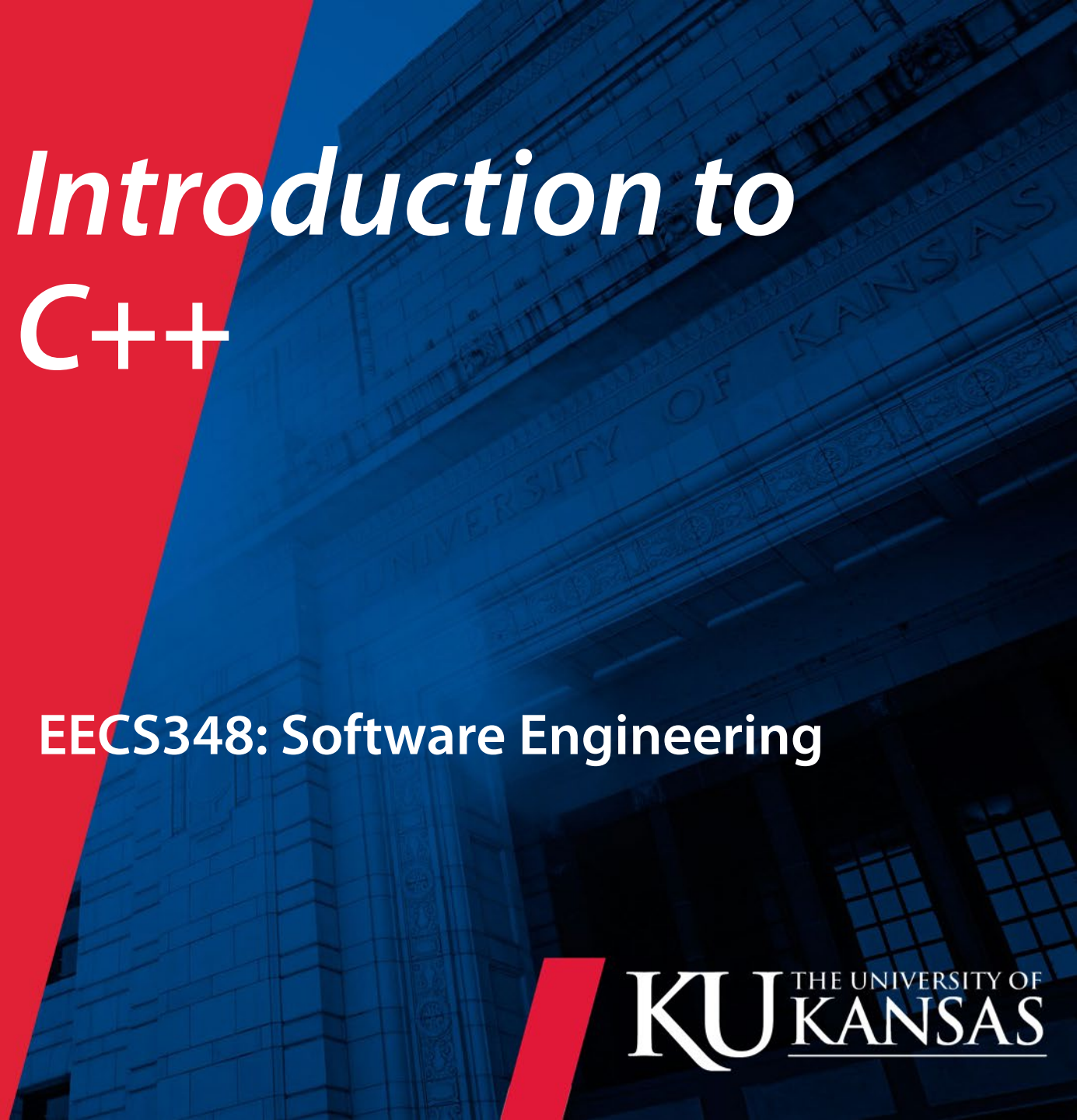
Introduction to C++

EECS348: Software Engineering



THE UNIVERSITY OF KANSAS

C++

- C++: a powerful computer programming language that's appropriate for technically oriented people with little or no programming experience, and for experienced programmers to use in building substantial information systems
- C++ is one of today's most popular software development languages
- C++ evolved from C, which was developed by Dennis Ritchie at Bell Laboratories
  - C++, an extension of C, was developed by Bjarne Stroustrup in 1979 at Bell Laboratories
  - C++ provides several features that "spruce up" the C language

C++ Standard Library

- C++ Standard Library
  - C++ programs consist of pieces called classes and functions
  - Most C++ programmers take advantage of the rich collections of classes and functions in the C++ Standard Library
  - Two parts to learning the C++ world
    * The C++ language itself, and
    * How to use the classes and functions in the C++ Standard Library
  - Many special-purpose class libraries are supplied by independent software vendors

SE observations

Software Engineering Observation 1.1

Use a "building-block" approach to create programs. Avoid reinventing the wheel. Use existing pieces wherever possible. Called software reuse, this practice is central to object-oriented programming.

Software Engineering Observation 1.2

When programming in C++, you typically will use the following building blocks: classes and functions from the C++ Standard Library, classes and functions you and your colleagues create and classes and functions from various popular third-party libraries.

Performance Tip 1.2

Using C++ Standard Library functions and classes instead of writing your own versions can improve program performance, because they're written carefully to perform efficiently. This technique also shortens program development time.

Compiled programs

- Compiling a high-level language program into machine language can take a considerable amount of computer time
- Interpreter programs were developed to execute high-level language programs directly (without the need for compilation), although more slowly than compiled programs
- Scripting languages such as the popular web languages JavaScript and PHP are processed by interpreters

Data hierarchy

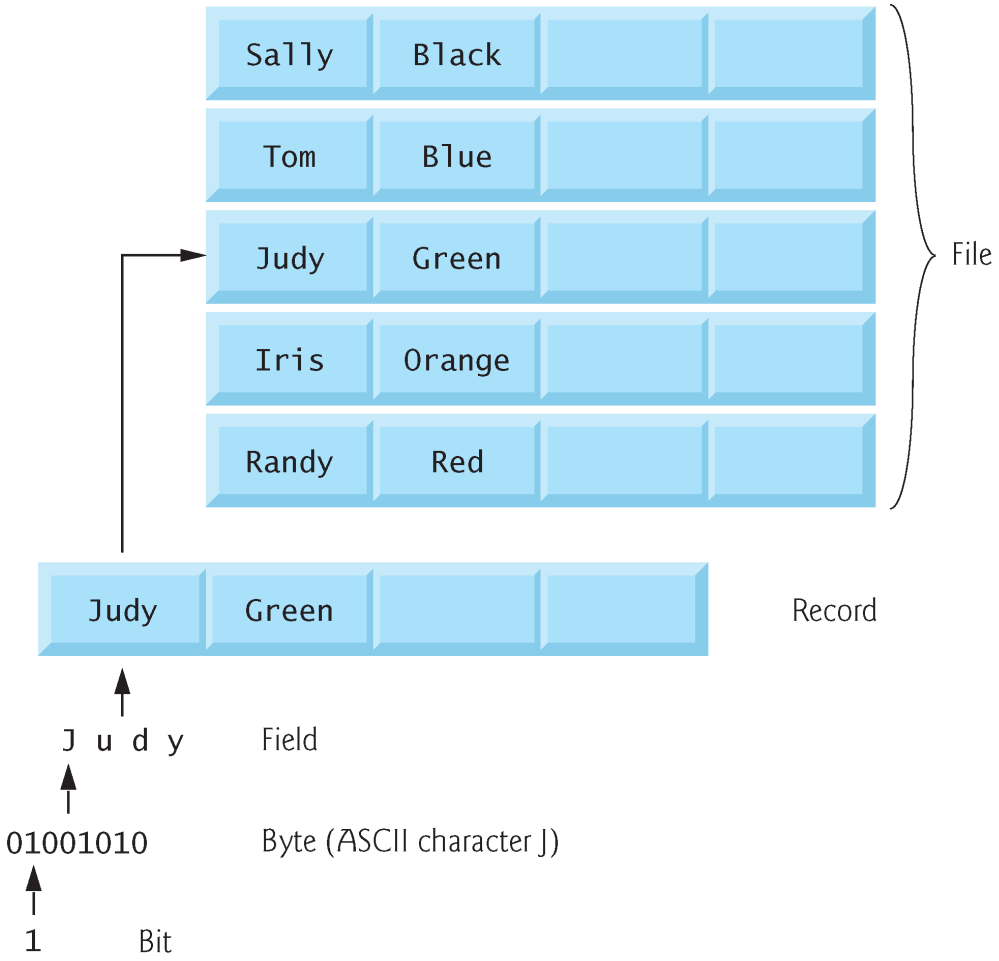


Fig. 1.3 | Data hierarchy.

Key SE concepts supported by OO languages

- Classes
  - Attributes and methods
  - Member functions
  - Maybe public, private, or protected
- Encapsulation
  - Information hiding
- Inheritance
  - Single, double
  - Facilitate reusability
- Polymorphism

Welcome to C++ program

```
1  // Fig. 2.1: fig02_01.cpp
2  // Text-printing program.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome to C++!\n"; // display message
9
10    return 0; // indicate that program ended successfully
11 } // end function main
```

```
Welcome to C++!
```

Fig. 2.1 | Text-printing program.

Welcome to C++ program

- // indicates a comment; extends to the end of the line
- Can still use /* …. */ combination
- A # directive is a message to the preprocessor
- `#include <iostream>` is for I/O
- Like C, white spaces are ignored
- `main()` function is part of every C++ program
  - C++ programs begin executing at function `main()`
  - The rest of the program consists of classes and functions

Welcome to C++ program

- Typically, output and input in C++ are accomplished with streams of characters
- When a **`cout`** statement executes, it sends a stream of characters to the standard output stream object **`std::cout`** which is normally connected to the screen
  - The notation **`std::cout`** specifies that we are using a name, in this case **`cout`**, that belongs to "namespace" **std**
  - The names **`cin`** (the standard input stream) and **`cerr`** (the standard error stream) also belong to namespace std

Printing a line of text

```
std::cout << "Welcome to C++\n";
```

- The **<<** operator is referred to as the stream insertion operator
- The value to the operator's right is inserted in the output stream
- The escape sequence `\n` means newline
  - Causes the cursor to move to the beginning of the next line on the screen
- Using multiple stream insertion operators **(<<)** in a single statement is referred to as concatenating, chaining or cascading stream insertion operations

Printing a line of text

- The welcome message can be printed in other ways:

```
std::cout << “Welcome ”;
std::cout << “to C++\n”;
```

Other escape sequences

Escape sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
\\	Backslash. Used to print a backslash character.
\'	Single quote. Used to print a single quote character.
\"	Double quote. Used to print a double quote character.

Fig. 2.2 | Escape sequences.

The return statement

- When the `return` statement is used at the end of main the value 0 indicates that the program has terminated successfully
- According to the C++ standard, if program execution reaches the end of main without encountering a return statement, it's assumed that the program terminated successfully

Variable declarations, definitions like C

- Fundamental types are like in C (all in lowercase)
  - `Short, int, long, float, double, char`
- Identifiers (variables, function names, …) also like C
  - Letters, digits, _
  - Case sensitive
- Assignment operators also like in C: `+, -, *, /, %`

C++ operation	C++ arithmetic operator	Algebraic expression	C++ expression
Addition	+	$f + 7$	f + 7
Subtraction	-	$p - c$	p - c
Multiplication	*	bm or $b \cdot m$	b * m
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	x / y
Modulus	%	$r \bmod s$	r % s

Fig. 2.9 | Arithmetic operators.

Operator precedence

Operator(s)	Operation(s)	Order of evaluation (precedence)
`( )`	Parentheses	Evaluated first. If the parentheses are *nested*, such as in the expression `a * ( b + c / d + e )`, the expression in the *innermost* pair is evaluated first. [*Caution:* If you have an expression such as `(a + b) * (c - d)` in which two sets of parentheses are not nested, but appear "on the same level," the C++ Standard does *not* specify the order in which these parenthesized subexpressions will be evaluated.]

`*`
`/`
`%`
	Multiplication
	Division
	Modulus
		Evaluated second. If there are several, they're evaluated left to right.

`+` `-`	Addition Subtraction	Evaluated last. If there are several, they're evaluated left to right.

Fig. 2.10 | Precedence of arithmetic operators.

Operator precedence

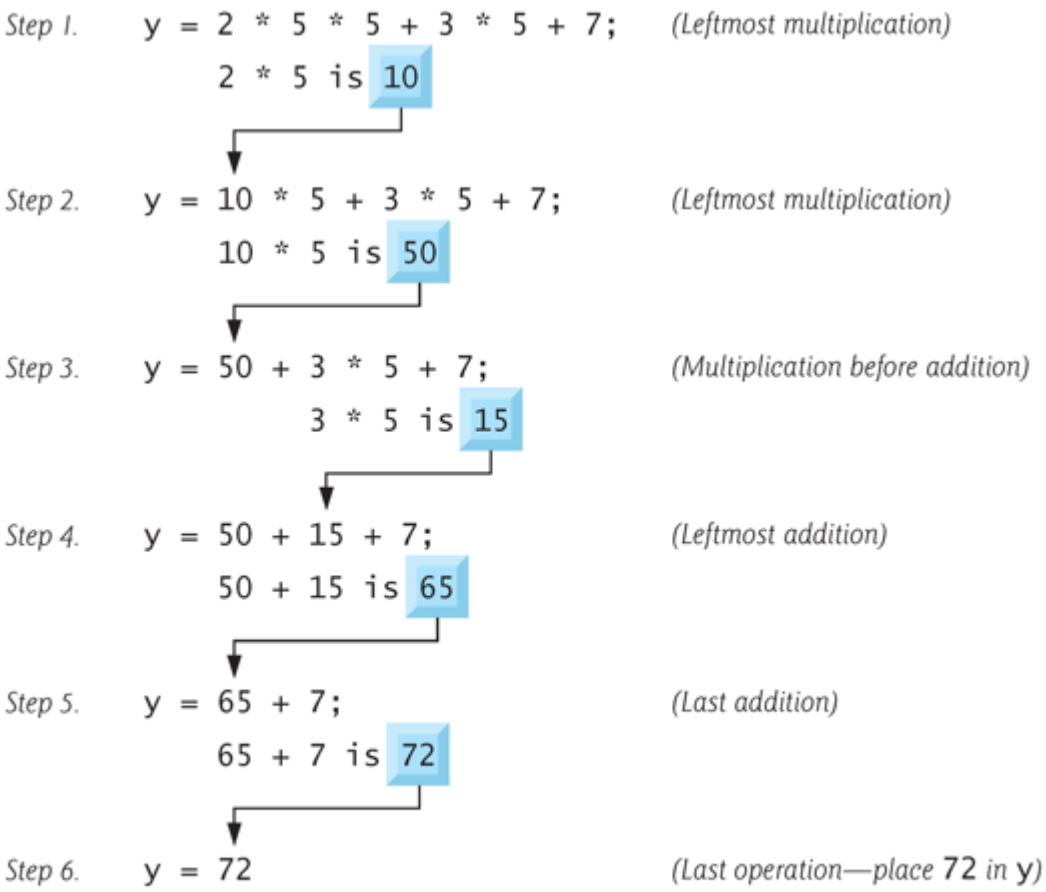


Fig. 2.11 | Order in which a second-degree polynomial is evaluated.

Other remarks

- There is no arithmetic operator for exponentiation in C++, so square of is represented as **x * x**
- Use parenthesis for grouping operations
  - For enforcing a particular precedence
  - As in algebra, it's acceptable to place unnecessary parentheses in an expression to make the expression clearer

Relational operators (like C)

Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Fig. 2.12 | Relational and equality operators.

Using the if statement

```
1  // Fig. 2.13: fig02_13.cpp
2  // Comparing integers using if statements, relational operators
3  // and equality operators.
4  #include <iostream> // allows program to perform input and output
5
6  using std::cout; // program uses cout
7  using std::cin; // program uses cin
8  using std::endl; // program uses endl
9
10 // function main begins program execution
11 int main()
12 {
13    int number1 = 0; // first integer to compare (initialized to 0)
14    int number2 = 0; // second integer to compare (initialized to 0)
15
16    cout << "Enter two integers to compare: "; // prompt user for data
17    cin >> number1 >> number2; // read two integers from user
18
19    if ( number1 == number2 )
20       cout << number1 << " == " << number2 << endl;
21
22    if ( number1 != number2 )
23       cout << number1 << " != " << number2 << endl;
24
25    if ( number1 < number2 )
26       cout << number1 << " < " << number2 << endl;
27
28    if ( number1 > number2 )
29       cout << number1 << " > " << number2 << endl;
30
31    if ( number1 <= number2 )
32       cout << number1 << " <= " << number2 << endl;
33
34    if ( number1 >= number2 )
35       cout << number1 << " >= " << number2 << endl;
36 } // end function main
```

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

Using the `if` statement

- `using` declarations eliminate the need to repeat the `std::` prefix as we did in earlier program
- Once we insert these using declarations, we can write
  - `cout` instead of `std::cout`
  - `cin` instead of `std::cin`
  - `endl` instead of `std::endl`
- `cin` is for input; `endl` is the end-of-the-line char (a better alternative for `\n`)
- From this point forward, we'll use the preceding declaration in our programs

Precedence of operator discuss so far

Operators	Associativity	Type
()	*[See caution in Fig. 2.10]*	grouping parentheses
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	stream insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
=	right to left	assignment

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

Assignment operators (like C)

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 4.17 | Arithmetic assignment operators.

Increment and decrement operators

Operator	Called	Sample expression	Explanation
++	preincrement	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postincrement	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	predecrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postdecrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 4.18 | Increment and decrement operators.

Precedence of operator discuss so far

Operators	Associativity	Type
`::` `()`	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
`++` `--` `static_cast<type>()`	left to right	postfix
`++` `--` `+` `-`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment

Fig. 4.20 | Operator precedence for the operators encountered so far in the text.

C and C++ keywords

C++ Keywords

Keywords common to the C and C++ programming languages

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

C++-only keywords

and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

Control structures

- Normally, statements in a program execute one after the other in the order in which they're written.
  - Called sequential execution
- All programs could be written in terms of only three control structures
  - the sequence structure
  - the selection structure and
  - the repetition structure
  - We'll refer to them in the terminology of the C++ standard document as control statements

Control structures

- C++ provides three types of selection statements
  - The `if` selection statement either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false
  - The `if...else` selection statement performs an action if a condition is true or performs a different action if the condition is false
  - The `switch` selection statement performs one of many different actions, depending on the value of an integer expression
  - The `switch` selection statement is called a multiple-selection statement because it selects among many different actions (or groups of actions).

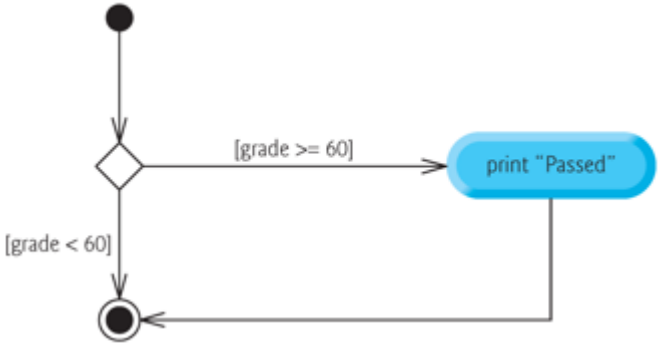
Control structures

- C++ provides three types of repetition statements (or loops) for performing statements repeatedly while a loop condition remains true
  - These are the `while`, `do...while` and `for` statements.
  - The `while` and for statements perform the action (or group of actions) in their bodies zero or more times.
  - The `do...while` statement performs the action (or group of actions) in its body at least once

The if selection statement

- An if statement can be written in C++ as

```
if ( grade >= 60 )
    cout << "Passed";
```

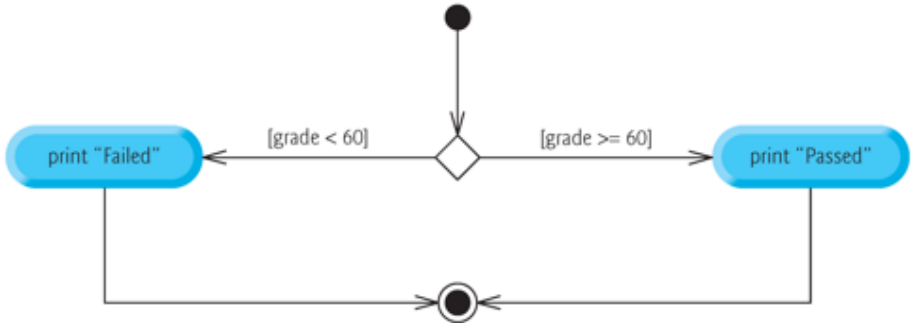


- A decision can be based on any expression—if the expression evaluates to zero, it's treated as false; if the expression evaluates to nonzero, it's treated as true.
- C++ provides the data type `bool` for variables that can hold only the values `true` and `false` each of these is a C++ keyword

The if…else double selection

- `if…else` double-selection statement specifies an action to perform when the condition is true and a different action to perform when the condition is false

```
if (grade >= 60)
   cout << "Passed";
else
   cout << "Failed";
```



Another if…else example

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
   cout << "A";
else if ( studentGrade >= 80 ) // 80-89 gets "B"
   cout << "B";
else if ( studentGrade >= 70 ) // 70-79 gets "C"
   cout << "C";
else if ( studentGrade >= 60 ) // 60-69 gets "D"
   cout << "D";
else // less than 60 gets "F"
   cout << "F";
```

Be careful of the dangling-else problem

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

Solution for the dangling-else problem

- The compiler interprets the statement as

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

- To force the nested if…else statement to execute as intended, use

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

Block and null statements

- The `if` selection statement expects only one statement in its body
- Similarly, the `if` and else parts of an `if...else` statement each expect only one body statement
- To include several statements in the body of an `if` or in either part of an `if...else`, enclose the statements in braces `{` and `}`
- A set of statements contained within a pair of braces is called a **compound** statement or a **block**
- Just as a block can be placed anywhere a single statement can be placed, it's possible to have no statement at all (called a **null** statement) and represented by a semicolon `;`

The while statement

- Consider a program segment designed to find the first power of 3 larger than 100

```
int product = 3;

while (product <= 100)
    product = 3 * product;
```

Typical exam/quiz question

An empty statement

```
If (x ==3)
   ;
else
   print(“Test\n”);
```

Counter-controlled loop

```
1  // Fig. 5.1: fig05_01.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int counter = 1; // declare and initialize control variable
9
10    while ( counter <= 10 ) // loop-continuation condition
11    {
12       cout << counter << " ";
13       ++counter; // increment control variable by 1
14    } // end while
15
16    cout << endl; // output a newline
17 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.1 | Counter-controlled repetition.

The for statement

- The `for` statement specifies the counter-controlled repetition details in a single line of code
  - The initialization occurs once when the loop is encountered
  - The condition is tested next and each time the body completes
  - The body executes if the condition is true
  - The increment occurs after the body executes
  - Then, the condition is tested again
  - If there is more than one statement in the body of the for, braces are required to enclose the body of the loop

The for statement

```
1  // Fig. 5.2: fig05_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // for statement header includes initialization,
9     // loop-continuation condition and increment.
10    for ( unsigned int counter = 1; counter <= 10; ++counter )
11       cout << counter << " ";
12
13    cout << endl; // output a newline
14 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.2 | Counter-controlled repetition with the for statement.

A closer look at the for statement

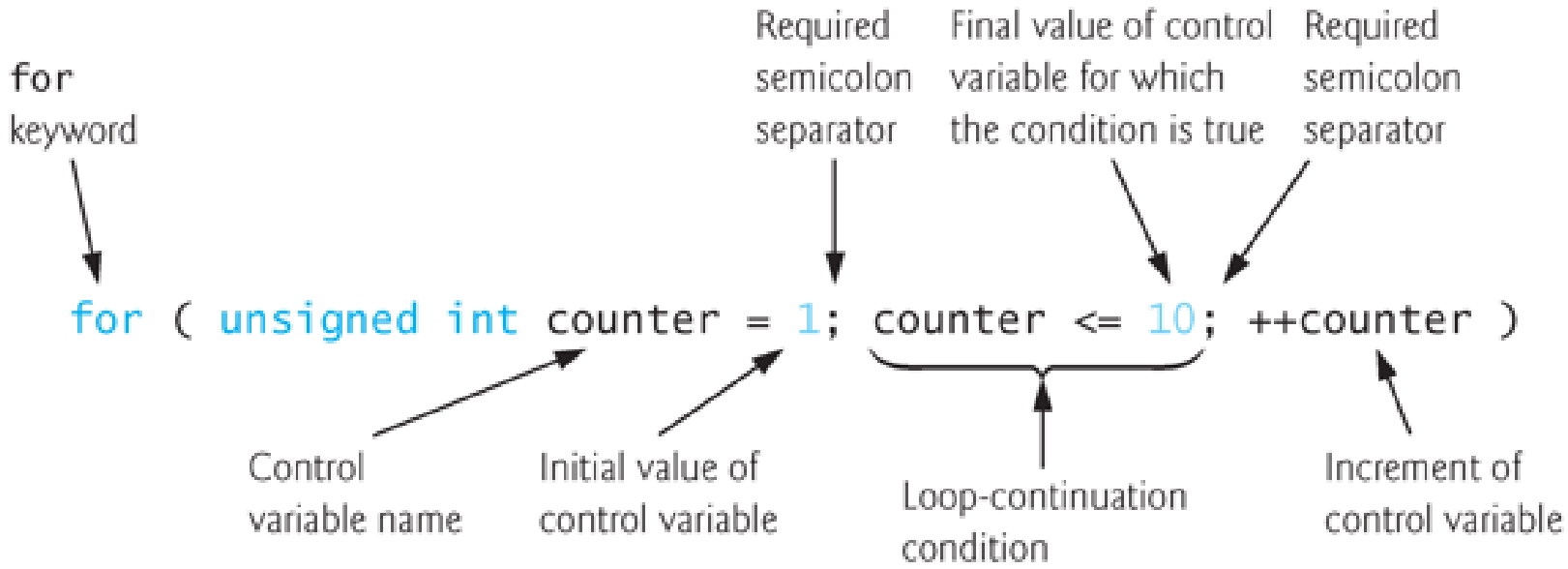


Fig. 5.3 | for statement header components.

More about the for statement

- The three expressions in the `for` statement header are optional (but the two semicolon separators are required)
  - If the loop condition is omitted, C++ assumes that the condition is true, thus creating an infinite loop
  - One might omit the initialization expression if the control variable is initialized earlier in the program
  - One might omit the increment expression if the increment is calculated by statements in the body of the for or if no increment is needed

More examples of the for statement

```
for ( unsigned int i = 1; i <= 100; ++i )...

for ( int i = 100; i >= 0; --i )...

for (unsigned int i = 7; i <= 77; i += 7 )...

for (unsigned int i = 20; i >= 2; i -= 2 )...

for (unsigned int i = 99; i >= 55; i -= 11 )
```

The do...while statement

- Similar to the **while** statement
- The **do...while** statement tests the loop continuation condition after the loop body executes, the loop body always executes at least once

```
1  // Fig. 5.7: fig05_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int counter = 1; // initialize counter
9
10    do
11    {
12       cout << counter << " "; // display counter
13       ++counter; // increment counter
14    } while ( counter <= 10 ); // end do...while
15
16    cout << endl; // output a newline
17 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

The do...while statement

- A `do...while` with no braces around the single statement body

```
do
    statement
while (condition);
```

- It is recommended to always use a brace

```
do
{
  statement
} while ( condition );
```

The switch statement

- The switch multiple-selection statement performs many different actions based on the possible values of a variable or expression
- Each action is associated with the value of an integral constant expression (i.e., any combination of character and integer constants that evaluates to a constant integer value)

A sample switch statement

```
56     // loop until user types end-of-file key sequence
57     while ( ( grade = cin.get() ) != EOF )
58     {
59        // determine which grade was entered
60        switch ( grade ) // switch statement nested in while
61        {
62           case 'A': // grade was uppercase A
63           case 'a': // or lowercase a
64              ++aCount; // increment aCount
65              break; // necessary to exit switch
66
67           case 'B': // grade was uppercase B
68           case 'b': // or lowercase b
69              ++bCount; // increment bCount
70              break; // exit switch
71
72           case 'C': // grade was uppercase C
73           case 'c': // or lowercase c
74              ++cCount; // increment cCount
75              break; // exit switch
76
77           case 'D': // grade was uppercase D
78           case 'd': // or lowercase d
79              ++dCount; // increment dCount
80              break; // exit switch
81
82           case 'F': // grade was uppercase F
83           case 'f': // or lowercase f
84              ++fCount; // increment fCount
85              break; // exit switch
86
87           case '\n': // ignore newlines,
88           case '\t': // tabs,
89           case ' ': // and spaces in input
90              break; // exit switch
91
92           default: // catch all other characters
93              cout << "Incorrect letter grade entered."
94                 << " Enter a new grade." << endl;
95              break; // optional; will exit switch anyway
96        } // end switch
97     } // end while
98  } // end function inputGrades
```

Some notes about the case statement

- The `cin.get()` function reads one character from the keyboard
- `EOF` stands for "end-of-fil"; it is used as a sentinel value
- The `switch` statement consists of a series of case labels and an optional default case
- The `switch` statement compares the value of the controlling expression with each `case` label
- If a match occurs, the program executes the statements for that `case`
- The `break` statement causes program control to proceed with the first statement after the switch

Some notes about the case statement

- Listing `case`s consecutively with no statements between them enables the `case`s to perform the same set of statements
- Each `case` can have multiple statements
- The `switch` selection statement does not require braces around multiple statements in each case
- Without `break` statements, each time a match occurs in the `switch`, the statements for that `case` and subsequent `case`s execute until a `break` statement or the end of the `switch` is encountered

Some notes about the case statement

- If no match occurs between the controlling expression's value and a `case` label, the `default` case executes.
- If no match occurs in a `switch` statement that does not contain a `default` case, program control continues with the first statement after the `switch`
- The `break` statement, when executed in a `while`, `for`, `do...while` or switch statement, causes immediate exit from that statement

Using the break in a for statement

```
1  // Fig. 5.13: fig05_13.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int count; // control variable also used after loop terminates
9
10    for ( count = 1; count <= 10; ++count ) // loop 10 times
11    {
12       if ( count == 5 )
13          break; // break loop only if count is 5
14
15       cout << count << " ";
16    } // end for
17
18    cout << "\nBroke out of loop at count = " << count << endl;
19 } // end main
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 5.13 | break statement exiting a for statement.

The continue statement

- The `continue` statement, when executed in a `while`, `for` or `do...while` statement, skips the remaining statements in the body of that statement and proceeds with the next iteration of the loop
  - In `while` and `do...while` statements, the loop continuation test evaluates immediately after the `continue` statement executes
  - In the `for` statement, the increment expression executes, then the loop-continuation test evaluates

The continue statement

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     for ( unsigned int count = 1; count <= 10; ++count ) // loop 10 times
9     {
10       if ( count == 5 ) // if count is 5,
11          continue;      // skip remaining code in loop
12
13       cout << count << " ";
14    } // end for
15
16    cout << "\nUsed continue to skip printing 5" << endl;
17 } // end main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 5.14 | continue statement terminating an iteration of a for statement.

Logical operators

- C++ provides logical operators that are used to form more complex conditions by combining simple conditions.
- The logical operators are **&&** (logical AND), **||** (logical OR) and **!** (logical NOT, also called logical negation)
  - C++ provides the **!** (logical NOT, also called logical negation) operator to "reverse" a condition's meaning

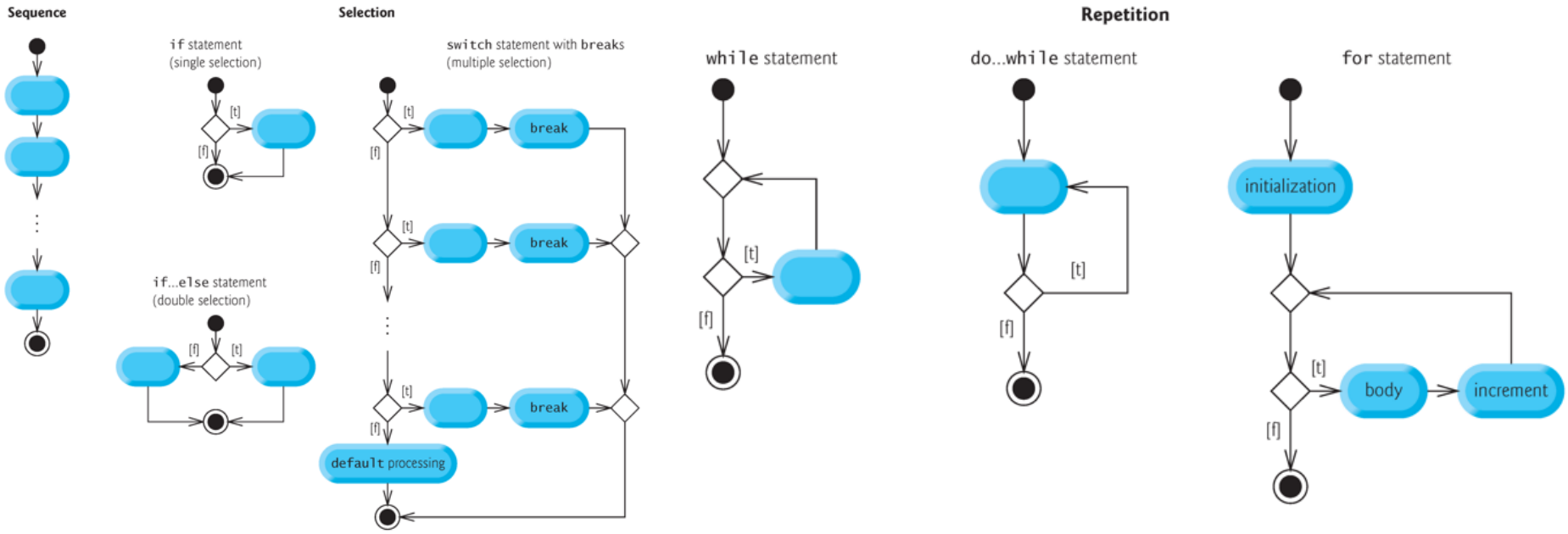
Operator precedence

Operators	Associativity	Type
`::` `()`	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
`++` `--` `static_cast< type >()`	left to right	postfix
`++` `--` `+` `-` `!`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 5.19 | Operator precedence and associativity.

Structured programming

- Structured programming produces programs that are easier than unstructured programs to understand, test, debug, modify, and even prove correct in a mathematical sense



The math library

- The <**cmath**> header file provides a collection of functions that enable you to perform common mathematical calculations

Function	Description	Example
`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 ) is 10.0` `ceil( -9.8 ) is -9.0`
`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 ) is 1.0`
`exp( x )`	exponential function e^x	`exp( 1.0 ) is 2.718282` `exp( 2.0 ) is 7.389056`

`fabs( x )`	absolute value of x	`fabs( 5.1 ) is 5.1`
		`fabs( 0.0 ) is 0.0`
		`fabs( -8.76 ) is 8.76`

`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 ) is 9.0` `floor( -9.8 ) is -10.0`
`fmod( x, y )`	remainder of x/y as a floating-point number	`fmod( 2.6, 1.2 ) is 0.2`
`log( x )`	natural logarithm of x (base e)	`log( 2.718282 ) is 1.0` `log( 7.389056 ) is 2.0`

Function	Description	Example
`log10( x )`	logarithm of x (base 10)	`log10( 10.0 ) is 1.0` `log10( 100.0 ) is 2.0`
`pow( x, y )`	x raised to power y (x^y)	`pow( 2, 7 ) is 128` `pow( 9, .5 ) is 3`
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 ) is 0`
`sqrt( x )`	square root of x (where x is a nonnegative value)	`sqrt( 9.0 ) is 3.0`
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 ) is 0`

A function prototype

- Also known as a function prototype (also called a function declaration)
- A function prototype tells the compiler
  - the name of a function
  - the type of data returned by the function
  - the number of parameters the function expects to receive
  - the types of those parameters and
  - the order in which the parameters of those types are expected

A function signature

- The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature or simply the *signature*
  - Signature does not specify the function's return type
- The scope of a function is the region of a program in which the function is known and accessible

Function headers

- The header files contain the function prototypes for the related functions that form each portion of the library
- The header files also contain definitions of various class types and functions, as well as constants needed by those functions
- Some examples

Standard Library header	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.9 and is discussed in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
`<cmath>`	Contains function prototypes for math library functions (Section 6.3).
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class `string`; Chapter 17, Exception Handling: A Deeper Look; Chapter 22, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.

Storage classes

- Keywords **`extern`** and **`static`** declare identifiers for variables with static storage duration
  - Exist from the point at which the program begins execution and until the program terminates
  - **`extern`**: visibility from outside; **`static`**: visibility within same file
- Local variables declared **`static`** are still known only in the function in which they're declared, but, unlike automatic variables, **`static`** local variables retain their values when the function returns to its caller

Storage classes

- Local variables declared `static` are still known only in the function in which they're declared, but, unlike automatic variables, `static` local variables retain their values when the function returns to its caller
  - The next time the function is called, the `static` local variables contain the values they had when the function last completed execution

Scope rules

- The portion of the program where an identifier can be used is known as its scope
  - Block scope
  - Function scope
  - Global namespace scope
    * An identifier declared outside any function or class has global namespace scope
  - Function-prototype scope

Function call stack

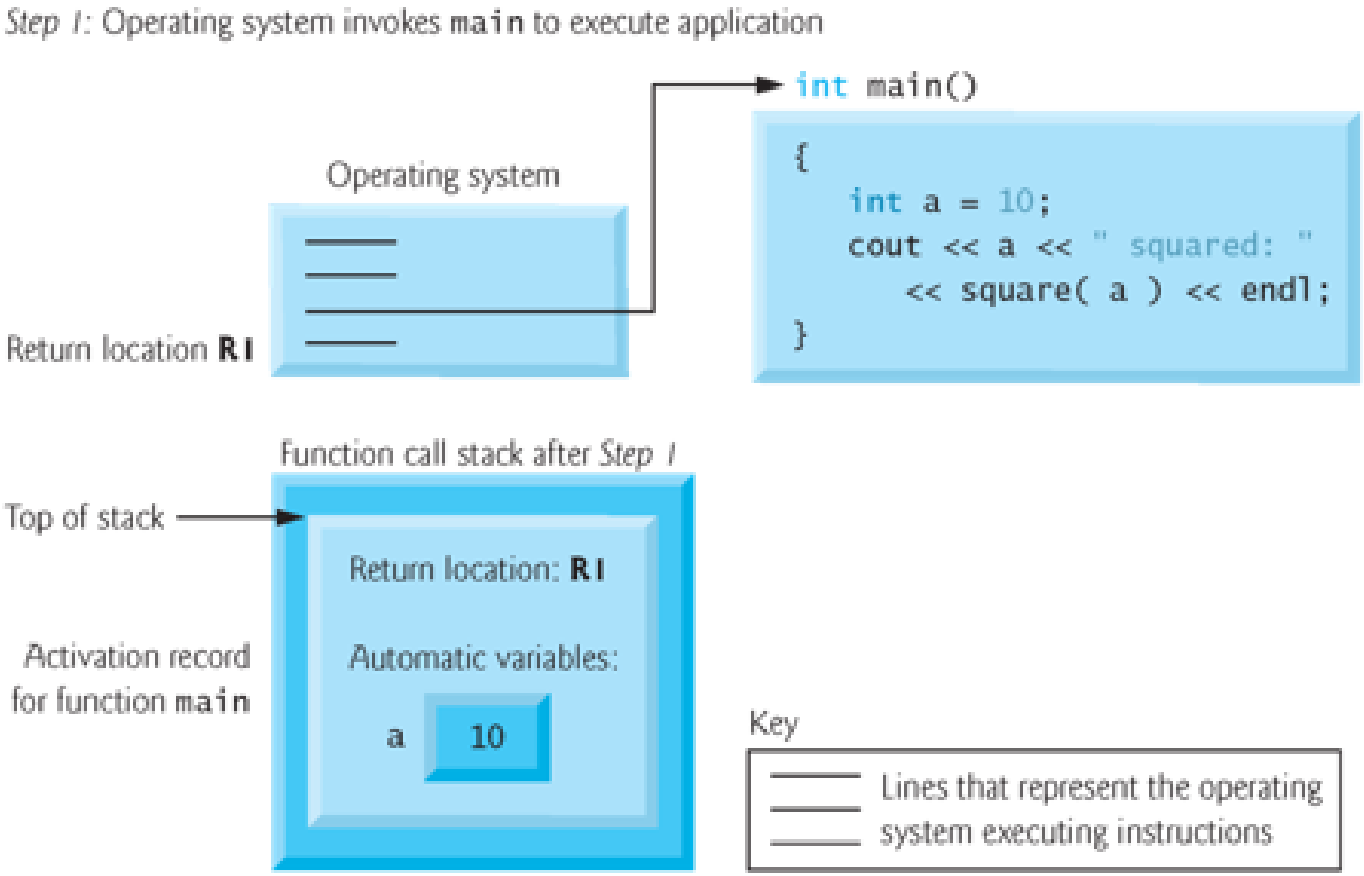
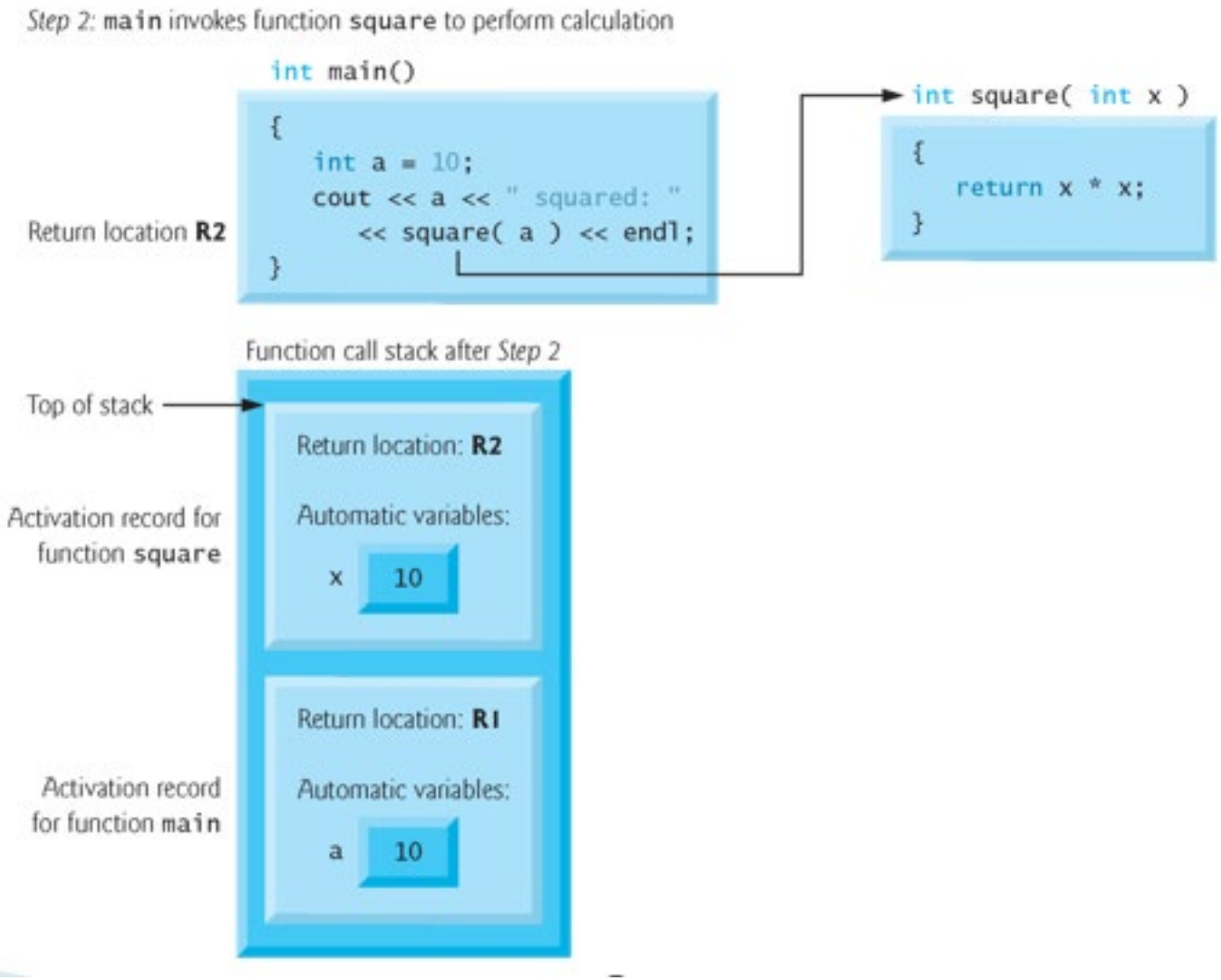


Fig. 6.15 | Function call stack after the operating system invokes `main` to execute the program.

Function call stack



Function call stack

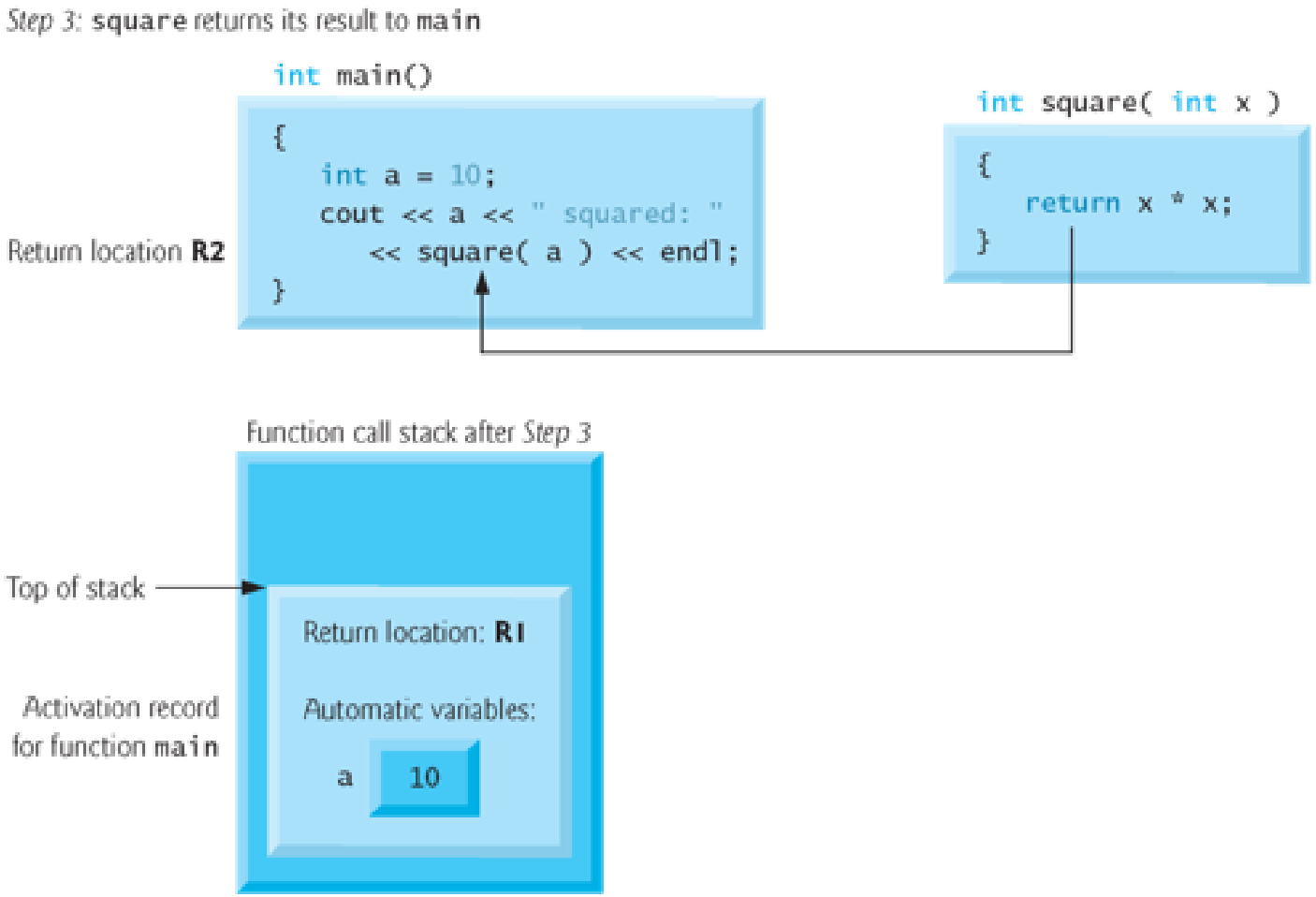


Fig. 6.17 | Function call stack after function `square` returns to `main`.

Functions with no parameters

- In C++, an empty parameter list is specified by writing either void or nothing at all in parentheses

```
1  // Fig. 6.18: fig06_18.cpp
2  // Functions that take no arguments.
3  #include <iostream>
4  using namespace std;
5
6  void function1(); // function that takes no arguments
7  void function2( void ); // function that takes no arguments
8
9  int main()
10 {
11    function1(); // call function1 with no arguments
12    function2(); // call function2 with no arguments
13 } // end main
14
15 // function1 uses an empty parameter list to specify that
16 // the function receives no arguments
17 void function1()
18 {
19    cout << "function1 takes no arguments" << endl;
20 } // end function1
21
22 // function2 uses a void parameter list to specify that
23 // the function receives no arguments
24 void function2( void )
25 {
26    cout << "function2 also takes no arguments" << endl;
27 } // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

Inline functions

- C++ provides inline functions to help reduce function call overhead
- Placing the qualifier **`inline`** before a function's return type in the function definition advises the compiler to generate a copy of the function's code in every place where the function is called

```
1  // Fig. 6.19: fig06_19.cpp
2  // inline function that calculates the volume of a cube.
3  #include <iostream>
4  using namespace std;
5
6  // Definition of inline function cube. Definition of function appears
7  // before function is called, so a function prototype is not required.
8  // First line of function definition acts as the prototype.
9  inline double cube( const double side )
10 {
11    return side * side * side; // calculate cube
12 } // end function cube
13
14 int main()
15 {
16    double sideValue; // stores value entered by user
17    cout << "Enter the side length of your cube: ";
18    cin >> sideValue; // read value from user
19
20    // calculate cube of sideValue and display result
21    cout << "Volume of cube with side "
22       << sideValue << " is " << cube( sideValue ) << endl;
23 } // end main
```

Fig. 6.19 | `inline` function that calculates the volume of a cube. (Part 1 of 2.)

Reference parameters

- Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference
- When an argument is passed by value, a copy of the argument's value is made and passed (on the function call stack) to the called function
  - Changes to the copy do not affect the original variable's value in the caller
- To specify a reference to a constant, place the **`const`** qualifier before the type specifier in the parameter declaration

Reference parameters

- With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data
- A reference parameter is an alias for its corresponding argument in a function call
- To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header

Reference parameters

```
1  // Fig. 6.20: fig06_20.cpp
2  // Passing arguments by value and by reference.
3  #include <iostream>
4  using namespace std;
5
6  int squareByValue( int ); // function prototype (value pass)
7  void squareByReference( int & ); // function prototype (reference pass)
8
9  int main()
10 {
11    int x = 2; // value to square using squareByValue
12    int z = 4; // value to square using squareByReference
13
14    // demonstrate squareByValue
15    cout << "x = " << x << " before squareByValue\n";
16    cout << "Value returned by squareByValue: "
17       << squareByValue( x ) << endl;
18    cout << "x = " << x << " after squareByValue\n" << endl;
19
20    // demonstrate squareByReference
21    cout << "z = " << z << " before squareByReference" << endl;
22    squareByReference( z );
23    cout << "z = " << z << " after squareByReference" << endl;
24 } // end main
```

Fig. 6.20 | Passing arguments by value and by reference. (Part 1 of 2.)

Reference parameters

```
25
26  // squareByValue multiplies number by itself, stores the
27  // result in number and returns the new value of number
28  int squareByValue( int number )
29  {
30     return number *= number; // caller's argument not modified
31  } // end function squareByValue
32
33  // squareByReference multiplies numberRef by itself and stores the result
34  // in the variable to which numberRef refers in function main
35  void squareByReference( int &numberRef )
36  {
37     numberRef *= numberRef; // caller's argument modified
38  } // end function squareByReference
```

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

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

Fig. 6.20 | Passing arguments by value and by reference. (Part 2 of 2.)

Default arguments

```
1  // Fig. 6.21: fig06_21.cpp
2  // Using default arguments.
3  #include <iostream>
4  using namespace std;
5
6  // function prototype that specifies default arguments
7  unsigned int boxVolume( unsigned int length = 1, unsigned int width = 1,
8     unsigned int height = 1 );
9
```

Fig. 6.21 | Using default arguments. (Part 1 of 3.)

Default arguments

```
10  int main()
11  {
12     // no arguments--use default values for all dimensions
13     cout << "The default box volume is: " << boxVolume();
14
15     // specify length; default width and height
16     cout << "\n\nThe volume of a box with length 10,\n"
17        << "width 1 and height 1 is: " << boxVolume( 10 );
18
19     // specify length and width; default height
20     cout << "\n\nThe volume of a box with length 10,\n"
21        << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23     // specify all arguments
24     cout << "\n\nThe volume of a box with length 10,\n"
25        << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26        << endl;
27  } // end main
28
29  // function boxVolume calculates the volume of a box
30  unsigned int boxVolume( unsigned int length, unsigned int width,
31     unsigned int height )
32  {
33     return length * width * height;
34  } // end function boxVolume
```

Fig. 6.21 | Using default arguments. (Part 2 of 3.)

Unary scope resolution operator

- It's possible to declare local and global variables of the same name
- C++ provides the unary scope resolution operator (`::`) to access a global variable when a local variable of the same name is in scope

```
1  // Fig. 6.22: fig06_22.cpp
2  // Unary scope resolution operator.
3  #include <iostream>
4  using namespace std;
5
6  int number = 7; // global variable named number
7
8  int main()
9  {
10    double number = 10.5; // local variable named number
11
12    // display values of local and global variables
13    cout << "Local double value of number = " << number
14       << "\nGlobal int value of number = " << ::number << endl;
15 } // end main
```

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

Function overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call
- Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types

Function overloading

```
1  // Fig. 6.23: fig06_23.cpp
2  // Overloaded square functions.
3  #include <iostream>
4  using namespace std;
5
6  // function square for int values
7  int square( int x )
8  {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11 } // end function square with int argument
12
13 // function square for double values
14 double square( double y )
15 {
16    cout << "square of double " << y << " is ";
17    return y * y;
18 } // end function square with double argument
19
20 int main()
21 {
22    cout << square( 7 ); // calls int version
23    cout << endl;
24    cout << square( 7.5 ); // calls double version
25    cout << endl;
26 } // end main
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 6.23 | Overloaded square functions. (Part 2 of 2.)

Function templates

- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates
- You write a single function template definition
- All function template definitions begin with the `template` keyword followed by a template parameter list to the function template enclosed in angle brackets (< and >).

Function templates

```
1  // Fig. 6.25: maximum.h
2  // Function template maximum header.
3  template < typename T >  // or template< class T >
4  T maximum( T value1, T value2, T value3 )
5  {
6     T maximumValue = value1; // assume value1 is maximum
7
8     // determine whether value2 is greater than maximumValue
9     if ( value2 > maximumValue )
10       maximumValue = value2;
11
12    // determine whether value3 is greater than maximumValue
13    if ( value3 > maximumValue )
14       maximumValue = value3;
15
16    return maximumValue;
17 } // end function template maximum
```

Fig. 6.25 | Function template `maximum` header.

Data types in C++

Name	Name	Value
char	unsigned char	8-bit integer
short	unsigned short	16-bit integer
int	unsigned int	32-bit integer
long	unsigned long	64-bit integer
bool		true or false

Name	Value
float	32-bit floating point
double	64-bit floating point
long long	128-bit integer
long double	128-bit floating point

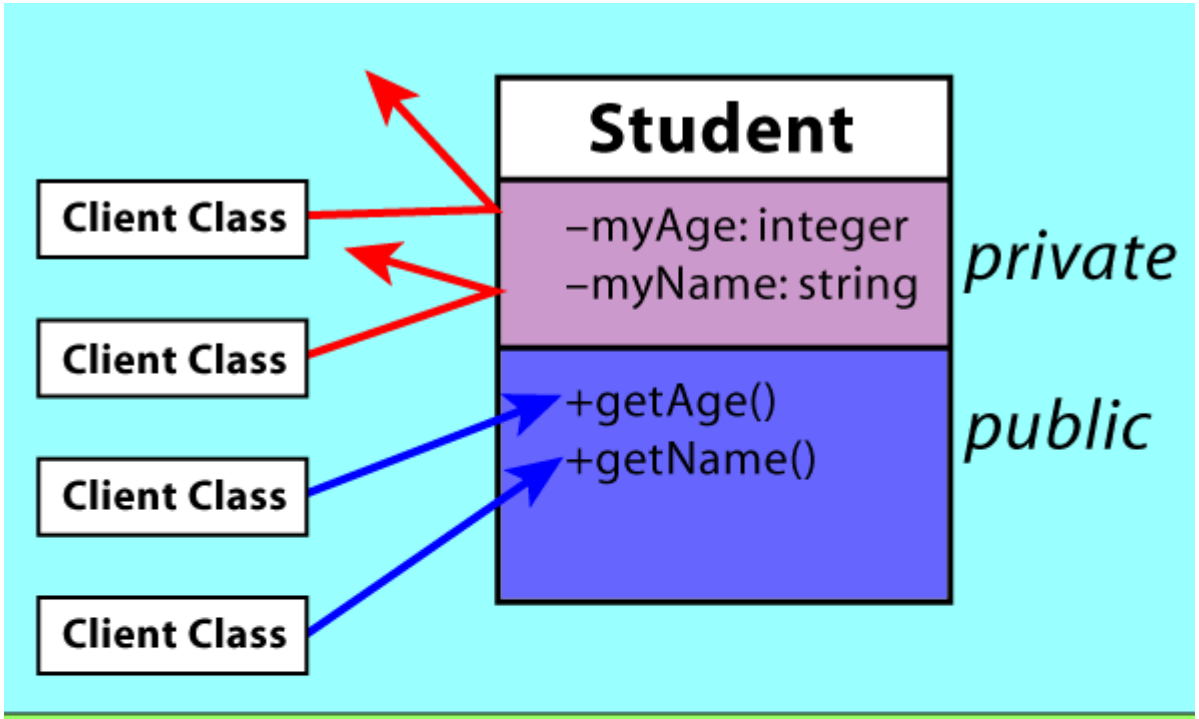
- Type casting
  - C++ is strongly typed; it will auto-convert a variable of one type to another in a limited fashion (won't change the value)

```
short x = 1 ;
int y = x ;    // OK
short z = y ; // NO!
```

Object-orientation in C++

- Classes
  - Defines a set of data items and structures (data members)
  - Defines a set of operations (operation members)
    * Also called methods
    * Also called functions
    * Also called services
    * Also called a class behavior
  - Encapsulates the two
  - Indicates which of the above members are accessible (public) and which are inaccessible (private)
    * Use of access specifier: `public`, `private`, `protected`
- A class defines an abstract data type
  - Examples: `course`, `student`, `teacher`, `book`, `account`, `printer`

Object-orientation in C++



In C++:

```
class Student {
public:
   int getAge();
   string getName();
private:
   int myAge;
   string myName;
}
```

C++ classes

```
1  class Time {
2  public:
3     Time();
4     void setTime( int, int, int );
5     void printMilitary();
6     void printStandard();
7  private:
8     int hour;      // 0 - 23
9     int minute;    // 0 - 59
10    int second;    // 0 - 59
11 };
```

Public: and **Private:** are member-access specifiers.

setTime, **printMilitary**, and **printStandard** are **member functions**.
Time is the **constructor.**

hour, **minute**, and **second** are **data members**.

Public and private members

- Member access specifiers
  - Classes can limit the access to their member functions and data
  - The three types of access a class can grant
  - Public: Accessible wherever the program has access to an object of the class
  - Private: accessible only to member functions of the class
  - Protected: Similar to private; discussed later

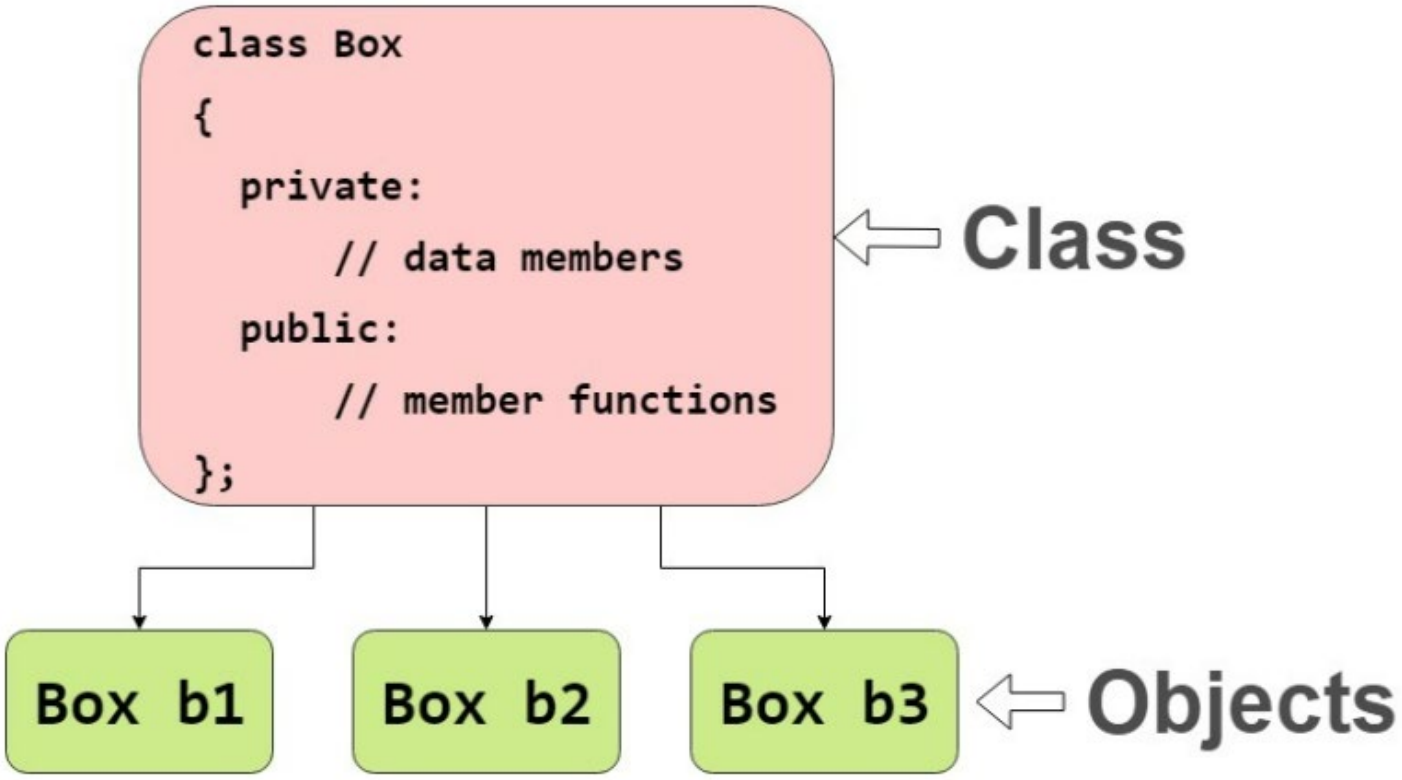
The constructor and destructor methods

- The constructor function
  - A special member function that initializes the data members of a class object
  - Cannot return values
  - The same name as the class
- The destructor function
  - A special function with the same name as the class but preceded with a tilde character (~)
  - Cannot take arguments and cannot be overloaded
  - Performs termination housekeeping

Class and object definition

```
class Box {
private:
     // data members
     // function members
public:
     // function members
};

Box Box1;     // Declare Box1 of type Box
Box Box2;     // Declare Box2 of type Box
```

Classes vs objects



Accessing class members

- Dot (.) for objects and arrow (->) for pointers
  - Assume an object **t**

```
t.hour
t.printMilitary();
```

Defining a class

```
1  // Fig. 3.1: fig03_01.cpp
2  // Define class GradeBook with a member function displayMessage,
3  // create a GradeBook object, and call its displayMessage function.
4  #include <iostream>
5  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    // function that displays a welcome message to the GradeBook user
12    void displayMessage() const
13    {
14       cout << "Welcome to the Grade Book!" << endl;
15    } // end function displayMessage
16 }; // end class GradeBook
17
18 // function main begins program execution
19 int main()
20 {
21    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
22    myGradeBook.displayMessage(); // call object's displayMessage function
23 } // end main
```

Defining a class

```
1  // Fig. 6.3: fig06_03.cpp
2  // Time class.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // Time abstract data type (ADT) definition
9  class Time {
10 public:
11    Time();                          // constructor
12    void setTime( int, int, int ); // set hour, minute, second
13    void printMilitary();            // print military time format
14    void printStandard();            // print standard time format
15 private:
16    int hour;     // 0 - 23
17    int minute;   // 0 - 59
18    int second;   // 0 - 59
19 };
20
21 // Time constructor initializes each data member to zero.
22 // Ensures all Time objects start in a consistent state.
23 Time::Time() { hour = minute = second = 0; }
24
25 // Set a new Time value using military time. Perform validity
26 // checks on the data values. Set invalid values to zero.
27 void Time::setTime( int h, int m, int s )
28 {
29    hour = ( h >= 0 && h < 24 ) ? h : 0;
30    minute = ( m >= 0 && m < 60 ) ? m : 0;
31    second = ( s >= 0 && s < 60 ) ? s : 0;
32 }
```

Note the `::` preceding the function names.

Defining a class

```
33
34 // Print Time in military format
35 void Time::printMilitary()
36 {
37    cout << ( hour < 10 ? "0" : "" ) << hour << ":"
38         << ( minute < 10 ? "0" : "" ) << minute;
39 }
40
41 // Print Time in standard format
42 void Time::printStandard()
43 {
44    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
45         << ":" << ( minute < 10 ? "0" : "" ) << minute
46         << ":" << ( second < 10 ? "0" : "" ) << second
47         << ( hour < 12 ? " AM" : " PM" );
48 }
49
50 // Driver to test simple class Time
51 int main()
52 {
53    Time t;  // instantiate object t of class Time
54
55    cout << "The initial military time is ";
56    t.printMilitary();
57    cout << "\nThe initial standard time is ";
58    t.printStandard();
59
```

Defining a class

```
60     t.setTime( 13, 27, 6 );
61     cout << "\n\nMilitary time after setTime is ";
62     t.printMilitary();
63     cout << "\nStandard time after setTime is ";
64     t.printStandard();
65
66     t.setTime( 99, 99, 99 );  // attempt invalid settings
67     cout << "\n\nAfter attempting invalid settings:"
68          << "\nMilitary time: ";
69     t.printMilitary();
70     cout << "\nStandard time: ";
71     t.printStandard();
72     cout << endl;
73     return 0;
74 }
```

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

Initializing class objects

- Constructors
  - Initialize class members
  - Same name as the class
  - No return type
  - Member variables can be initialized by the constructor or set afterwards
- Passing arguments to a constructor
  - When an object of a class is declared, initializers can be provided
  - Format of declaration with initializers:

```
Class-type ObjectName( value1,value2,…);
```

  - Default arguments may also be specified in the constructor prototype

Initializing class objects

```
10 // Time abstract data type definition
11 class Time {
12 public:
13    Time( int = 0, int = 0, int = 0 );  // default constructor
14    void setTime( int, int, int ); // set hour, minute, second
15    void printMilitary();          // print military time format
16    void printStandard();          // print standard time format
17 private:
18    int hour;     // 0 - 23
19    int minute;   // 0 - 59
20    int second;   // 0 - 59
21 };
22
23 #endif
```

```
71 int main()
72 {
73    Time t1,              // all arguments defaulted
74         t2(2),           // minute and second defaulted
75         t3(21, 34),      // second defaulted
76         t4(12, 25, 42), // all values specified
77         t5(27, 74, 99); // all bad values specified
```

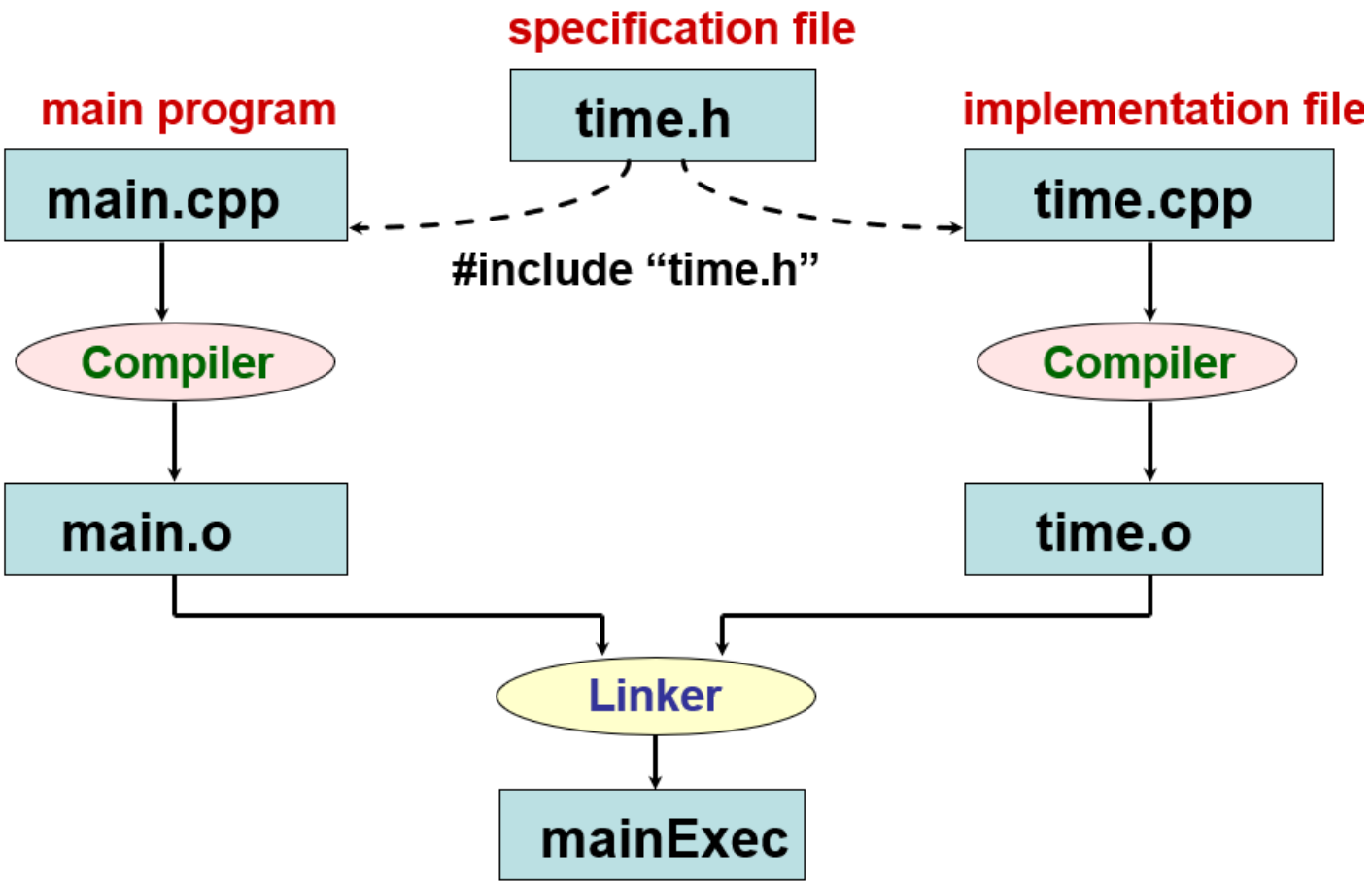
Another example

```
#include <iostream.h>

class circle
{
   private:
      double radius;

   public:
      void store(double);
      double area(void);
      void display(void);

};
```

```
// member function definitions

void circle::store(double r)
{
   radius = r;
}
double circle::area(void)
{
   return 3.14*radius*radius;
}
void circle::display(void)
{
   cout << "r = " << radius <<
   endl;
}
```

```
int main(void) {
   circle c;   // an object of circle class
   c.store(5.0);
   cout << "The area of circle c is " << c.area() << endl;
   c.display();
}
```

Separate compilation and linking



To be continued

- Contents mostly from Deitel and Deitel, *C++ How to Program*, Pearson, 2016