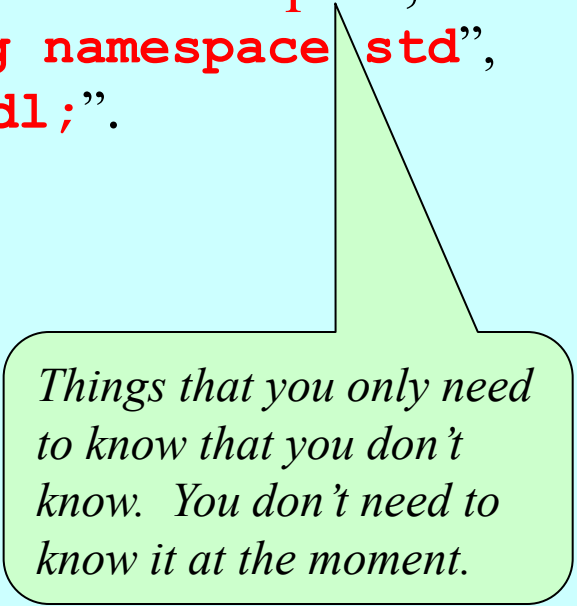
Things that you only need to know that you don't know. I don't need you to memorize it for the exam.

Using “using namespace”

- When programs get very large and make heavy use of different libraries, **name collisions** may happen, i.e., an identifier (e.g., a function name) is used by different libraries to refer to different things. The compiler won't let the same name be used for two different things!
- The **namespace** idea is that identifiers can be grouped into separate sets, and each such set, or namespace, has a name. If the same identifier appears in two libraries, each with their own namespace, the collision can be resolved by qualifying the identifier with the namespace name.
- To avoid the inconvenience of writing the namespace name all the time, we may use “**using namespace x**” to let the compiler know that an unqualified identifier might come from namespace **x**.

Using “using namespace”


- To avoid always using the entire namespace, one can also use specific using declarations for just the names you need, e.g.,
“**using x::name1; using x::name2;**”
- You can declare and put things into **your own namespace**, but in this course we mostly just need “**using namespace std**”, or “**using std::cout; using std::endl;**”.



Things that you only need to know that you don't know. You don't need to know it at the moment.

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;  
}
```



main program

The Structure of a C++ Program

- Every C++ program must contain a function with the name **main**. It specifies the starting point for the computation and is called when the program starts up. When **main** has finished its work and returns, execution of the program ends.
- A **function** is a named section of code that performs a specific operation.
- The **main** function can **call** other functions, e.g., **raiseToPower** in the **PowersOfTwo** program on the next slide.
- **Imperative programming paradigm**: an explicit sequence of statements that change a program's state, specifying **how** to achieve the result.
 - Structured: Programs have clean, **goto**-free, nested **control structures**.
 - Procedural: Imperative programming with **procedures** operating on data.

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;
}
```

program comment

library inclusions

function prototype

main program

The Structure of a C++ Program

```
/*  
 * Function: raiseToPower  
 * Usage: int p = raiseToPower(n, k);  
 * -----  
 * Returns the integer n raised to the kth power.  
 */  
  
int raiseToPower(int n, int k) {  
    int result = 1;  
    for (int i = 0; i < k; i++) {  
        result *= n;  
    }  
    return result;  
}
```

function comment

function definition

Question

- Is **PowerOfTwo** a good program?



The “declare-before-use” rule

- A C++ program consists of various entities such as variables, functions, types, and namespaces. Each of these entities **must be declared before they can be used**. (Think about the inclusion of libraries in the very beginning of the program).
- Notice that function **raiseToPower** is defined in the function definition part after the **main** function. To comply with the “declare-before-use” rule, a declaration of the function, called a **function prototype**, must appear before the **main** function.
- The function prototype specifies a unique name for the function, along with information about its type and other characteristics (more details later).
- Can we avoid using the function prototypes?
 - How about we always define a function before we use it?



C++ vs. Python

	C++	Python
Execution	Compiled and linked into an executable.	Interpreted and executed by an engine.
Types	Static typing , explicitly declared, bound to names, checked at compile time	Dynamic typing , bound to values, checked at run time, and are not so easily subverted
Memory Management	Requires much more attention to bookkeeping and storage details	Nearly no attention to be paid to memory management. Supports garbage collection .
Language Complexity	Complex and full-featured. C++ tries to give you every language feature under the sun while at the same time never (forcibly) abstracting anything away that could potentially affect performance.	Simple . Python tries to give you only one or a few ways to do things, and those ways are designed to be simple, even at the cost of some language power or running efficiency.

Data Types

- A data type is defined by a domain, which is the set of values that belong to that type, and a set of operations, which defines the behavior of that type.
- Although many data types are represented using **object classes** or other **compound structures** (more details later), C++ defines a set of ***primitive types*** to represent simple data.
- Of all the 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.

Variables

- A *variable* is a named *address* for storing a type of 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 more complicated type such as class, *name* is an identifier that indicates the name of the variable, and *value* is an expression specifying the initial value.

- Through the declaration, a memory *address* is associated with the variable name, which we will discuss later in memory management.



Variables

- A variable in C++ is most easily envisioned as a box capable of storing a value. For the following statement:

```
int total = 42;
```

(name is) **total**

(stores at) FFD0

42

(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.
 - For now, let's not worry about the **address** first.
- The **address** and **type** of a **named** variable are **fixed**. The value changes whenever you **assign** a new value to the variable.

Question

- What are the differences between Python variables and C++ variables?

Variables

- Variable name is **case sensitive**.
- Naming conventions (also a part of programming style)
 - The name must start with a letter or the underscore character (_).
 - All other characters in the name must be letters, digits, or the underscore.
No spaces or other special characters are permitted in names.
 - Use **meaningful words** (good variable names are like comments), but must not be one of the **reserved keywords**.

TABLE 1-1 Reserved words in C++

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

Variable *Scope* and *Extent*

- Most declarations appear as statements in the body of a function definition. Variables declared in this way are called **local variables** and are accessible only inside that function. They are also sometimes called **automatic variables** because they are automatically created when the function starts execution, and automatically go away when the function is finished executing.
- Variables may also be declared as part of a class. These are called **instance variables** and are covered in later chapters.
- Variables declared outside any function and class definition are **global variables**. **Avoid** global variables because they can be manipulated by any function in a program, and it is difficult to keep those functions from interfering with one another.



Richard
@zzaaho

Q: What is the best prefix for global variables?

A: //

4:41 AM - 21 Jan 2019

Variable *Scope* and *Extent*

- In computer programming, a ***name binding*** is an association of a name to an entity (e.g., data), such as a variable.
- The ***scope*** of a name binding is the ***region*** of a computer program where the binding is valid, i.e., where the name can be used to refer to the entity. From the perspective of the referenced entity, it is also known as the ***visibility*** of the entity.
- While the ***scope*** of a variable describes ***where*** in a program's text the variable may be used, the ***extent*** (or ***lifetime***) describes ***when*** in a program's execution a variable has a (meaningful) value.

Question

- Are variables defined in `main()` function global variables?

Constants

- Unlike a variable, which is a placeholder for a value that can be updated as the program runs, the value of a constant does not change during the course of a program.
- The format of a constant depends on its type:
 - Integral constants consist of a string of digits, optionally preceded by a minus sign, as in **0**, **42**, **-1**, or **1000000**.
 - Floating-point constants include a decimal point, as in **3.14159265** or **10.0**. Floating-point constants can also be expressed in scientific notation by adding the letter **E** and an exponent after the digits of the number, so that **5.646E-8** represents the number 5.646×10^{-8} .
 - The two constants of type **boolean** are **true** and **false**.
 - Character and string constants are discussed in detail in later lectures. For the moment, all you need to know is that a character constant is enclosed in single quotation marks, such as **'a'**, while a string constant consists of a sequence of characters enclosed in double quotation marks, such as **"hello, world"**.

Named constants

- If you use the same constant many times in a program, it would be better if you could give this constant a name and then refer to it by that name everywhere in the program.
- To create a named constant in C++, precede the regular variable declaration with the **keyword `const`**. This tells the compiler that a special variable has been created which has a value that is **invariable**.

```
const double PI = 3.14159265358979323846;
```

- When creating a named constant, it must be assigned a value. You cannot assign a value to a named constant after its initialization.

Named constants

- In C, the **preprocessor directive #define** was used to create a name for a constant so that the preprocessor can replace every instance of the name with the appropriate constant throughout the code.

```
#define PI 3.14159265358979323846
```

- This still works in C++, but since **a constant created this way exists only in the file where it is created**, it is possible to have the same definition in another file with a completely different value. This could lead to disastrous consequences.
- Using **const** is a good habit because we can control its scope.

Expressions

```
/*  
 * File: AddThreeNumbers.cpp  
 * -----  
 * This program adds three floating-point numbers and prints their sum.  
 */  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    double n1, n2, n3;  
    cout << "This program adds three numbers." << endl;  
    cout << "1st number: ";  
    cin >> n1;  
    cout << "2nd number: ";  
    cin >> n2;  
    cout << "3rd number: ";  
    cin >> n3;  
    double sum = n1 + n2 + n3;  
    cout << "The sum is " << sum << endl;  
    return 0;  
}
```

Expressions

- The heart of the **AddThreeNumbers** program is the line

```
double sum = n1 + n2 + n3;
```

that performs the actual addition.

- The **n1 + n2 + n3** 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 *value* (such as **3.14159265** or **"hello, world"**)
 - A constant/variable *name* (such as **PI**, **n1**, **n2**, or **total**)
 - A function call that returns a value (such as **readInt()**)
 - An expression enclosed in parentheses (recursive definition)

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 sub-expressions, 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)/double(5)`

`double(14)/5`

`14/double(5)`

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

Division and Type Casts

- Question: what is the result of the following expression?

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

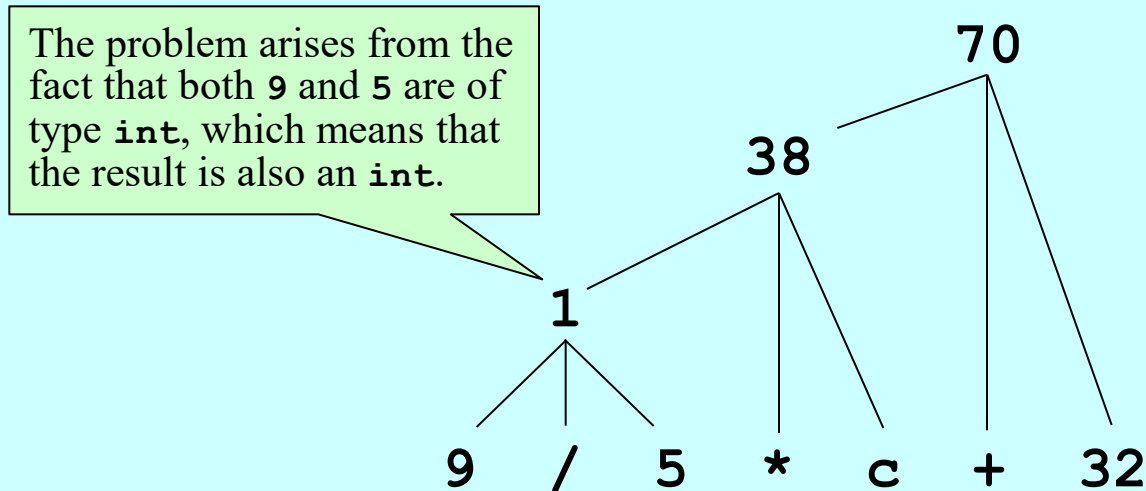
Exercise: The Pitfalls of Integer Division

Consider the following C++ statements, which are intended to convert 38° Celsius temperature to its Fahrenheit equivalent:

```
double c = 38;  
double f = 9 / 5 * c + 32;
```



The computation consists of evaluating the following expression:

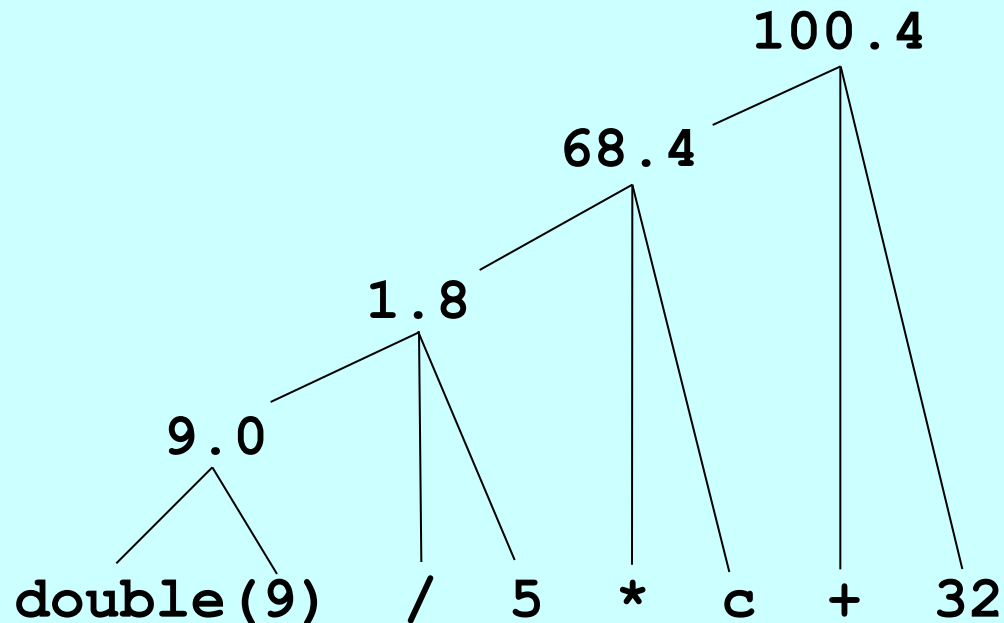


The Pitfalls of Integer Division

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 = 38;  
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 is divided by the second:

14	%	5	<i>returns</i>	4
14	%	7	<i>returns</i>	0
7	%	14	<i>returns</i>	7

- The result of the % operator makes intuitive sense only if both operands are positive. The examples in the textbook 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:

TABLE 1-4 Operators available in C++

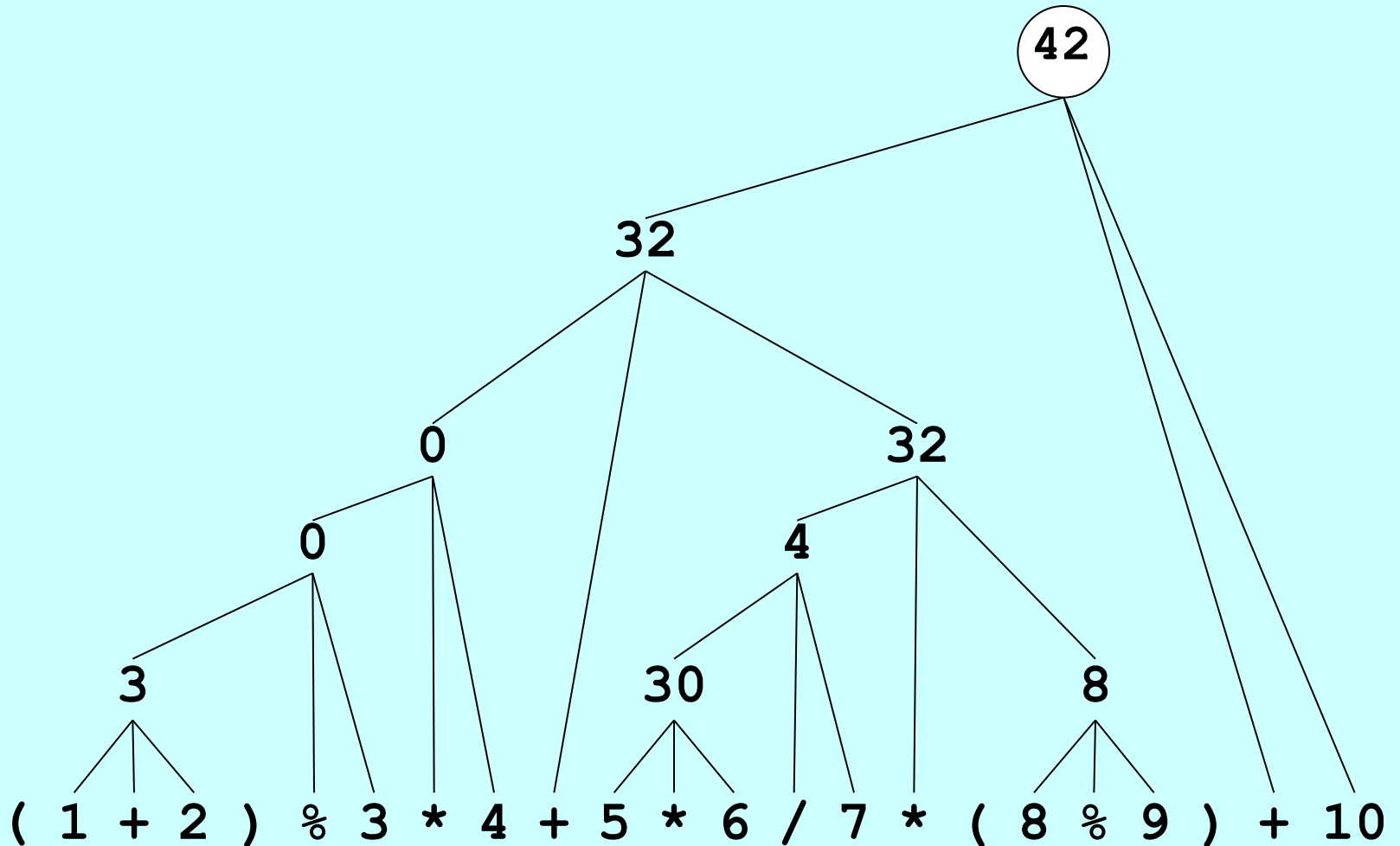
Operators organized into precedence groups	Associativity
() [] -> .	<i>left</i>
<i>unary operators:</i> - ++ -- ! & * ~ (type) sizeof	<i>right</i>
* / %	<i>left</i>
+ -	<i>left</i>
<< >>	<i>left</i>
< <= > >=	<i>left</i>
== !=	<i>left</i>
&	<i>left</i>
^	<i>left</i>
	<i>left</i>
&&	<i>left</i>
	<i>left</i>
? :	<i>right</i>
= op=	<i>right</i>

Precedence

- Precedence applies only when two operators compete for the same operand. If two operators have the same precedence, they are applied in the order specified by their *associativity*, which indicates whether that operator groups to the left or to the right.
- Most operators in C++ are left-associative, which means that the leftmost operator is evaluated first.
- A few operators – most notably the assignment operator – are right-associative, which means that they group from right to left.
- **Parentheses** may be used to change the order of operations. It is always a good habit to use parentheses to clearly identify the precedence and associativity (like writing comments).

Exercise: Precedence Evaluation

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.

Question



As a mathematical equation, this is wrong of course. But as an assignment statement in C++, is it legal, and why?

L-value: has an address, can appear on both sides of assignments.

R-value: no address, cannot appear on the left side of assignments.

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++;
```

```
++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 later.

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 recommended** in C++ to use more than one relational operator in a single comparison as in mathematics. To express the idea embodied in the mathematical expression

`0 <= x <= 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* (e.g., *exclusive 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.

Statements

- 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
- One can break a long line of code using a backslash (\). Most of the time compilers can recognize a broken line, but sometimes (e.g., in preprocessor directives) \ is needed.

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  
}
```

*conditions are
expressions that
evaluate to a
boolean value*

- 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) {  
    statementstrue  
} else {  
    statementsfalse  
}
```

Cascading **if** statement

```
if (condition1) {  
    statements1  
} else if (condition2) {  
    statements2  
... more else/if conditions ...  
} else {  
    statementselse  
}
```

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 ? expression1 : expression2
```

- 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  $v_1$ :  
        statements to be executed if expression =  $v_1$   
        break;  
    case  $v_2$ :  
        statements to be executed if expression =  $v_2$   
        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, Feb=2, etc.): ";  
    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 10: case 12:  
            cout << "31 days" << endl;  
            break;  
        default:  
            cout << "Illegal month number" << endl;  
            break; // not necessary but a good habit  
    }  
    return 0;  
}
```

*If **break** is not included in one case, all statements following the case are also executed, until a **break** is reached or the end of the switch block.*

The **while** Statement

- The **while** statement is the simplest of C++'s iterative control statements and has the following form:

```
while ( condition ) {  
    statements to be repeated  
}
```

- When C++ encounters a **while** statement, it begins by evaluating the condition in parentheses.
- If the value of *condition* is **true**, C++ executes the statements in the body of the loop.
- At the end of each cycle, C++ reevaluates *condition* to see whether its value has changed. If *condition* evaluates to **false**, C++ exits from the loop and continues with the statement following the closing brace at the end of the **while** body.

The **for** Statement

The **for** statement in C++ is a particularly powerful tool for specifying the control structure of a loop independently from the operations the loop body performs. The syntax looks like this:

```
for ( init ; test ; step ) {  
    statements to be repeated  
}
```

C++ evaluates a **for** statement by executing the following steps:

1. Evaluate *init*, which typically declares a ***control variable***.
2. Evaluate *test* and exit from the loop if the value is **false**.
3. Execute the statements in the body of the loop.
4. Evaluate *step*, which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

Comparing **for** and **while**

- The **for** statement

```
for ( init ; test ; step ) {  
    statements to be repeated  
}
```

is functionally equivalent to the following code using **while**:

```
init ;  
while ( test ) {  
    statements to be repeated  
    step ;  
}
```

- The advantage of the **for** statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

Exercise: Reading **for** Statements

Describe the effect of each of the following **for** statements:

1. `for (int i = 1; i <= 10; i++)`

*This statement executes the loop body **10** times, with the control variable **i** taking on each successive value between 1 and 10.*

2. `for (int i = 0; i < N; i++)`

*This statement executes the loop body **N** times, with **i** counting from 0 to $N - 1$. This version is *the standard Repeat-N-Times idiom*.*

3. `for (int n = 99; n >= 1; n -= 2)`

*This statement counts backward from 99 to 1 by 2's, for **50** times.*

4. `for (int x = 1; x <= 1024; x *= 2)`

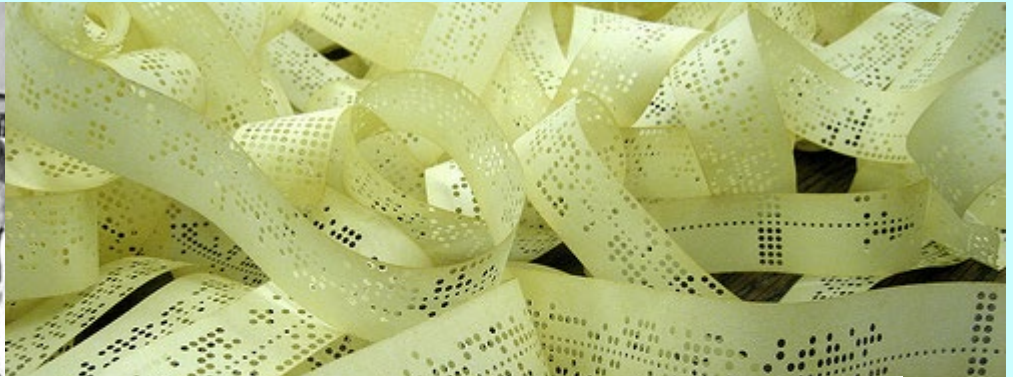
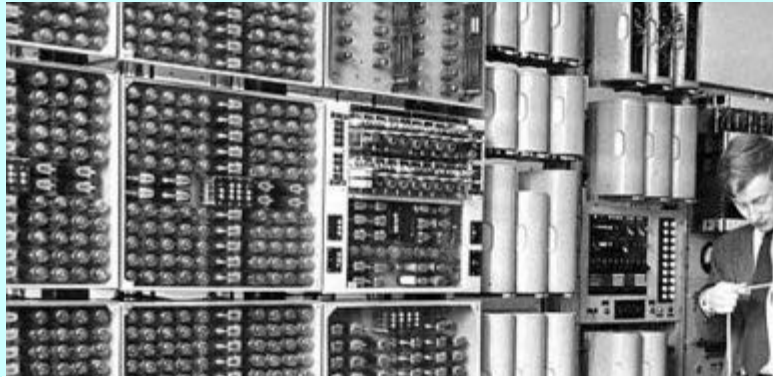
*This statement executes the loop body with the variable **x** taking on successive powers of 2 from 1 up to 1024, for **11** times.*

C++ Programming Style Guide

- *Style*, also known as readability, is what we call the conventions that govern our C++ code. The term Style is a bit of a misnomer, since these conventions cover far more than just source file formatting.
- Use common sense and *be consistent*.
- <https://google.github.io/styleguide/cppguide.html>
- Style matters!



Debug: Finding the Bugs



Joe Groff

@jckarter

After printf, sleeping is the
second best debugger

2:16 AM - 15 Feb 2019

@爱可可-爱生活 u.com

Unit Testing

- One of the most important responsibilities you have as a programmer is to **test your code as thoroughly as you can**. Although testing can never guarantee the absence of errors, adopting a deliberate testing methodology helps you to find at least some of the errors in your code.
- Whenever you write a module, it is good practice to create a test that checks the correctness of that module in isolation from the rest of the code. Such tests are called ***unit tests***.
- You can use the **assert macro** from the `<cassert>` library to implement the unit-testing strategy. If the conditional expression in the **assert** macro is **true**, execution continues normally. If the expression is **false**, the **assert** macro prints a message identifying the source of the error and exits from the program.
- A **macro** is a fragment of code that has been given a name. Whenever the name is used, it is replaced by the contents of the macro.

Example of the `assert` macro

```
int main()
{
    int x = 7;

    /* Some big code in between and let's say x
       is accidentally changed to 9 */
    x = 9;

    // Programmer assumes x to be 7 in rest of the code
    assert(x==7);

    /* Rest of the code */

    return 0;
}
```

Things that you only need to know that you don't know. I don't need you to fully understand it now.

Assertion failed: `x==7`, file `test.cpp`, line 13

This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

The End