Chapter 2

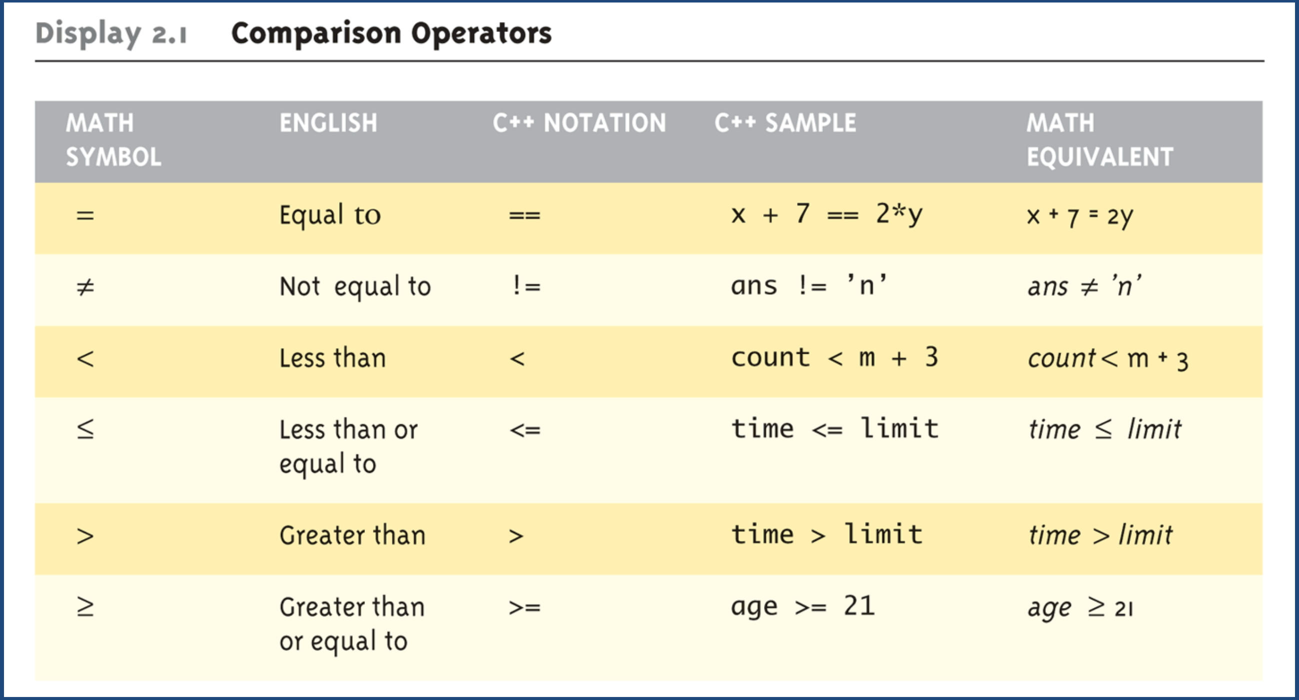
Flow of Control

Learning Objectives

- Boolean Expressions
  - Building, Evaluating & Precedence Rules
- Branching Mechanisms
  - if-else
  - switch
  - Nesting if-else
- Loops
  - While, do-while, for
  - Nesting loops
- Introduction to File Input

Boolean Expressions: **Display 2.1** Comparison Operators

- Logical Operators
  - Logical AND (&&)
  - Logical OR (||)



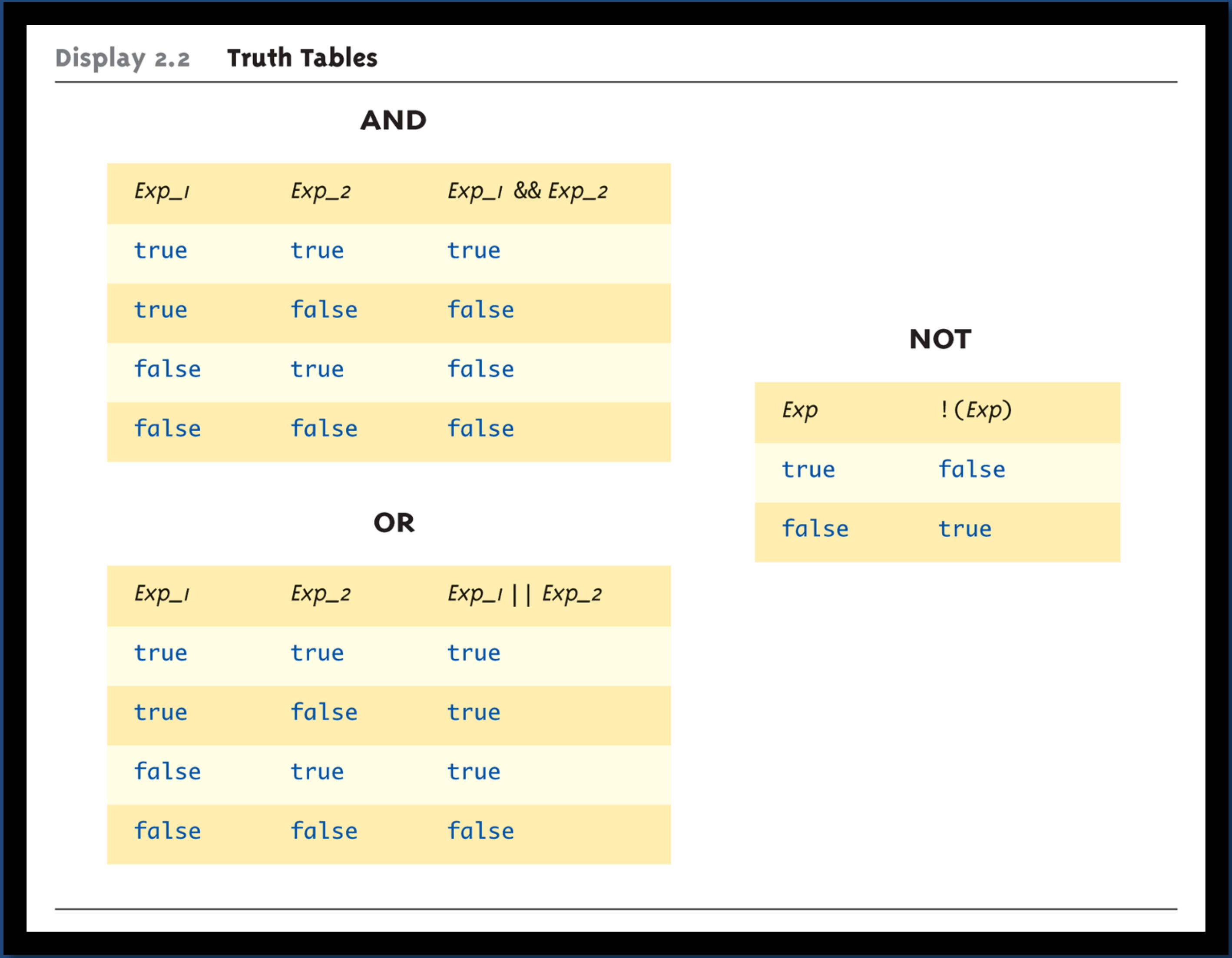
Display 2.1 Comparison Operators

MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	`==`	`x + 7 == 2*y`	$x + 7 = 2y$
≠	Not equal to	`!=`	`ans != 'n'`	$ans \neq 'n'$
<	Less than	`<`	`count < m + 3`	$count < m + 3$
≤	Less than or equal to	`<=`	`time <= limit`	$time \leq limit$
>	Greater than	`>`	`time > limit`	$time > limit$
≥	Greater than or equal to	`>=`	`age >= 21`	$age \geq 21$

Evaluating Boolean Expressions

- Data type bool
  - Returns true or false
  - true, false are predefined library consts
- Truth tables
  - Display 2.2 next slide

Evaluating Boolean Expressions: **Display 2.2** Truth Tables



Display 2.2 **Truth Tables**

AND

Exp_1	*Exp_2*	*Exp_1* && *Exp_2*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

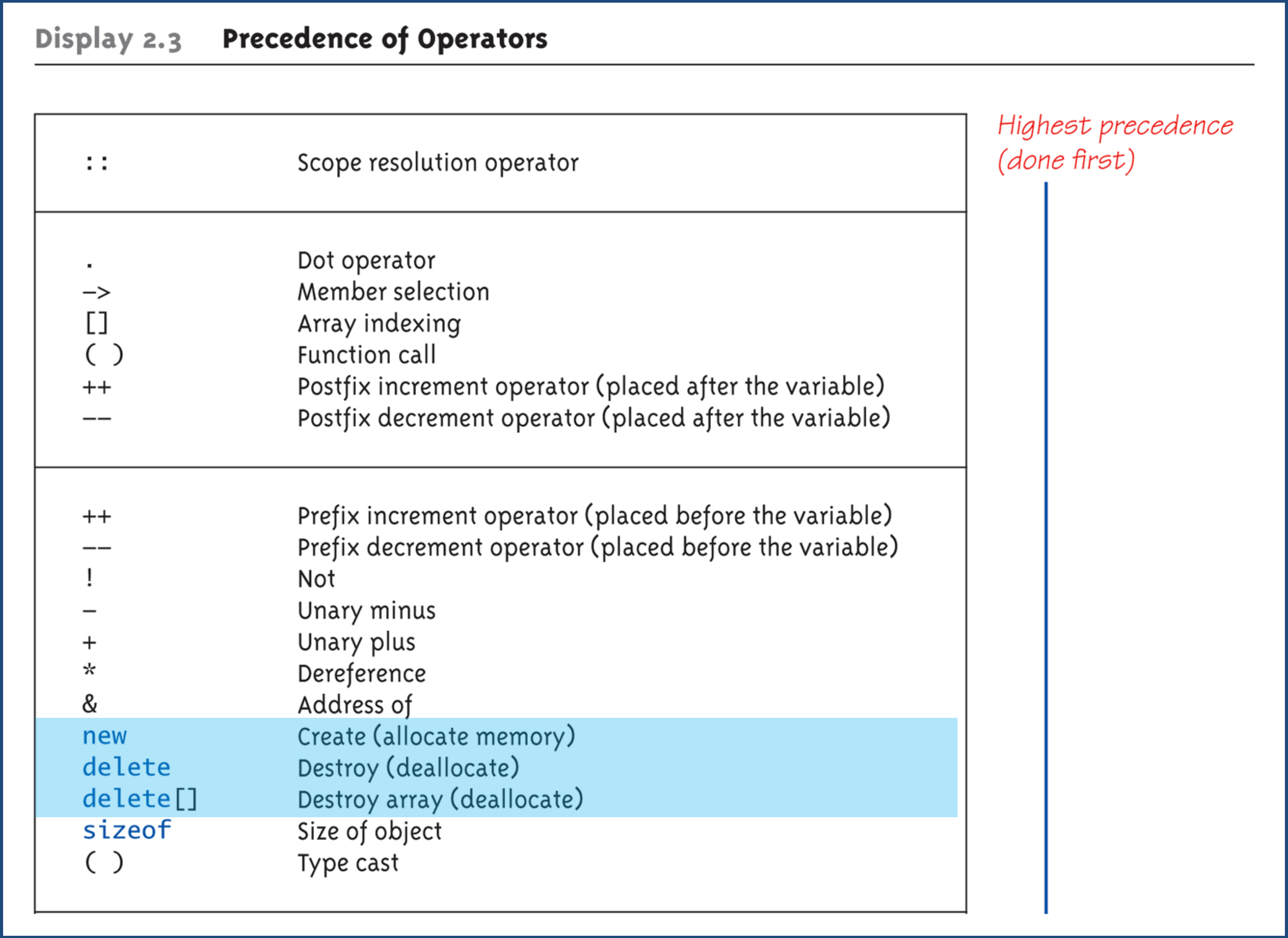
OR

Exp_1	*Exp_2*	*Exp_1* || *Exp_2*
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

NOT

Exp	!(*Exp*)
`true`	`false`
`false`	`true`

Display 2.3
Precedence of Operators (1 of 4)

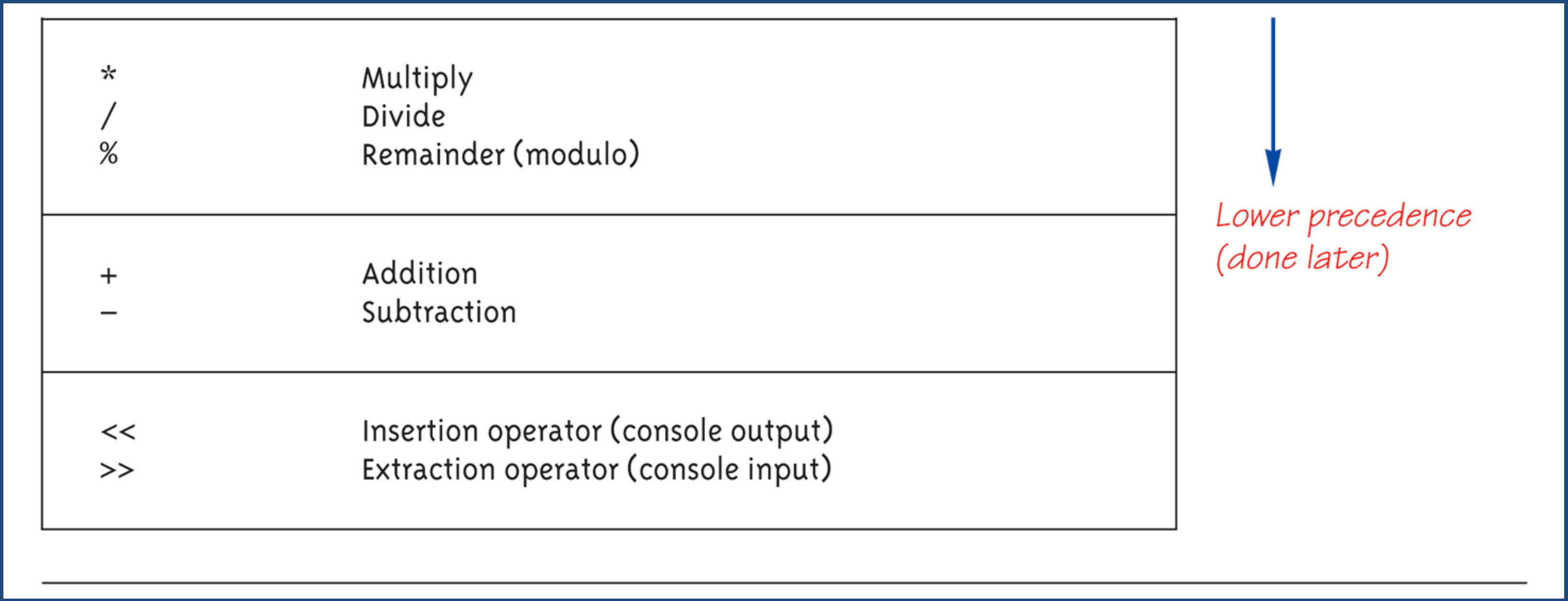


Display 2.3 Precedence of Operators

Highest precedence (done first)

`::`	Scope resolution operator
`.`	Dot operator
`->`	Member selection
`[]`	Array indexing
`( )`	Function call
`++`	Postfix increment operator (placed after the variable)
`--`	Postfix decrement operator (placed after the variable)
`++`	Prefix increment operator (placed before the variable)
`--`	Prefix decrement operator (placed before the variable)
`!`	Not
`-`	Unary minus
`+`	Unary plus
`*`	Dereference
`&`	Address of
`new`	Create (allocate memory)
`delete`	Destroy (deallocate)
`delete[]`	Destroy array (deallocate)
`sizeof`	Size of object
`( )`	Type cast

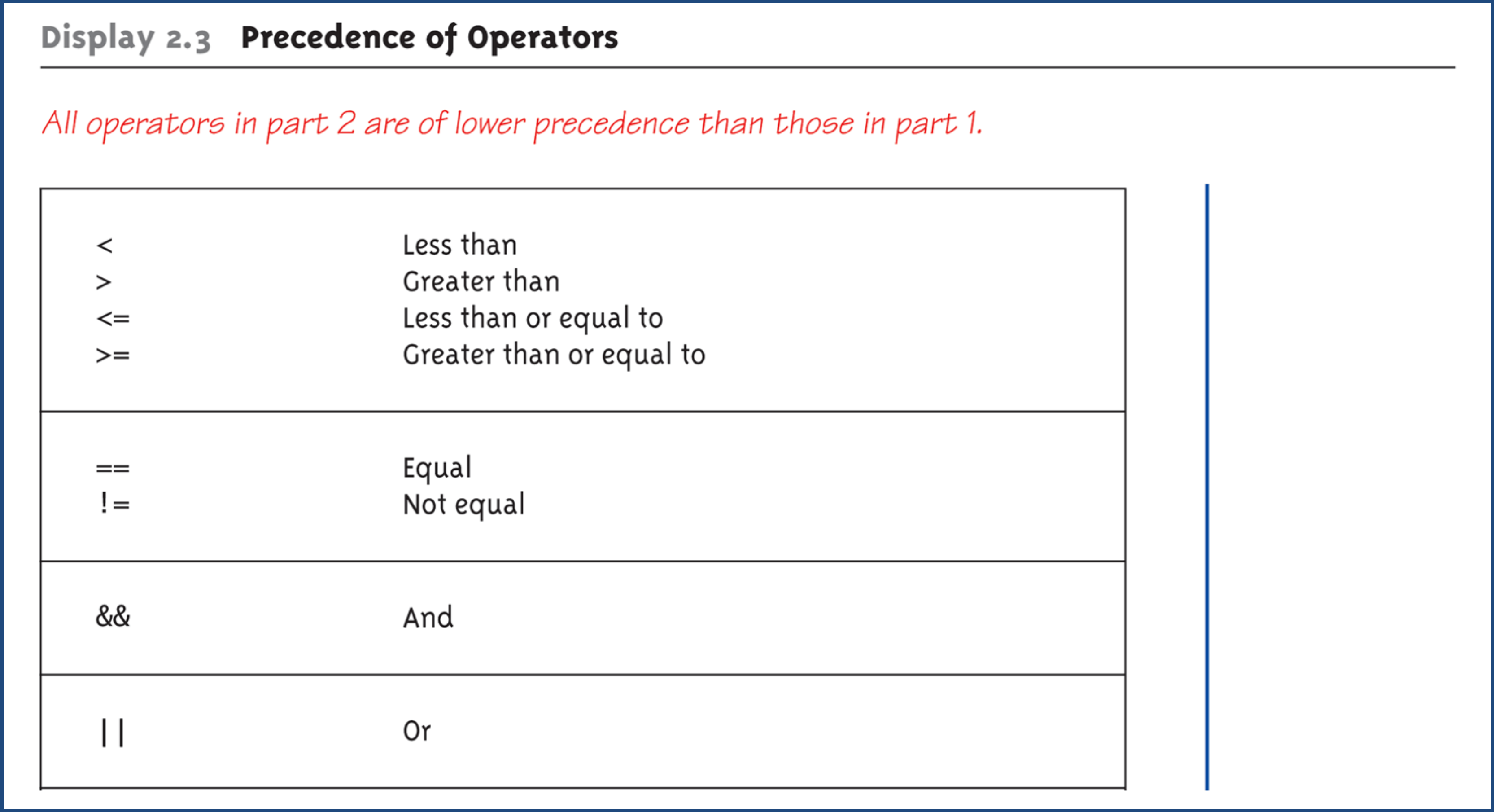
Display 2.3
Precedence of Operators (2 of 4)



*	Multiply
/	Divide
%	Remainder (modulo)
+	Addition
–	Subtraction
<<	Insertion operator (console output)
>>	Extraction operator (console input)

Lower precedence (done later)

Display 2.3
Precedence of Operators (3 of 4)

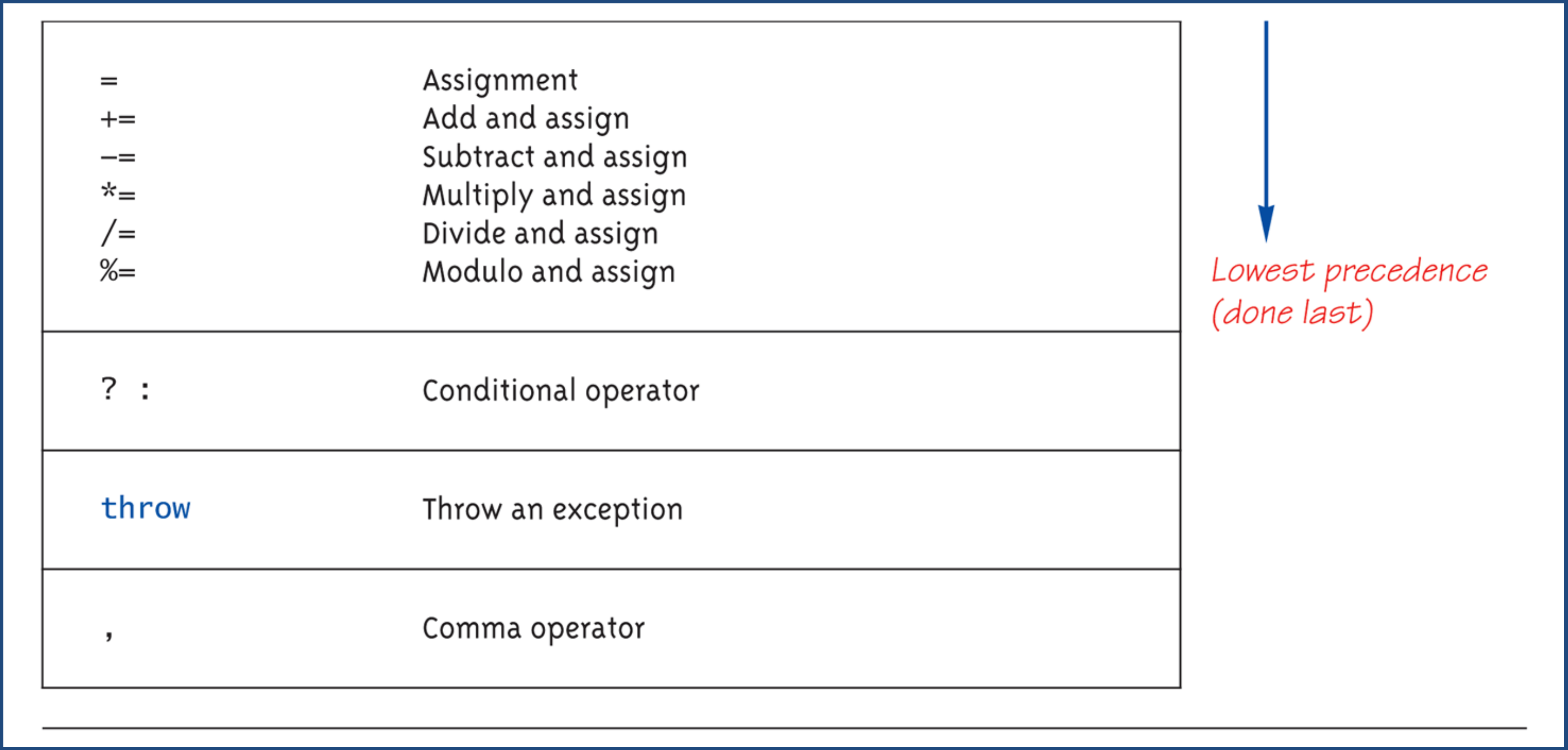


Display 2.3 Precedence of Operators

All operators in part 2 are of lower precedence than those in part 1.

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal
!=	Not equal
&&	And
||	Or

Display 2.3
Precedence of Operators (4 of 4)



`=`
`+=`
`-=`
`*=`
`/=`
`%=`
	Assignment
	Add and assign
	Subtract and assign
	Multiply and assign
	Divide and assign
	Modulo and assign

`? :`	Conditional operator
`throw`	Throw an exception
`,`	Comma operator

Lowest precedence
(done last)

Precedence Examples

- Arithmetic before logical
  - x + 1 > 2 || x + 1 < -3 means:
    - (x + 1) > 2 || (x + 1) < -3
- Short-circuit evaluation
  - (x >= 0) && (y > 1)
  - Be careful with increment operators!
    - (x > 1) && (y++)
- Integers as boolean values
  - All non-zero values → true
  - Zero value → false

Branching Mechanisms

- if-else statements
  - Choice of two alternate statements based on condition expression
  - Example:

```
if (hrs > 40)
  grossPay = rate*40 + 1.5*rate*(hrs-40);
else
  grossPay = rate*hrs;
```

if-else Statement Syntax

- Formal syntax:
```
if (<boolean_expression>)
    <yes_statement>
else
    <no_statement>
```
- Note each alternative is only ONE statement!
- To have multiple statements execute in either branch → use compound statement

Compound/Block Statement

- Only "get" one statement per branch
- Must use compound statement { } for multiples
  - Also called a "block" stmt
- Each block should have block statement
  - Even if just one statement
  - Enhances readability

Compound Statement in Action

- Note indenting in this example:

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

Common Pitfalls

- Operator "=" vs. operator "=="
- One means "assignment" (=)
- One means "equality" (==)
  - VERY different in C++!
  - Example:

```
if (x = 12)  ←Note operator used!
    Do_Something
else
    Do_Something_Else
```

The Optional else

- else clause is optional
  - If, in the false branch (else), you want "nothing" to happen, leave it out
  - Example:
    ```
    if (sales >= minimum)
        salary = salary + bonus;
    cout << "Salary = %" << salary;
    ```
  - Note: nothing to do for false condition, so there is no else clause!
  - Execution continues with cout statement

Nested Statements

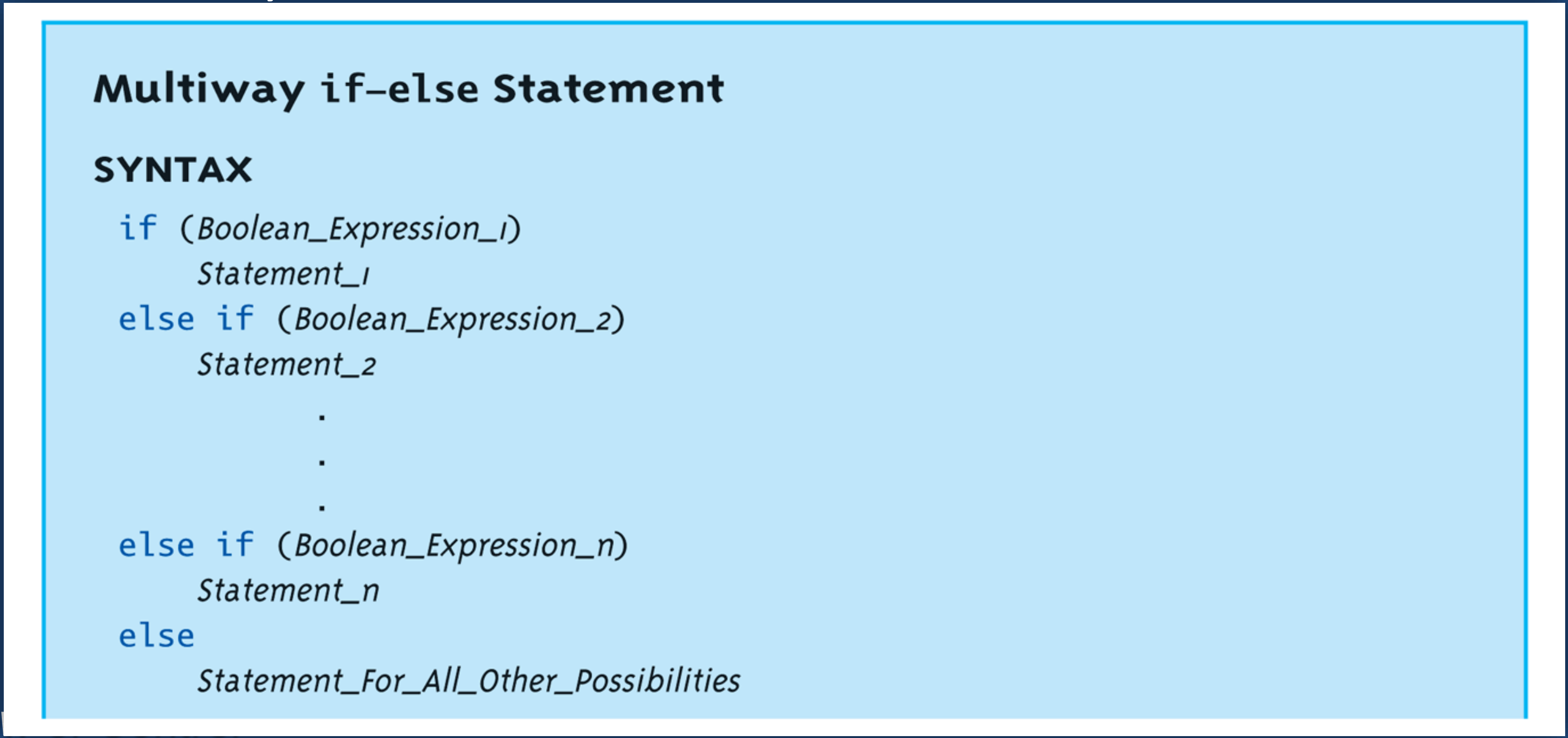
- if-else statements contain smaller statements
  - Compound or simple statements (we've seen)
  - Can also contain any statement at all, including another if-else stmt!
  - Example:

```
if (speed > 55)
    if (speed > 80)
        cout << "You're really speeding!";
    else
        cout << "You're speeding.";
```

    - Note proper indenting!

Multiway if-else

- Not new, just different indenting
- Avoids "excessive" indenting
  - Syntax:

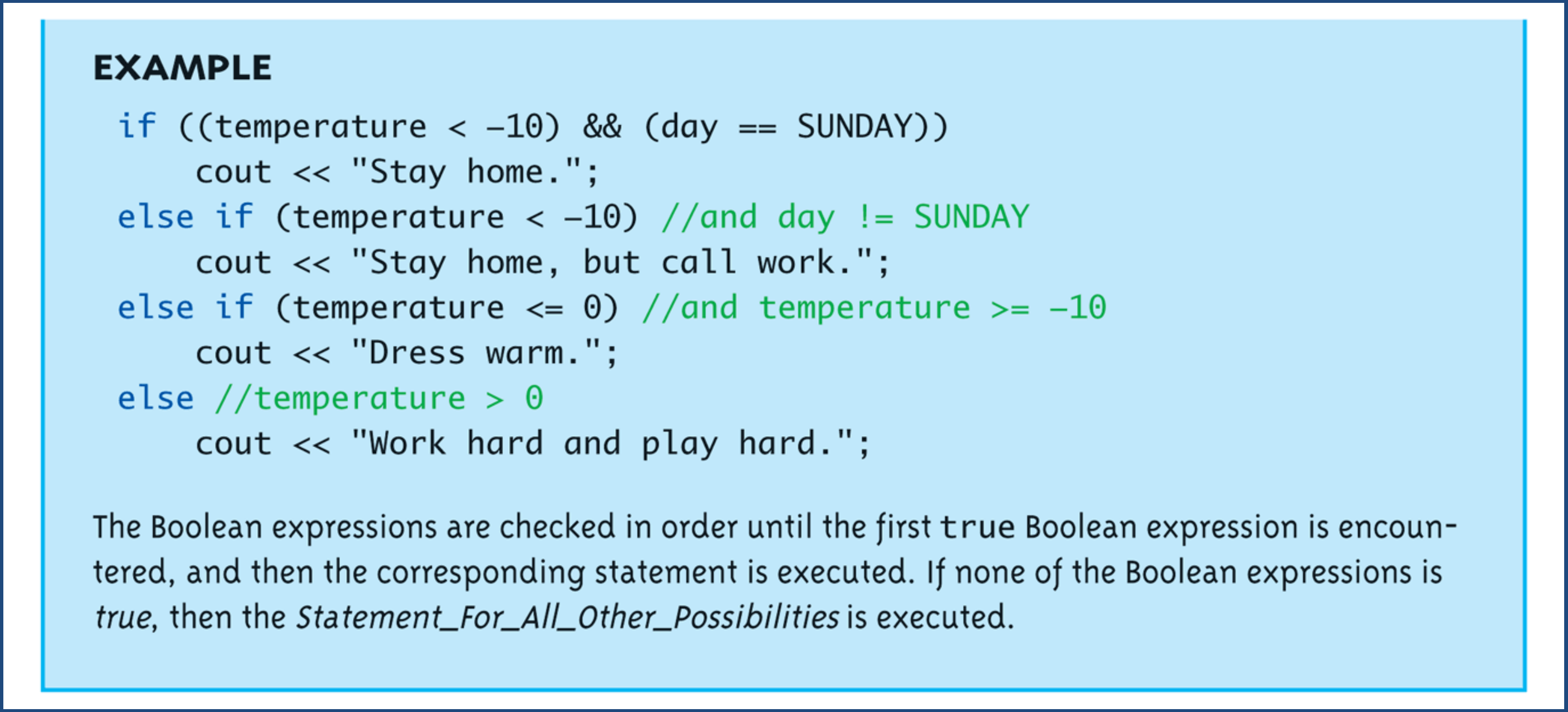


Multiway if-else Statement

SYNTAX

```
if (Boolean_Expression_1)
      Statement_1
else if (Boolean_Expression_2)
      Statement_2
             .
             .
             .
else if (Boolean_Expression_n)
      Statement_n
else
      Statement_For_All_Other_Possibilities
```

Multiway if-else Example



EXAMPLE

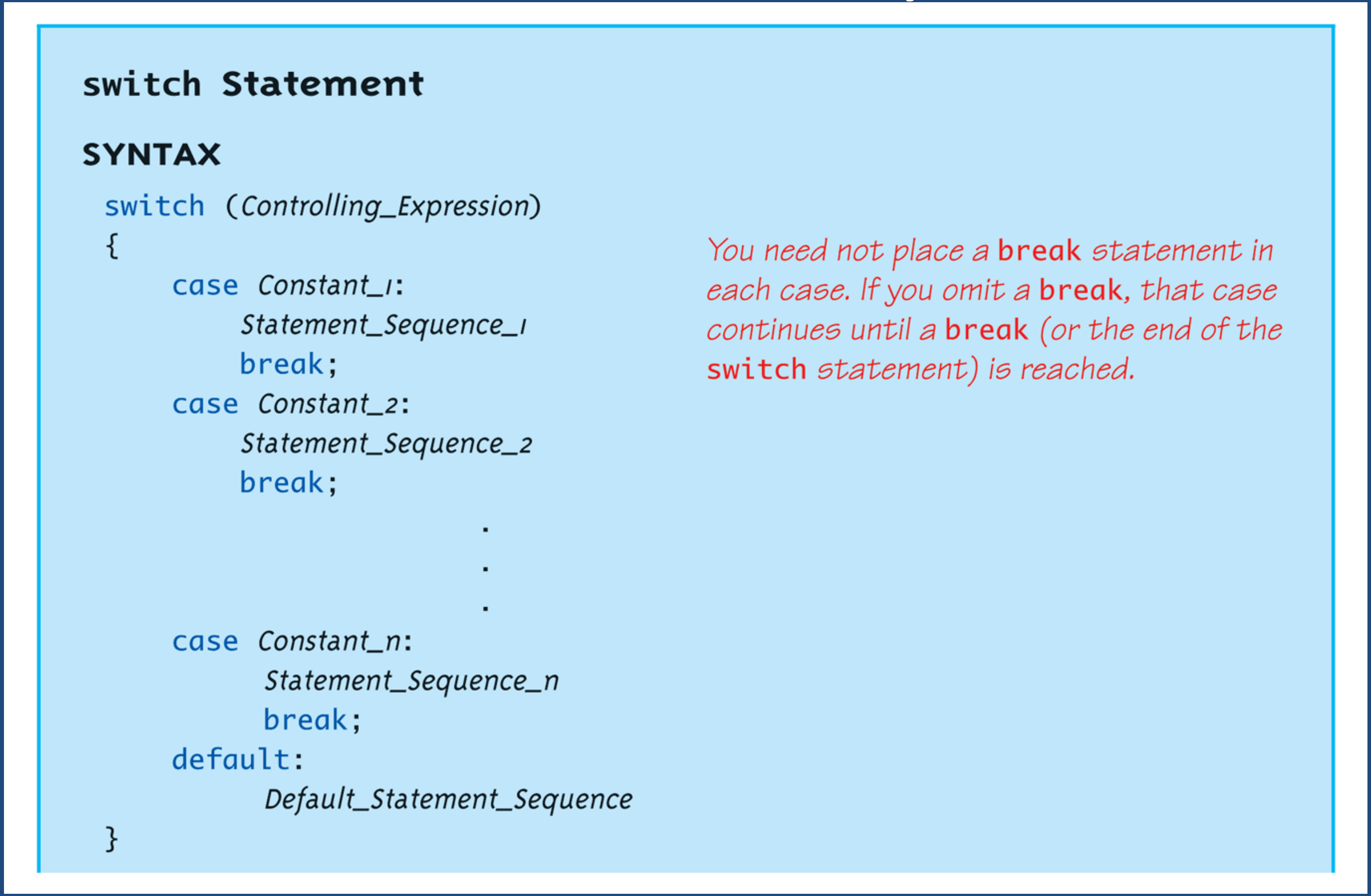
```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10) //and day != SUNDAY
    cout << "Stay home, but call work.";
else if (temperature <= 0) //and temperature >= -10
    cout << "Dress warm.";
else //temperature > 0
    cout << "Work hard and play hard.";
```

The Boolean expressions are checked in order until the first `true` Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is *true*, then the *Statement_For_All_Other_Possibilities* is executed.

The switch Statement

- A statement for controlling multiple branches
- Can do the same thing with if statements but sometimes switch is more convenient
- Uses controlling expression which returns bool data type (true or false)
- Syntax:
  - Next slide

switch Statement Syntax



switch Statement

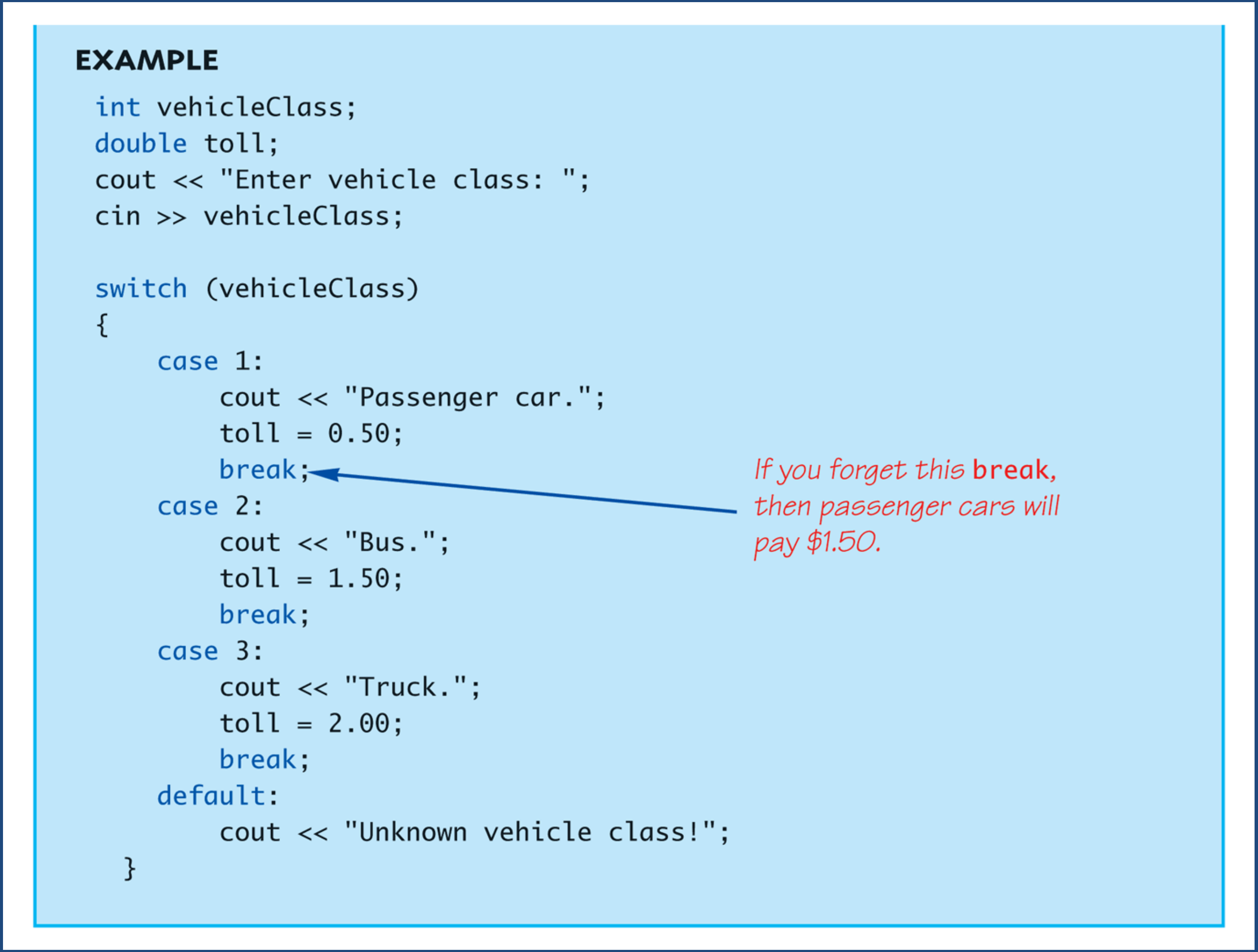
SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
          Statement_Sequence_1
          break;
    case Constant_2:
          Statement_Sequence_2
          break;
                      .
                      .
                      .
    case Constant_n:
           Statement_Sequence_n
           break;
    default:
           Default_Statement_Sequence
}
```

You need not place a **break** *statement in each case. If you omit a* **break**, *that case continues until a* **break** *(or the end of the* **switch** *statement) is reached.*

The controlling expression must be integral! This includes char.

The switch Statement in Action



EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
 }
```

If you forget this **break**, *then passenger cars will pay $1.50.*

The switch: multiple case labels

- Execution "falls thru" until break
  - switch provides a "point of entry"
  - Example:

```
case 'A':
case 'a':
    cout << "Excellent: you got an "A"!\n";
    break;
case 'B':
case 'b':
    cout << "Good: you got a "B"!\n";
    break;
```

  - Note multiple labels provide same "entry"

switch Pitfalls/Tip

- Forgetting the break;
  - No compiler error
  - Execution simply "falls thru" other cases until break;
- Biggest use: MENUs
  - Provides clearer "big-picture" view
  - Shows menu structure effectively
  - Each branch is one menu choice

switch Menu Example

- Switch stmt "perfect" for menus:

```
switch (response)
{
    case 1:
        // Execute menu option 1
        break;
    case 2:
        // Execute menu option 2
        break;
    case 3:
        // Execute menu option 3
        break;
    default:
        cout << "Please enter valid response.";
}
```

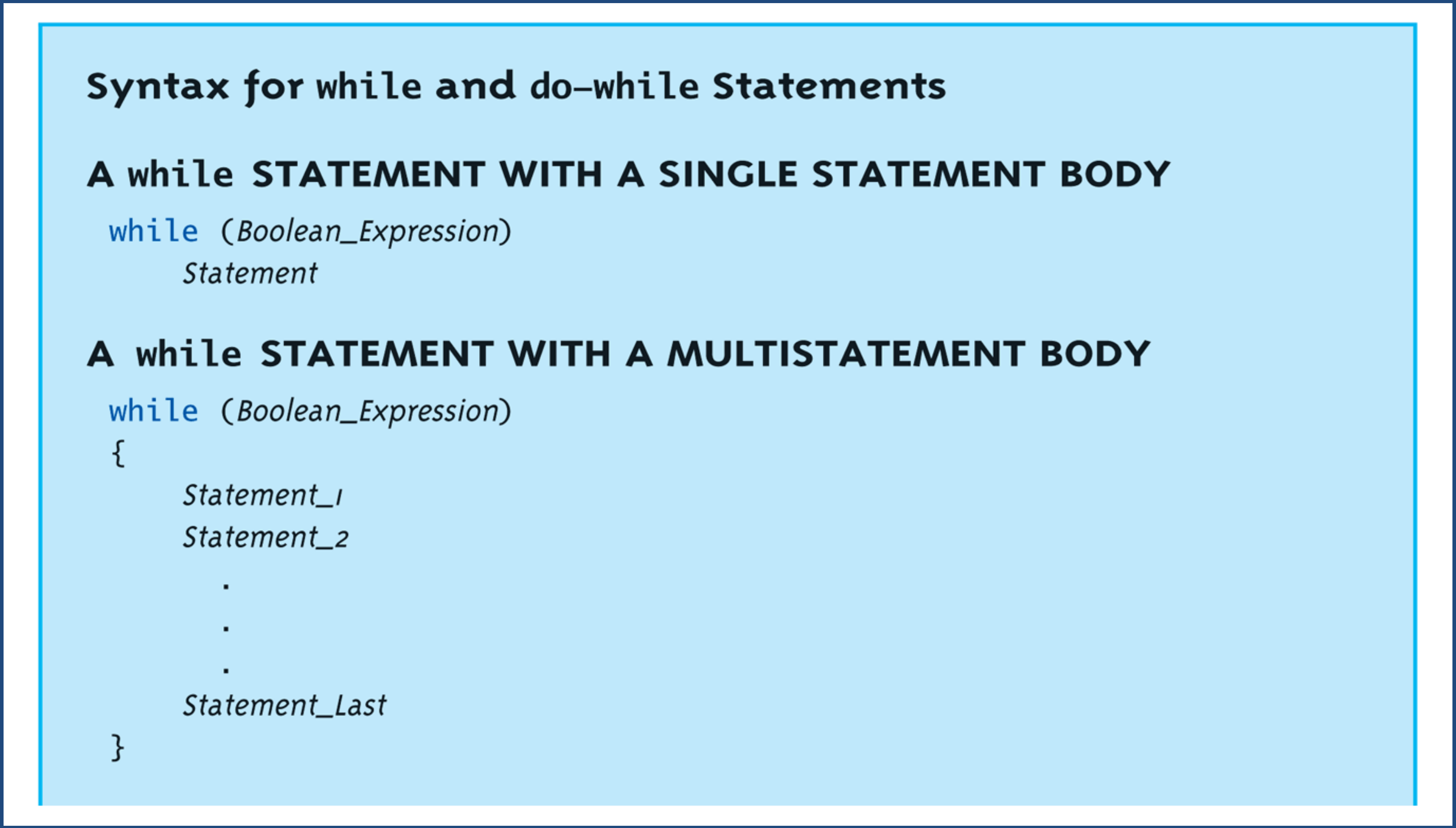
Conditional Operator

- Also called "ternary operator"
  - Allows embedded conditional in expression
  - Essentially "shorthand if-else" operator
  - Example:
    ```
    if (n1 > n2)
        max = n1;
    else
        max = n2;
    ```
  - Can be written:
    ```
    max = (n1 > n2) ? N1 : n2;
    ```
    - "?" and ":" form this "ternary" operator

Loops

- 3 Types of loops in C++
  - while
    - Most flexible
    - No "restrictions"
  - do-while
    - Least flexible
    - Always executes loop body at least once
  - for
    - Natural "counting" loop

while Loops Syntax



Syntax for while and do-while Statements

A while STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
      Statement
```

A while STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
      Statement_1
      Statement_2
         .
         .
         .
      Statement_Last
}
```

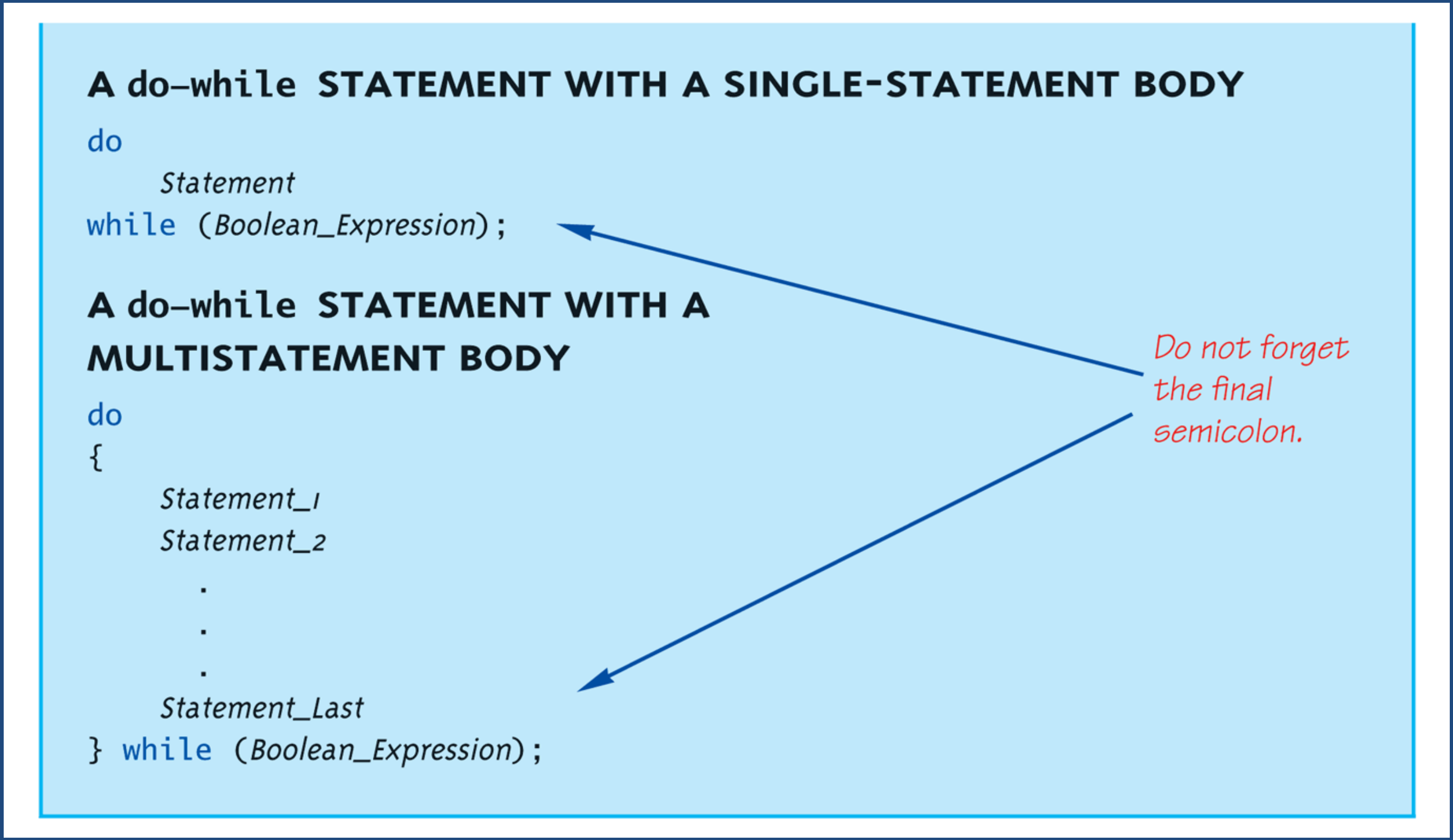
while Loop Example

- Consider:

```
count = 0;                    // Initialization
while (count < 3)                    // Loop Condition
{
      cout << "Hi ";                 // Loop Body
      count++;                       // Update expression
}
```

  - Loop body executes how many times?

do-while Loop Syntax



A do–while STATEMENT WITH A SINGLE-STATEMENT BODY

```
do
    Statement
while (Boolean_Expression);
```

A do–while STATEMENT WITH A MULTISTATEMENT BODY

```
do
{
    Statement_1
    Statement_2
       .
       .
       .
    Statement_Last
} while (Boolean_Expression);
```

Do not forget the final semicolon.

do-while Loop Example

-
  ```
  count = 0;          // Initialization
  do
  {
       cout << "Hi ";          // Loop Body
       count++;                // Update expression
  } while (count < 3);         // Loop Condition
  ```
  - Loop body executes how many times?
  - do-while loops always execute body at least once!

while vs. do-while

- Very similar, but…
  - One important difference
    - Issue is "WHEN" boolean expression is checked
    - while: checks BEFORE body is executed
    - do-while: checked AFTER body is executed
- After this difference, they're essentially identical!
- while is more common, due to it's ultimate "flexibility"

Comma Operator

- Evaluate list of expressions, returning value of the last expression
- Most often used in a for-loop
- Example:
  first = (first = 2, second = first + 1);
  - first gets assigned the value 3
  - second gets assigned the value 3
- No guarantee what order expressions will be evaluated.

for Loop Syntax

```
for (Init_Action; Bool_Exp; Update_Action)
    Body_Statement
```

- Like if-else, Body_Statement can be a block statement
  - Much more typical

for Loop Example

-
  ```
  for (count=0; count<3; count++)
  {
      cout << "Hi ";        // Loop Body
  }
  ```
- How many times does loop body execute?
- Initialization, loop condition and update all "built into" the for-loop structure!
- A natural "counting" loop

Loop Issues

- Loop’s condition expression can be ANY boolean expression
- Examples:

```
while (count<3 && done!=0)
{
    // Do something
}
for (index=0;index<10 && entry!=-99)
{
    // Do something
}
```

Loop Pitfalls: Misplaced ;

- Watch the misplaced ; (semicolon)
  - Example:

```
while (response != 0) ;
{
    cout << "Enter val: ";
    cin >> response;
}
```

  - Notice the ";" after the while condition!
- Result here: INFINITE LOOP!

Loop Pitfalls: Infinite Loops

- Loop condition must evaluate to false at some iteration through loop
  - If not → infinite loop.
  - Example:

    ```
    while (1)
    {
        cout << "Hello ";
    }
    ```
  - A perfectly legal C++ loop → always infinite!
- Infinite loops can be desirable
  - e.g., "Embedded Systems"

The break and continue Statements

- Flow of Control
  - Recall how loops provide "graceful" and clear flow of control in and out
  - In RARE instances, can alter natural flow
- **break**;
  - Forces loop to exit immediately.
- **continue**;
  - Skips rest of loop body
- These statements violate natural flow
  - Only used when absolutely necessary!

Nested Loops

- Recall: ANY valid C++ statements can be inside body of loop
- This includes additional loop statements!
  - Called "nested loops"
- Requires careful indenting:

```
for (outer=0; outer<5; outer++)
    for (inner=7; inner>2; inner--)
        cout << outer << inner;
```

  - Notice no { } since each body is one statement
  - Good style dictates we use { } anyway

Introduction to File Input

- We can use cin to read from a file in a manner very similar to reading from the keyboard
- Only an introduction is given here, more details are in chapter 12
  - Just enough so you can read from text files and process larger amounts of data that would be too much work to type in

Opening a Text File

- Add at the top

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

- You can then declare an input stream just as you would declare any other variable.

```
ifstream inputStream;
```

- Next you must connect the inputStream variable to a text file on the disk.

```
inputStream.open("filename.txt");
```

- The “filename.txt” is the pathname to a text file or a file in the current directory

Reading from a Text File

- Use

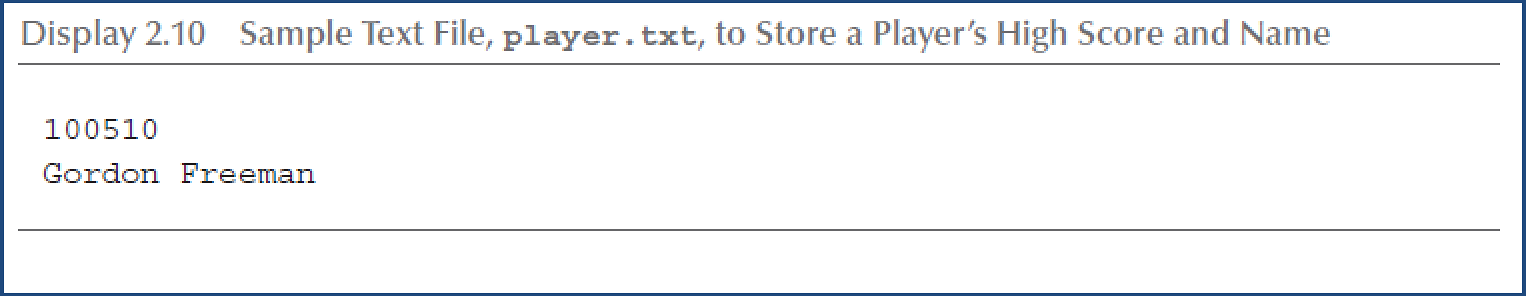
```
inputStream >> var;
```

- The result is the same as using `cin >> var` except the input is coming from the text file and not the keyboard
- When done with the file close it with

```
inputStream.close();
```

File Input Example (1 of 2)

- Consider a text file named *player.txt* with the following text


Display 2.10 Sample Text File, `player.txt`, to Store a Player's High Score and Name

```
100510
Gordon Freeman
```

File Input Example (2 of 2)

Display 2.11 Program to Read the Text File in Display 2.10

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;
int main( )
{
    string firstName, lastName;
    int score;
    fstream inputStream;

    inputStream.open("player.txt");

    inputStream >> score;
    inputStream >> firstName >> lastName;

    cout << "Name: " << firstName << " "
         << lastName << endl;
    cout << "Score: " << score << endl;
    inputStream.close();

    return 0;
}
```

Sample Dialogue

```
Name: Gordon Freeman
Score: 100510
```

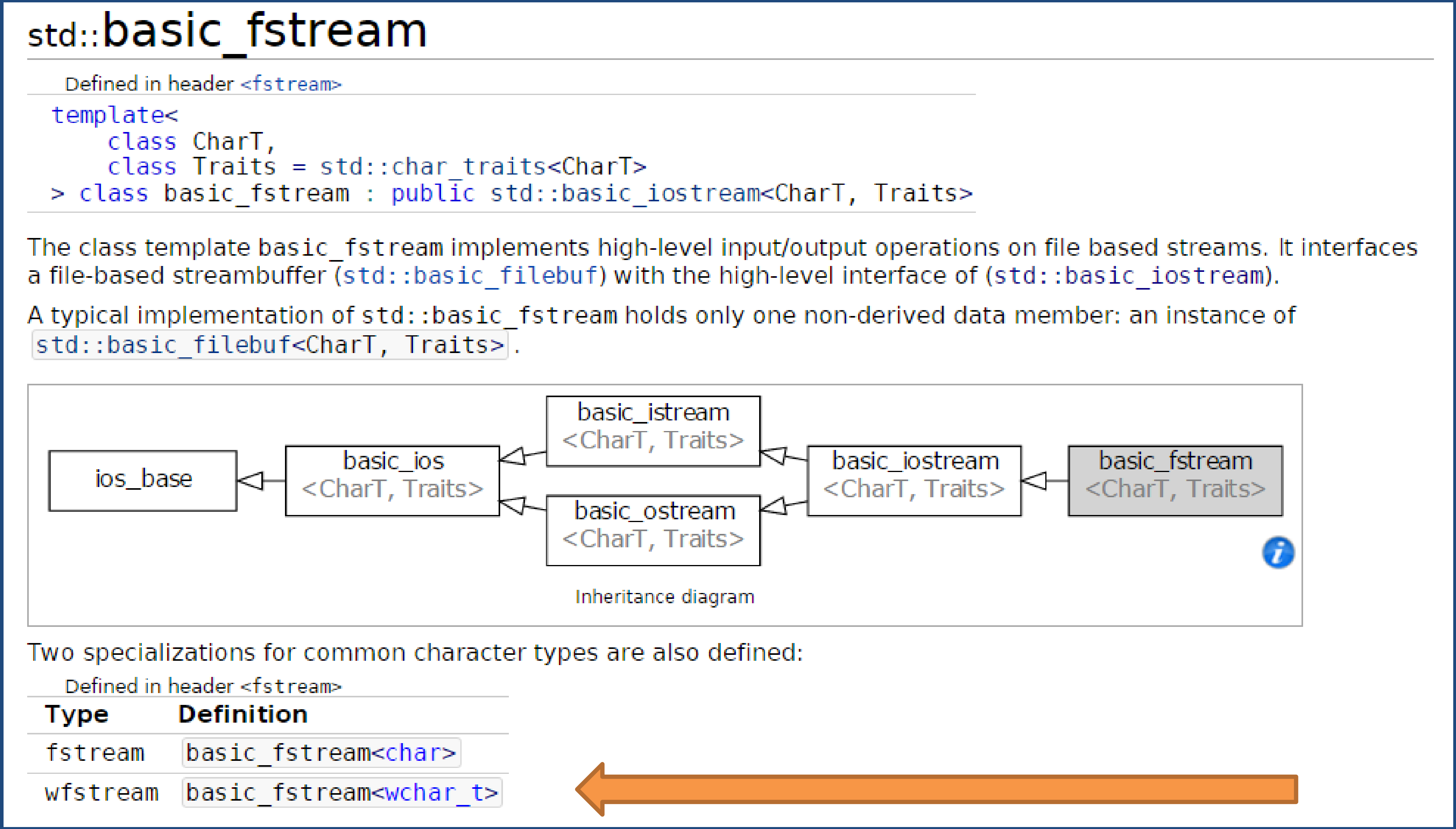
Display 2.10 Sample Text File, `player.txt`, to Store a Player's High Score and Name

```
100510
Gordon Freeman
```

Notes on Display 2.11

- **fstream**
  - https://msdn.microsoft.com/zh-tw/library/6z061fh0.aspx
  - basic_fstream: https://msdn.microsoft.com/zh-tw/library/a33ahe62.aspx
- Member function
  - .close():
  - .open()
  - http://en.cppreference.com/w/cpp/io/basic_fstream

fstream



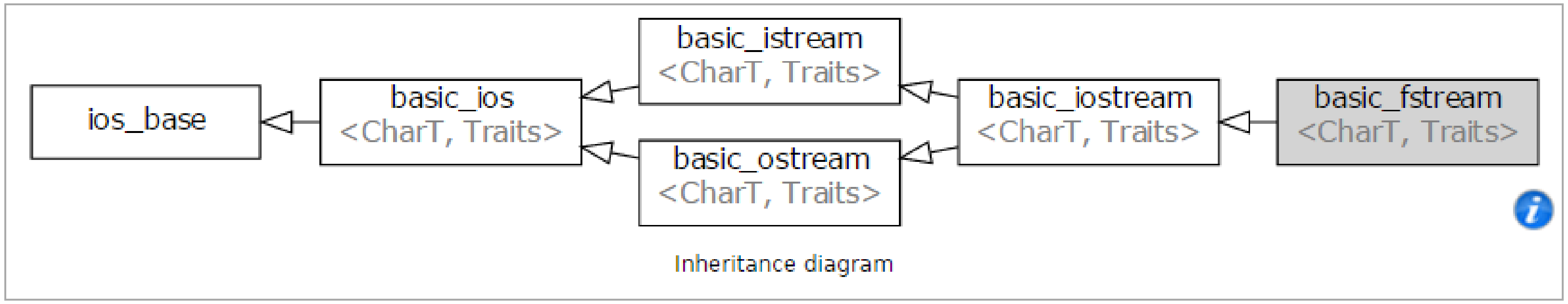
std::basic_fstream

Defined in header <fstream>

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>
> class basic_fstream : public std::basic_iostream<CharT, Traits>
```

The class template `basic_fstream` implements high-level input/output operations on file based streams. It interfaces a file-based streambuffer (`std::basic_filebuf`) with the high-level interface of (`std::basic_iostream`).

A typical implementation of `std::basic_fstream` holds only one non-derived data member: an instance of `std::basic_filebuf<CharT, Traits>`.

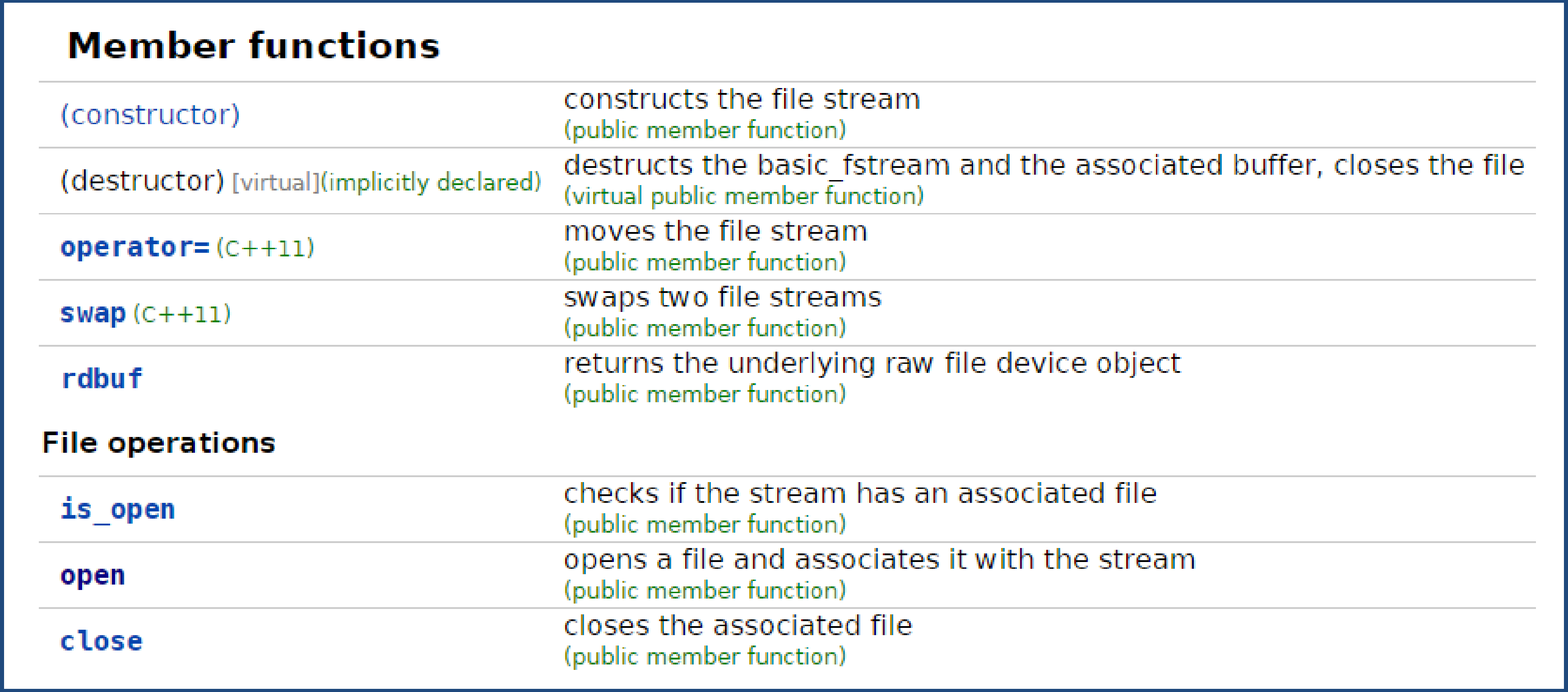

ios_base
basic_ios <CharT, Traits>
basic_istream <CharT, Traits>
basic_ostream <CharT, Traits>
basic_iostream <CharT, Traits>
basic_fstream <CharT, Traits>

Inheritance diagram

Two specializations for common character types are also defined:

Defined in header <fstream>

Type	Definition
`fstream`	`basic_fstream<char>`
`wfstream`	`basic_fstream<wchar_t>`

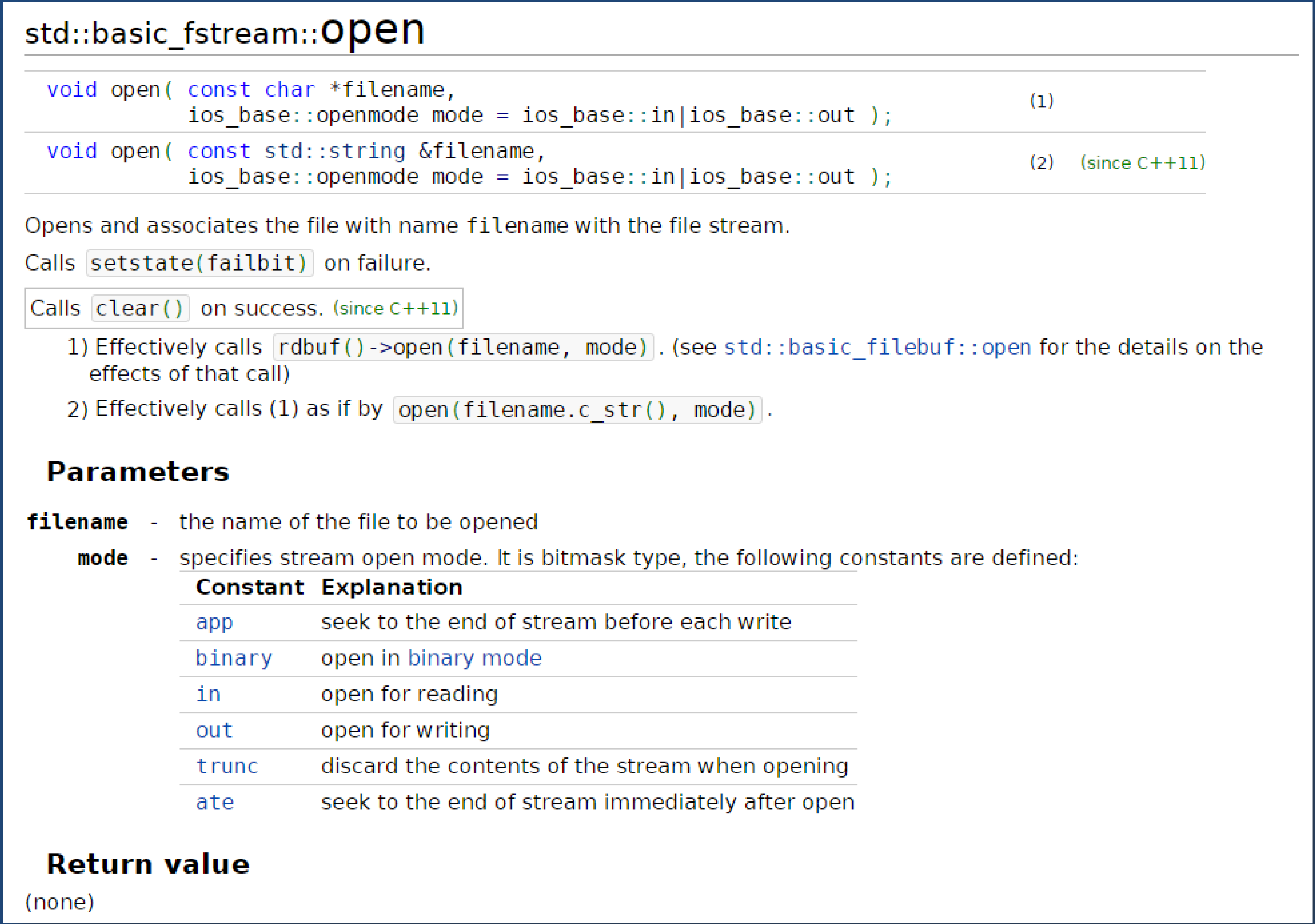


Member functions

(constructor)	constructs the file stream (public member function)
(destructor) [virtual](implicitly declared)	destructs the basic_fstream and the associated buffer, closes the file (virtual public member function)
`operator=` (C++11)	moves the file stream (public member function)
`swap` (C++11)	swaps two file streams (public member function)
`rdbuf`	returns the underlying raw file device object (public member function)

File operations

`is_open`	checks if the stream has an associated file (public member function)
`open`	opens a file and associates it with the stream (public member function)
`close`	closes the associated file (public member function)



std::basic_fstream::open

`void open( const char *filename, ios_base::openmode mode = ios_base::in|ios_base::out );`	(1)	
`void open( const std::string &filename, ios_base::openmode mode = ios_base::in|ios_base::out );`	(2)	(since C++11)

Opens and associates the file with name `filename` with the file stream.

Calls `setstate(failbit)` on failure.

Calls `clear()` on success. (since C++11)

1) Effectively calls `rdbuf()->open(filename, mode)`. (see `std::basic_filebuf::open` for the details on the effects of that call)

2) Effectively calls (1) as if by `open(filename.c_str(), mode)`.

Parameters

filename - the name of the file to be opened

mode - specifies stream open mode. It is bitmask type, the following constants are defined:

Constant	Explanation
`app`	seek to the end of stream before each write
`binary`	open in binary mode
`in`	open for reading
`out`	open for writing
`trunc`	discard the contents of the stream when opening
`ate`	seek to the end of stream immediately after open

Return value

(none)

fstream Example

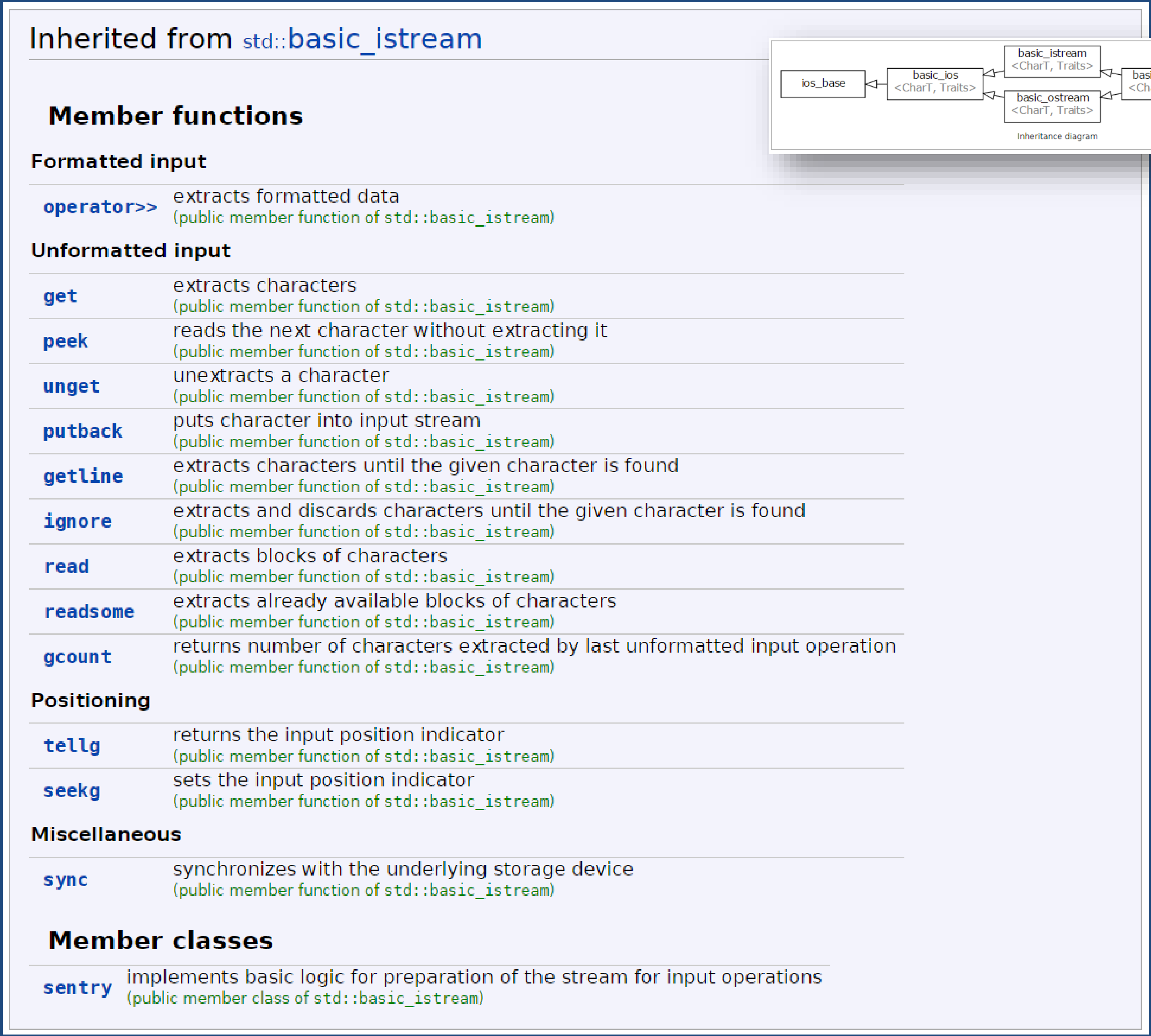
```
#include <string>
#include <fstream>
#include <iostream>

int main()
{
    std::string filename = "example.123";

    std::fstream fs;

    fs.open(filename);

    if(!fs.is_open())
    {
        fs.clear();
        fs.open(filename, std::ios::out); //Create file.
        fs.close();
        fs.open(filename);
    }
    std::cout << std::boolalpha;
    std::cout << "fs.is_open() = " << fs.is_open() << '\n';
    std::cout << "fs.good() = " << fs.good() << '\n';
}
```

Any Questions?

fstream Example

```
#include <string>
#include <fstream>
#include <iostream>

int main()
{
    std::string filename = "example.123";

    std::fstream fs;

    fs.open(filename);

    if(!fs.is_open())
    {
      fs.clear();
      fs.open(filename, std::ios::out); //Create file.
      fs.close();
      fs.open(filename);
    }

    std::cout << std::boolalpha;

    std::cout << "fs.is_open() = " << fs.is_open() << '\n';
    std::cout << "fs.good() = " << fs.good() << '\n';
}
```

When the boolalpha format flag is set, bool values are inserted/extracted by their textual representation: either true or false, instead of integral values.



Inherited from std::basic_istream

Member functions

Formatted input

operator>>	extracts formatted data (public member function of std::basic_istream)

Unformatted input

get	extracts characters (public member function of std::basic_istream)
peek	reads the next character without extracting it (public member function of std::basic_istream)
unget	unextracts a character (public member function of std::basic_istream)
putback	puts character into input stream (public member function of std::basic_istream)
getline	extracts characters until the given character is found (public member function of std::basic_istream)
ignore	extracts and discards characters until the given character is found (public member function of std::basic_istream)
read	extracts blocks of characters (public member function of std::basic_istream)
readsome	extracts already available blocks of characters (public member function of std::basic_istream)
gcount	returns number of characters extracted by last unformatted input operation (public member function of std::basic_istream)

Positioning

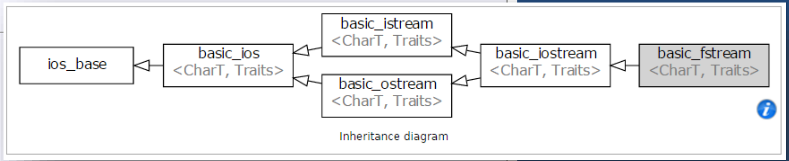
tellg	returns the input position indicator (public member function of std::basic_istream)
seekg	sets the input position indicator (public member function of std::basic_istream)

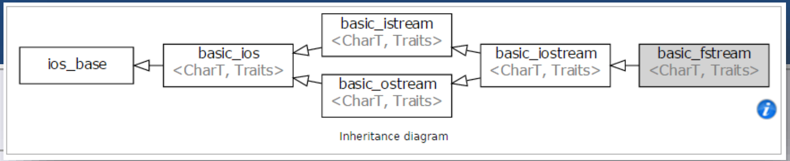
Miscellaneous

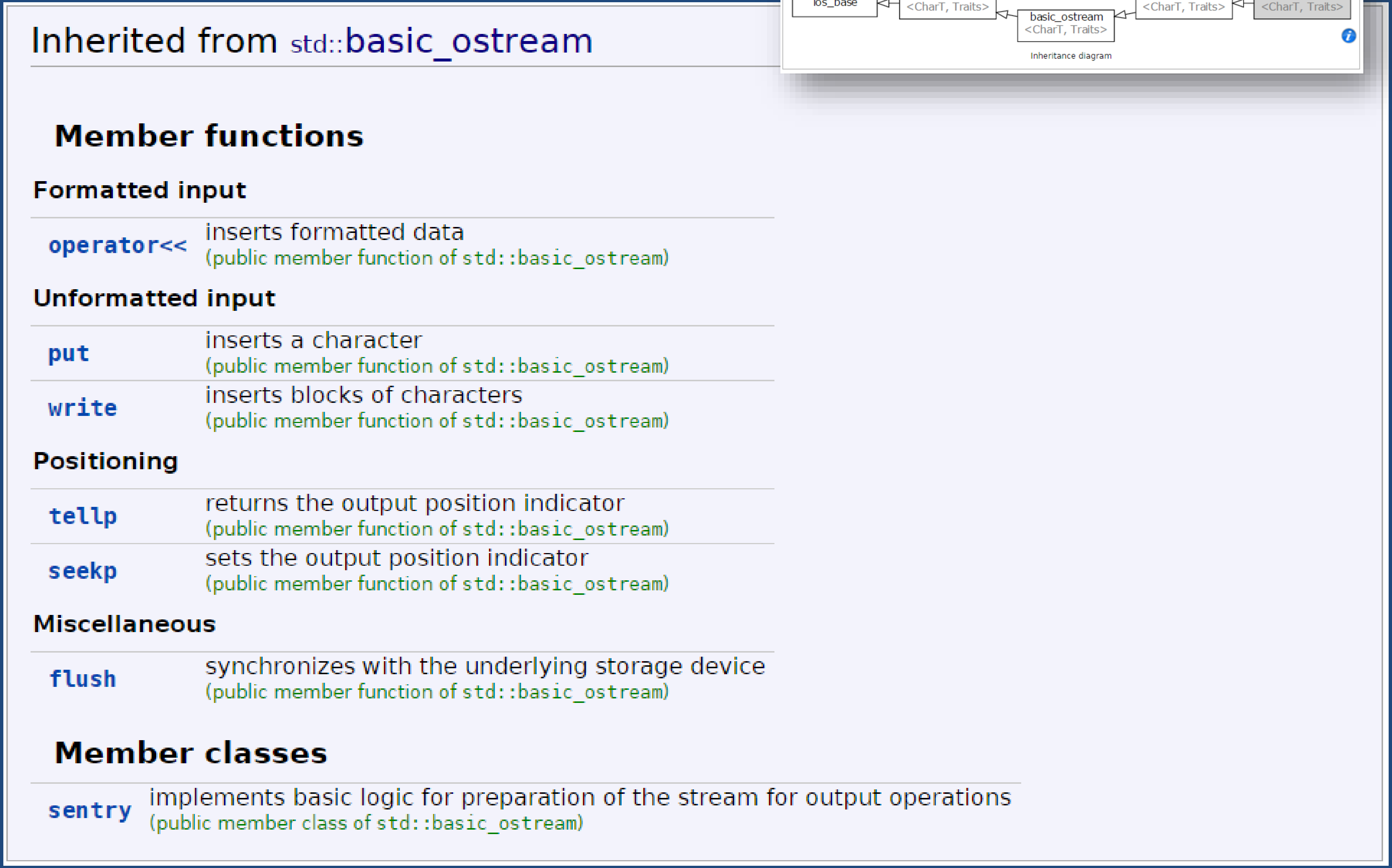
sync	synchronizes with the underlying storage device (public member function of std::basic_istream)

Member classes

sentry	implements basic logic for preparation of the stream for input operations (public member class of std::basic_istream)


ios_base
basic_ios <CharT, Traits>
basic_istream <CharT, Traits>
basic_ostream <CharT, Traits>
basic_iostream <CharT, Traits>
basic_fstream <CharT, Traits>
Inheritance diagram


ios_base
basic_ios <CharT, Traits>
basic_istream <CharT, Traits>
basic_ostream <CharT, Traits>
basic_iostream <CharT, Traits>
basic_fstream <CharT, Traits>
Inheritance diagram



Inherited from std::basic_ostream

Member functions

Formatted input

`operator<<`	inserts formatted data (public member function of `std::basic_ostream`)

Unformatted input

`put`	inserts a character (public member function of `std::basic_ostream`)
`write`	inserts blocks of characters (public member function of `std::basic_ostream`)

Positioning

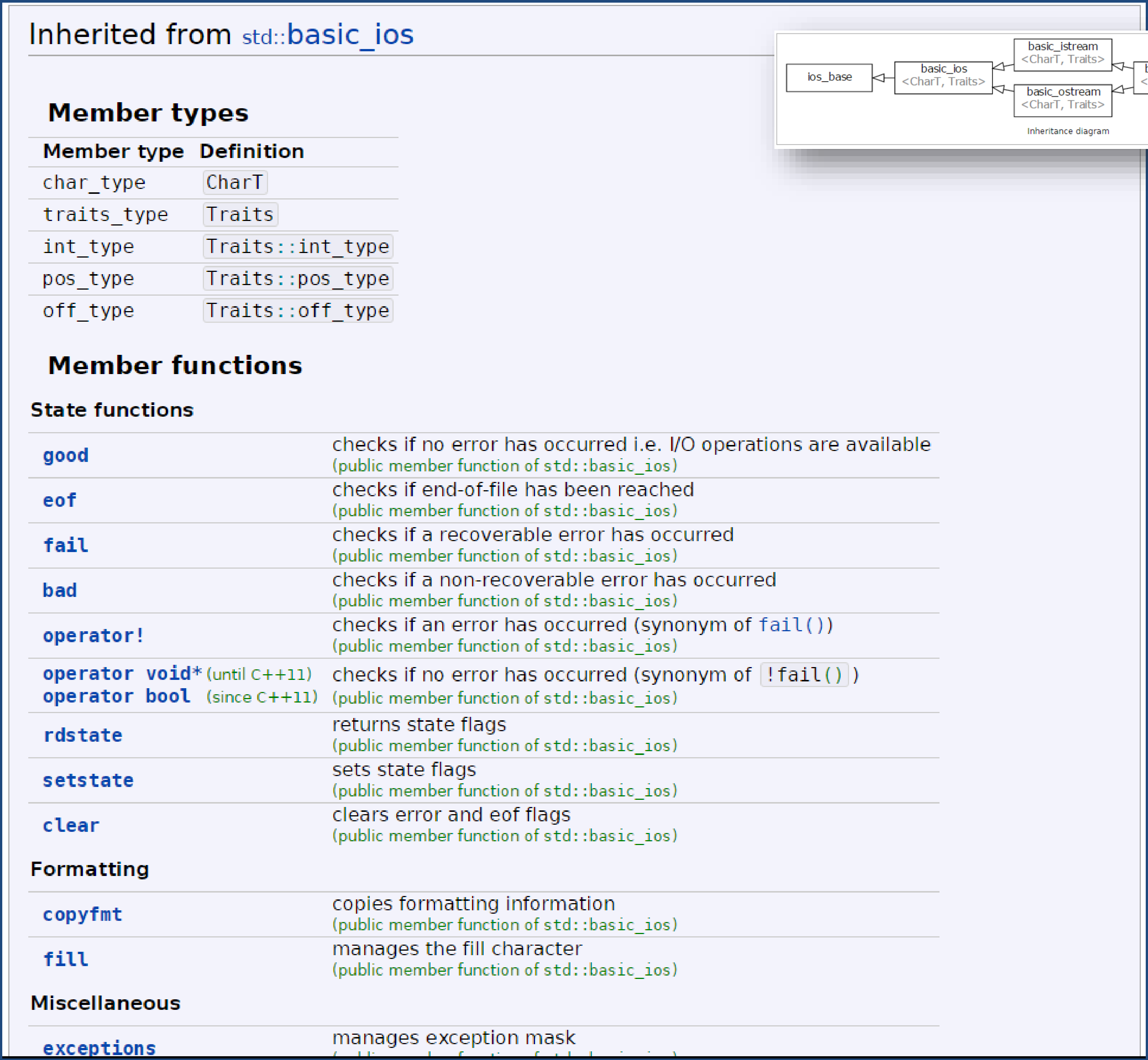
`tellp`	returns the output position indicator (public member function of `std::basic_ostream`)
`seekp`	sets the output position indicator (public member function of `std::basic_ostream`)

Miscellaneous

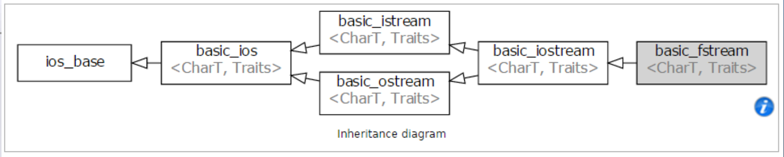
`flush`	synchronizes with the underlying storage device (public member function of `std::basic_ostream`)

Member classes

`sentry`	implements basic logic for preparation of the stream for output operations (public member class of `std::basic_ostream`)



Inherited from std::basic_ios


ios_base
basic_ios <CharT, Traits>
basic_istream <CharT, Traits>
basic_ostream <CharT, Traits>
basic_iostream <CharT, Traits>
basic_fstream <CharT, Traits>
Inheritance diagram

Member types

Member type	Definition
`char_type`	`CharT`
`traits_type`	`Traits`
`int_type`	`Traits::int_type`
`pos_type`	`Traits::pos_type`
`off_type`	`Traits::off_type`

Member functions

State functions

good	checks if no error has occurred i.e. I/O operations are available (public member function of `std::basic_ios`)
eof	checks if end-of-file has been reached (public member function of `std::basic_ios`)
fail	checks if a recoverable error has occurred (public member function of `std::basic_ios`)
bad	checks if a non-recoverable error has occurred (public member function of `std::basic_ios`)
operator!	checks if an error has occurred (synonym of `fail()`) (public member function of `std::basic_ios`)
operator void* (until C++11) **operator bool** (since C++11)	checks if no error has occurred (synonym of `!fail()`) (public member function of `std::basic_ios`)
rdstate	returns state flags (public member function of `std::basic_ios`)
setstate	sets state flags (public member function of `std::basic_ios`)
clear	clears error and eof flags (public member function of `std::basic_ios`)

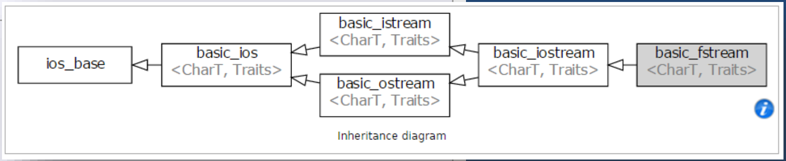
Formatting

copyfmt	copies formatting information (public member function of `std::basic_ios`)
fill	manages the fill character (public member function of `std::basic_ios`)

Miscellaneous

exceptions	manages exception mask

Inherited from std::ios_base


ios_base
basic_ios <CharT, Traits>
basic_istream <CharT, Traits>
basic_ostream <CharT, Traits>
basic_iostream <CharT, Traits>
basic_fstream <CharT, Traits>
Inheritance diagram

Member functions

Formatting

flags	manages format flags (public member function of std::ios_base)
setf	sets specific format flag (public member function of std::ios_base)
unsetf	clears specific format flag (public member function of std::ios_base)
precision	manages decimal precision of floating point operations (public member function of std::ios_base)
width	manages field width (public member function of std::ios_base)

Locales

imbue	sets locale (public member function of std::ios_base)
getloc	returns current locale (public member function of std::ios_base)

Internal extensible array

xalloc [static]	returns a program-wide unique integer that is safe to use as index to pword() and iword() (public static member function of std::ios_base)
iword	resizes the private storage if necessary and access to the `long` element at the given index (public member function of std::ios_base)
pword	resizes the private storage if necessary and access to the `void*` element at the given index (public member function of std::ios_base)

Miscellaneous

register_callback	registers event callback function (public member function of std::ios_base)
sync_with_stdio [static]	sets whether C++ and C IO libraries are interoperable (public static member function of std::ios_base)

Member classes

failure	stream exception (public member class of std::ios_base)
Init	initializes standard stream objects (public member class of std::ios_base)

Summary 1

- Boolean expressions
  - Similar to arithmetic → results in true or false
- C++ branching statements
  - if-else, switch
  - switch statement great for menus
- C++ loop statements
  - while
  - do-while
  - for

Summary 2

- do-while loops
  - Always execute their loop body at least once
- for-loop
  - A natural "counting" loop
- Loops can be exited early
  - break statement
  - continue statement
  - Usage restricted for style purposes
- Reading from a text file is similar to reading from cin