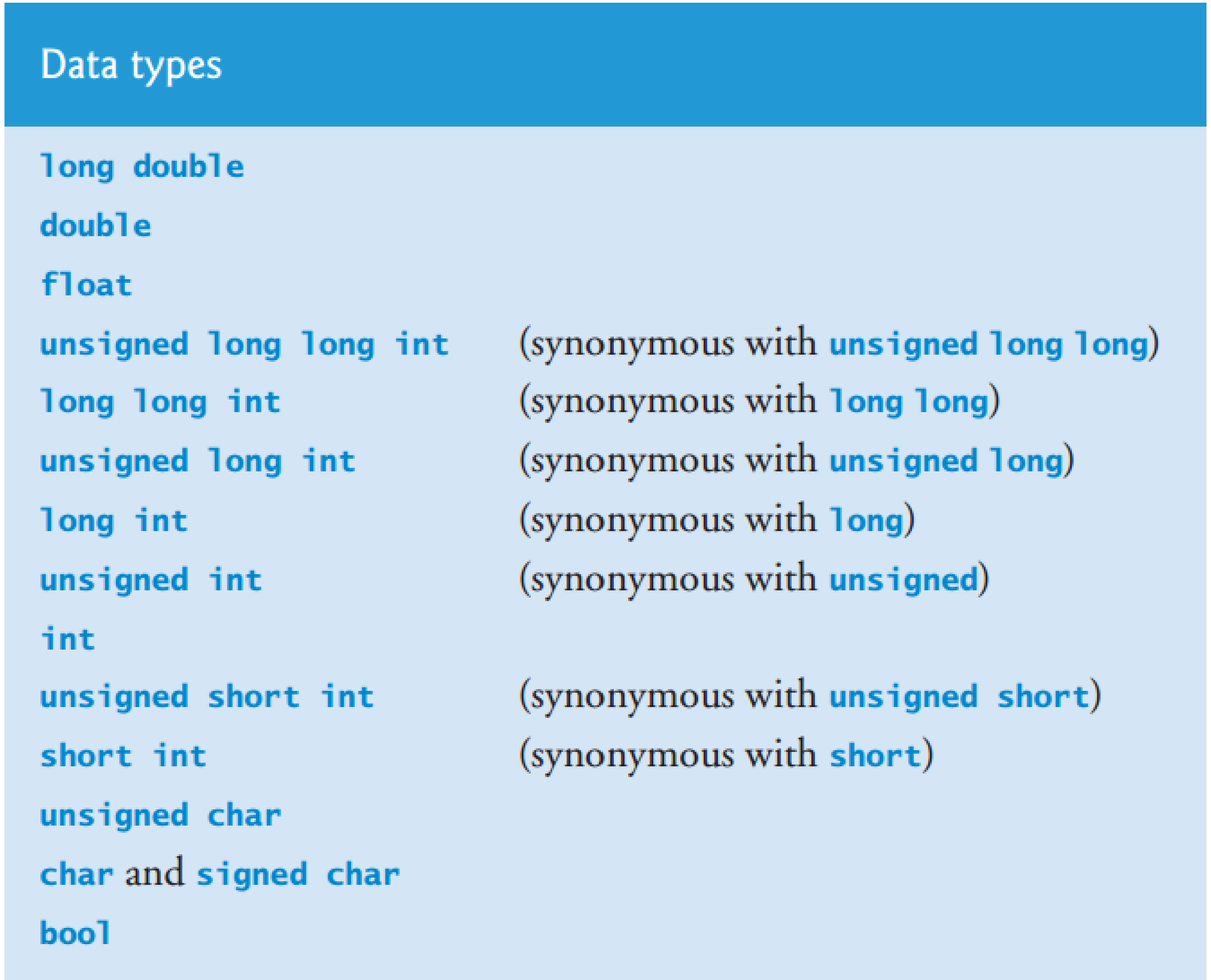
Fundamental Of C++ Programming

Fall 2016-17

Amin Paydar

Paul Deitel, Harvey Deitel-C++ How to Program-Pearson (2014)

Data Types



Data types	
long double	
double	
float	
unsigned long long int	(synonymous with unsigned long long)
long long int	(synonymous with long long)
unsigned long int	(synonymous with unsigned long)
long int	(synonymous with long)
unsigned int	(synonymous with unsigned)
int	
unsigned short int	(synonymous with unsigned short)
short int	(synonymous with short)
unsigned char	
char and signed char	
bool	

Character

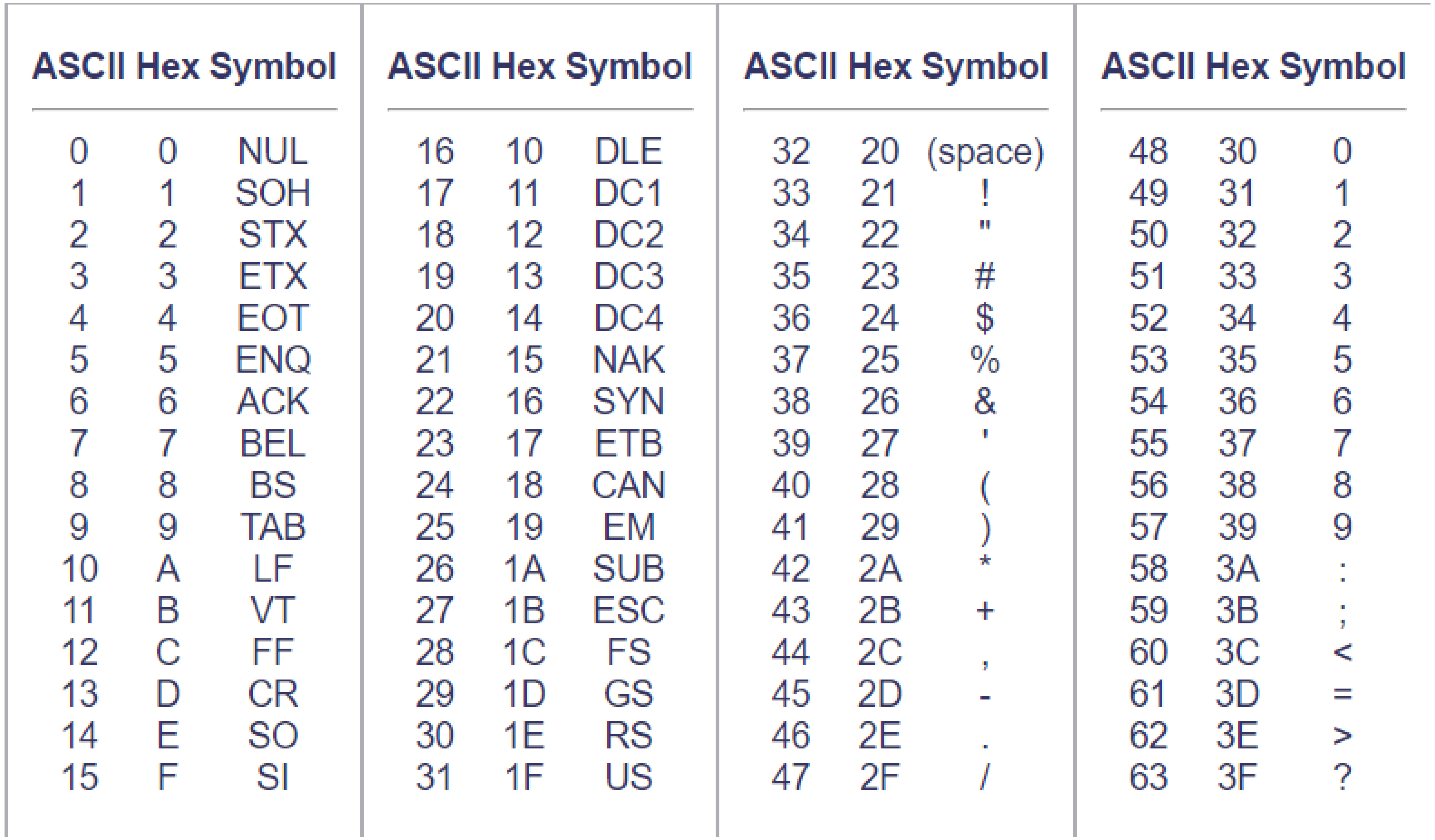
- Declaration
- Convertable to integer

- Ascii codes
  - American Standard characters
  - Each char has a code

```
char c = '$';

int a = (int) c;
c = (char) a;
```

Character



ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

Boolean

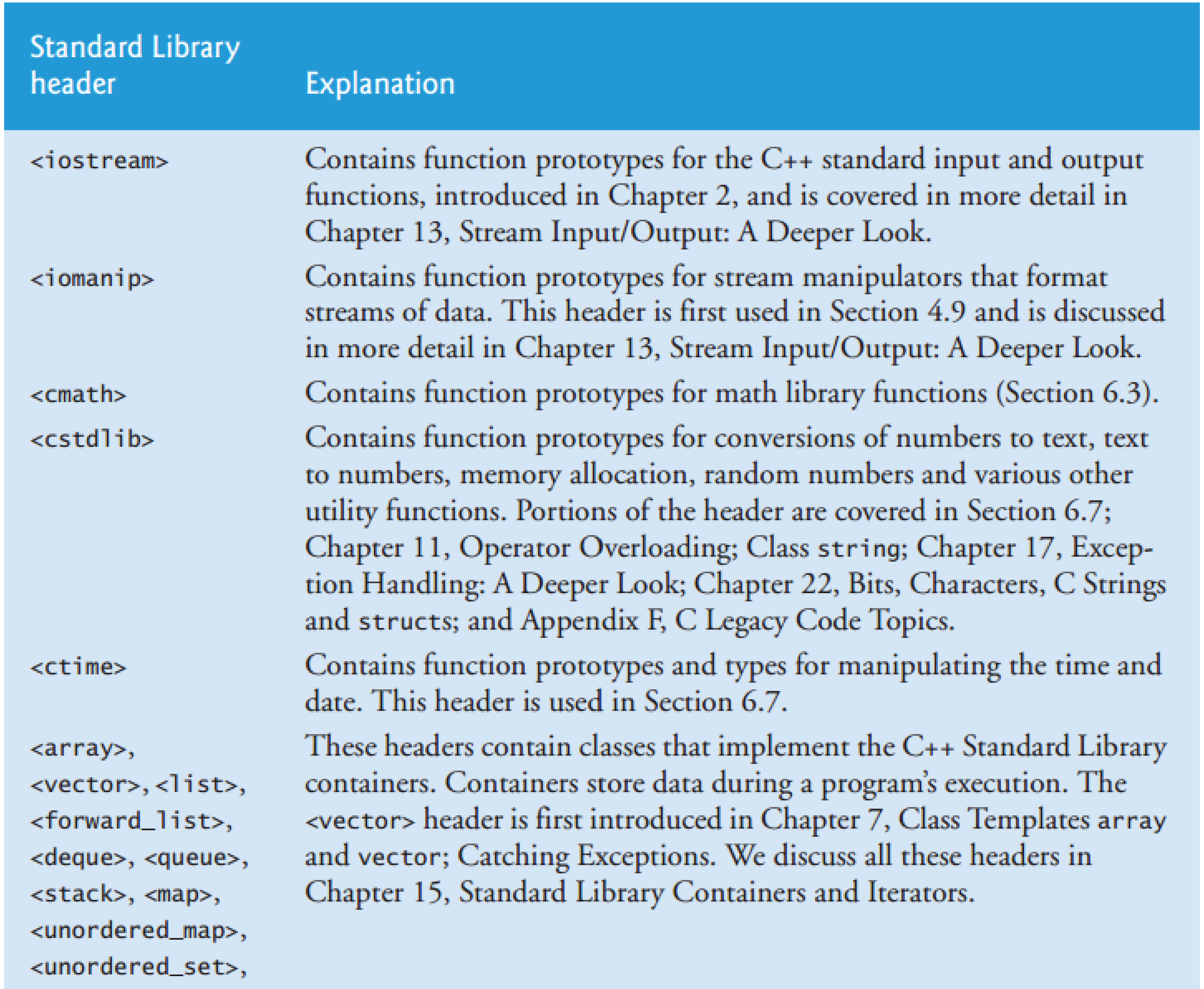
- Has value of true or false
- Convertable to int
  - True => 1
  - False => 0

- Example

```
bool b = 5 > 7;
bool my_bool = 1000;
bool my_bool = "asdf";
```

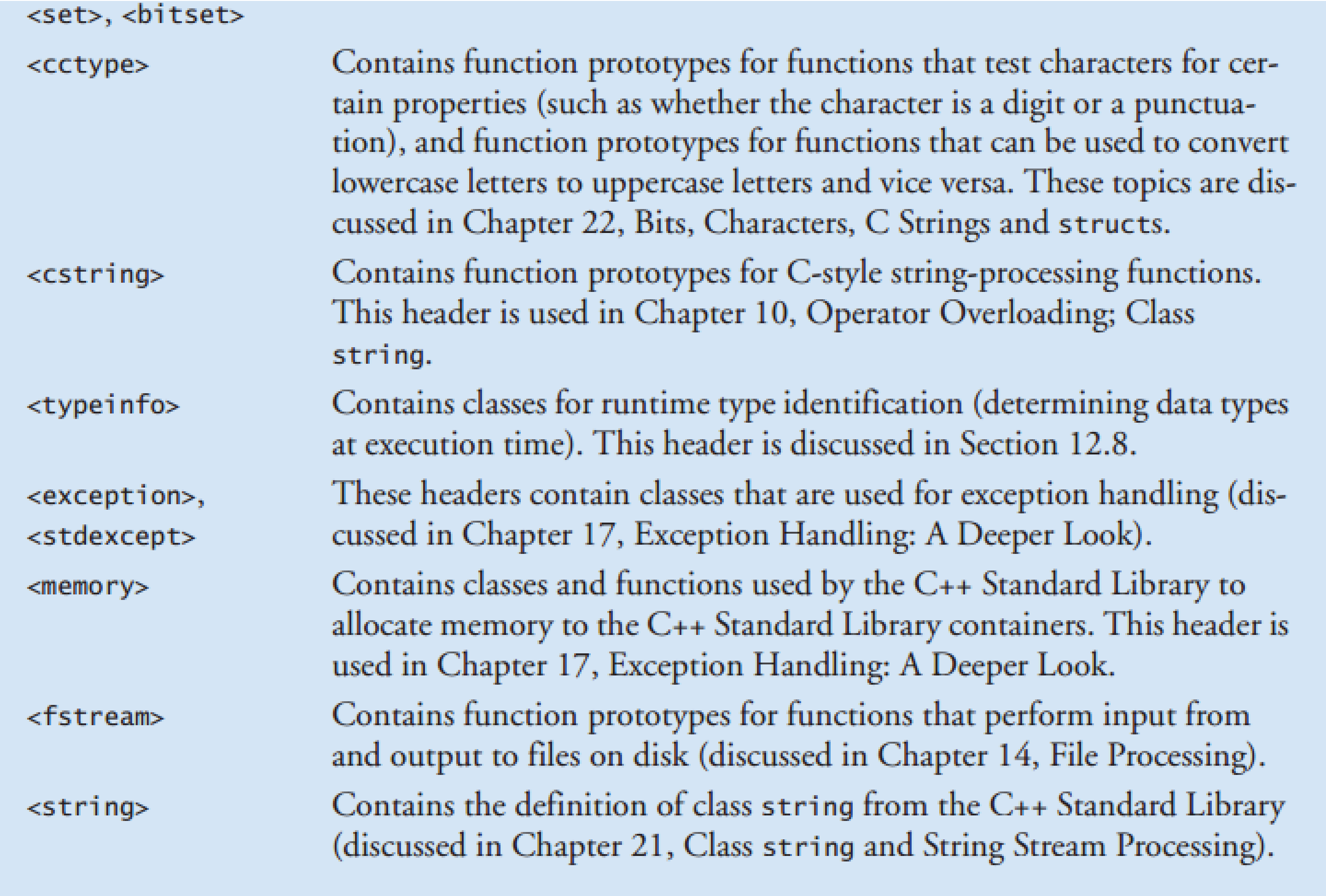
Boolean Algebra

Standard Libraries



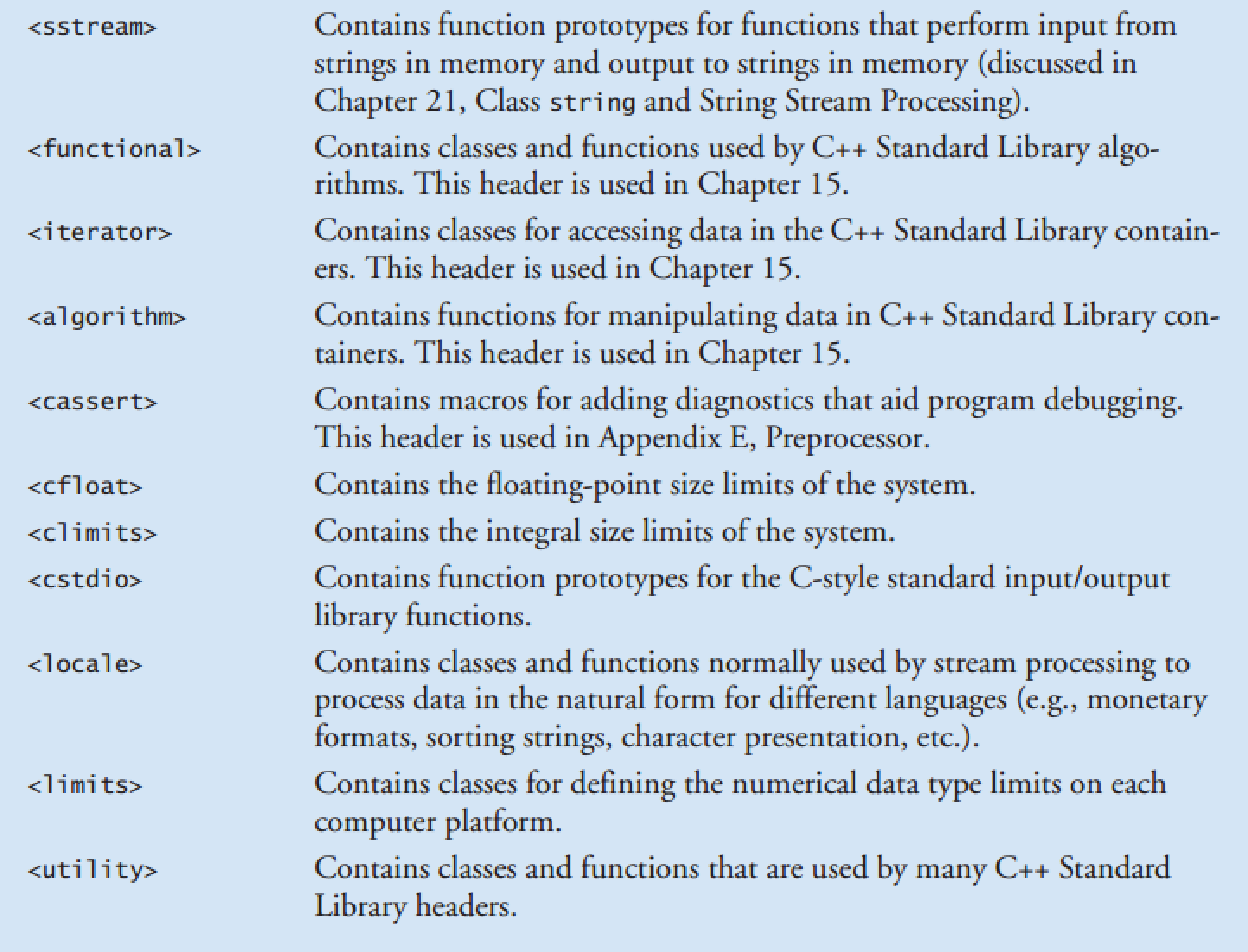
Standard Library header	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
`<iomanip>`	Contains function prototypes for stream manipulators that 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 `struct`s; 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.
`<array>`, `<vector>`, `<list>`, `<forward_list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<unordered_map>`, `<unordered_set>`,	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <vector> header is first introduced in Chapter 7, Class Templates `array` and `vector`; Catching Exceptions. We discuss all these headers in Chapter 15, Standard Library Containers and Iterators.

Standard Libraries



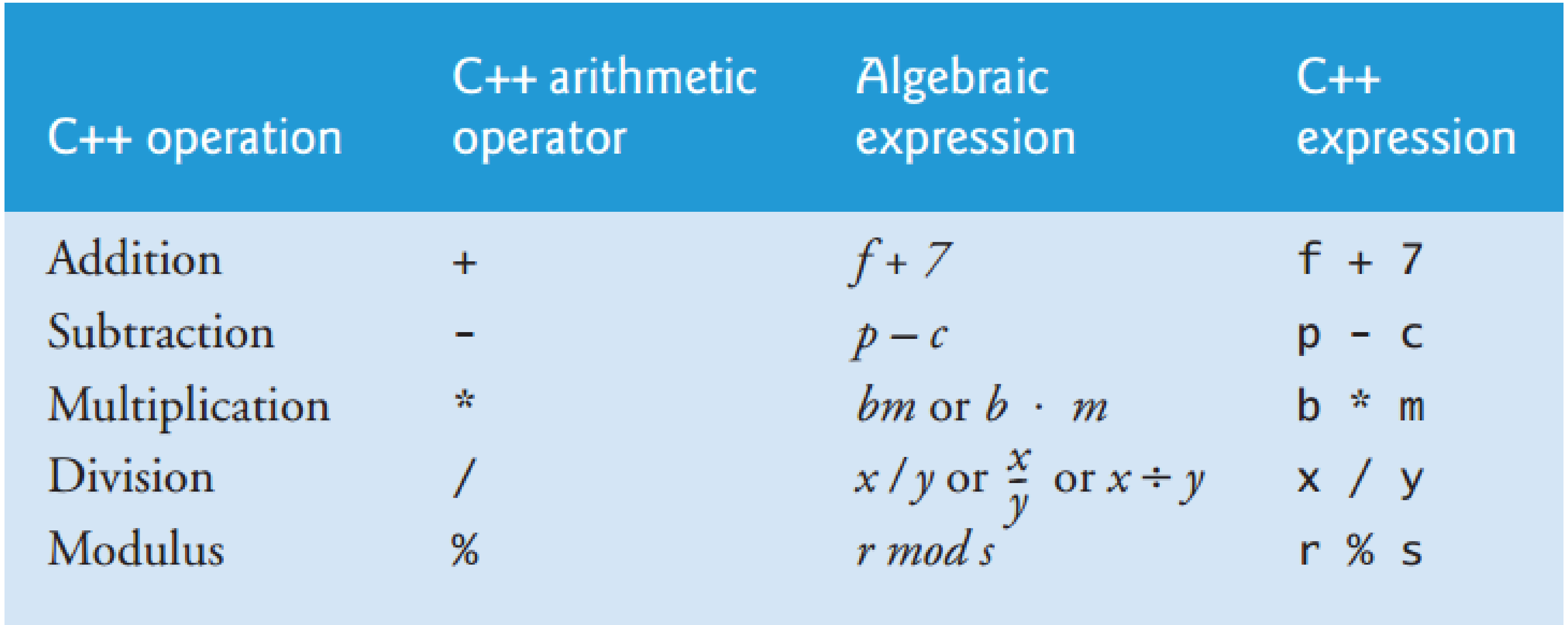
`<set>`, `<bitset>`	
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 22, Bits, Characters, C Strings and `struct`s.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 10, Operator Overloading; Class `string`.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 12.8.
`<exception>`, `<stdexcept>`	These headers contain classes that are used for exception handling (discussed in Chapter 17, Exception Handling: A Deeper Look).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 17, Exception Handling: A Deeper Look.
`<fstream>`	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 14, File Processing).
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 21, Class `string` and String Stream Processing).

Standard Libraries



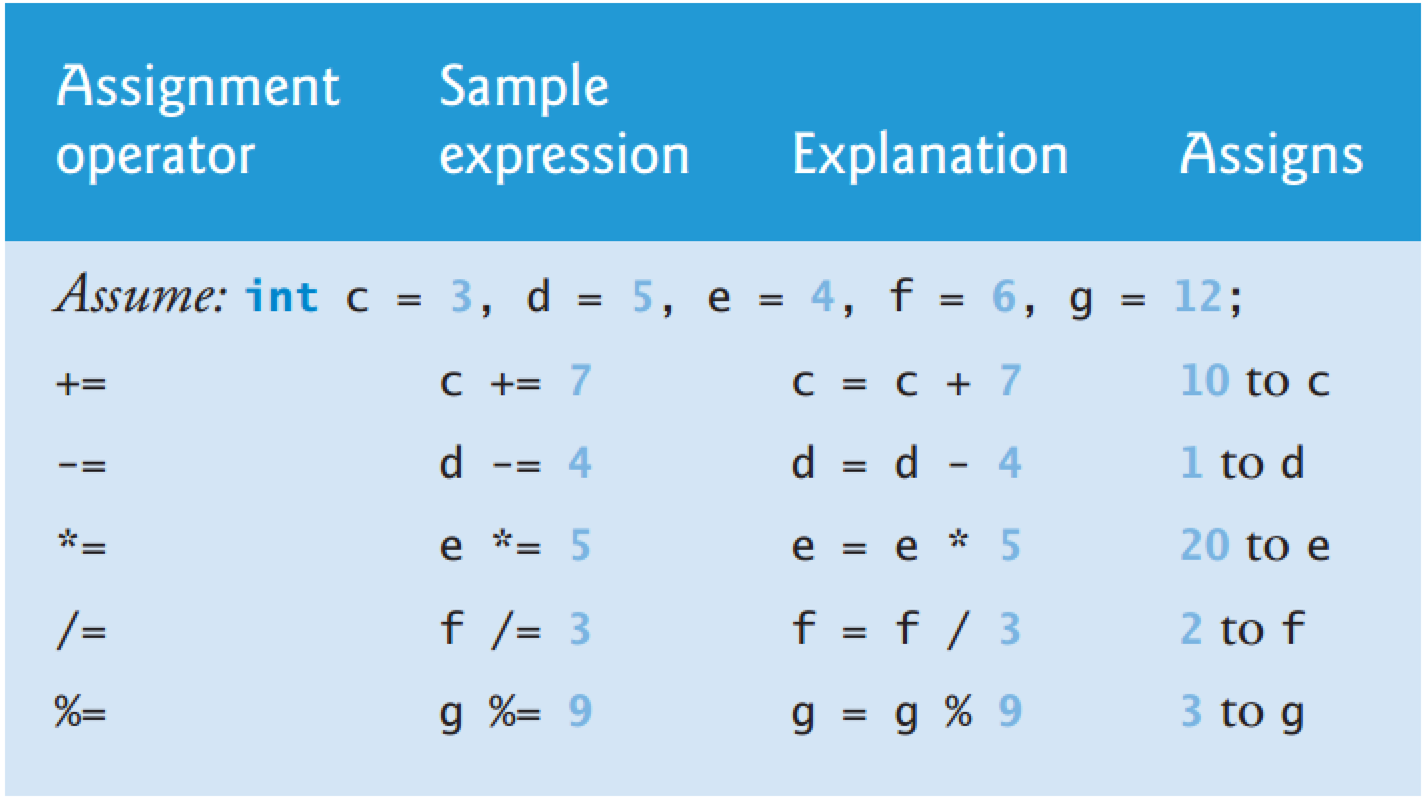
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 21, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 15.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 15.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 15.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
`<limits>`	Contains classes for defining the numerical data type limits on each computer platform.
`<utility>`	Contains classes and functions that are used by many C++ Standard Library headers.

Arithmetic operators



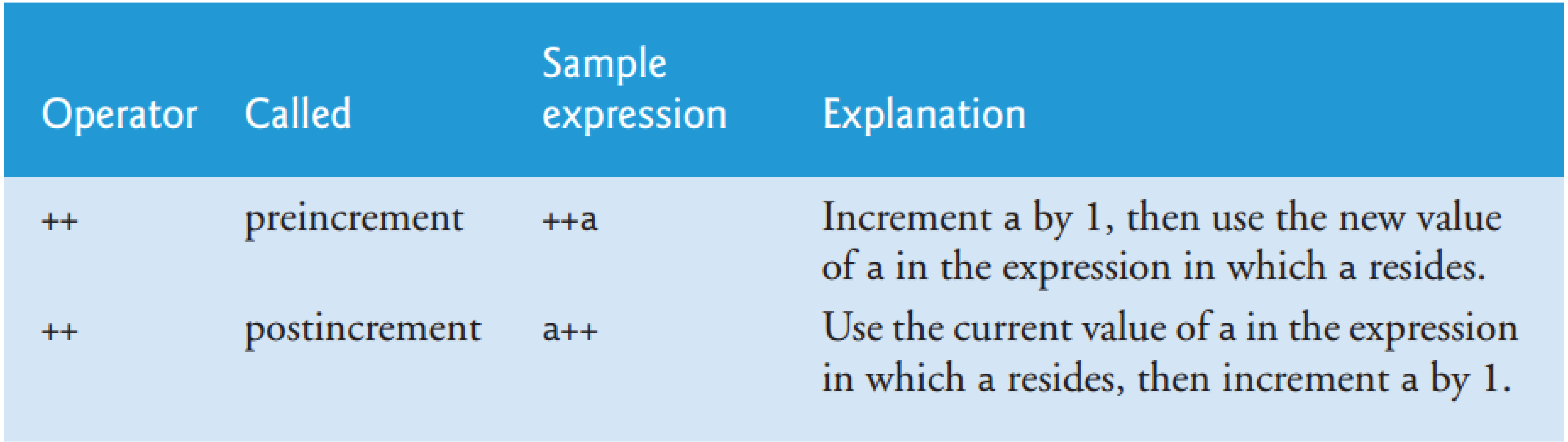
C++ operation	C++ arithmetic operator	Algebraic expression	C++ expression
Addition	+	$f + 7$	f + 7
Subtraction	-	$p - c$	p - c
Multiplication	*	bm or $b \cdot m$	b * m
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	x / y
Modulus	%	$r \bmod s$	r % s

Arithmetic Assignment operators



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

Increment and decrement Operators



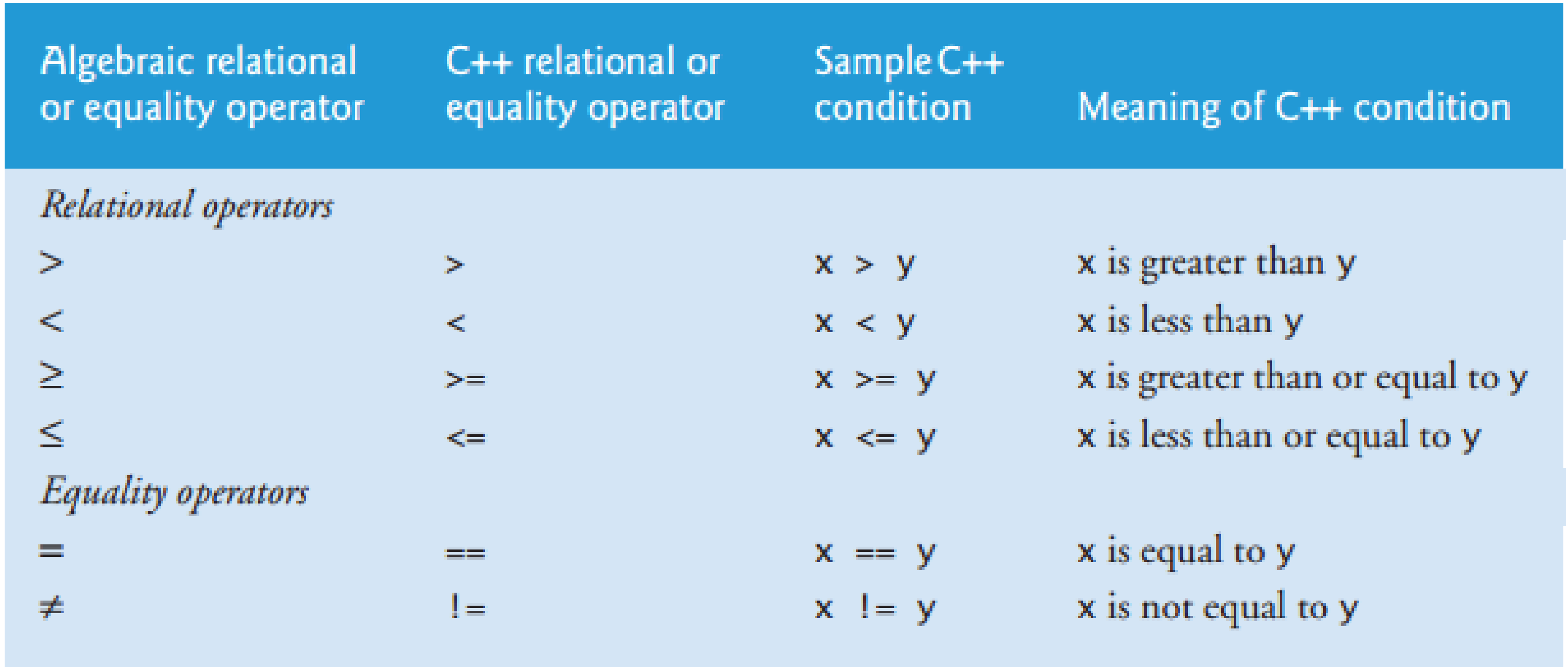
Operator	Called	Sample expression	Explanation
++	preincrement	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postincrement	a++	Use the current value of a in the expression in which a resides, then increment a by 1.

Increment and decrement Operators

- What is output ?

```
int a = 10;
cout << a++ << endl; // prints 10 and then increment
cout << ++a << endl; // increment and then prints 12
```

Relational operators



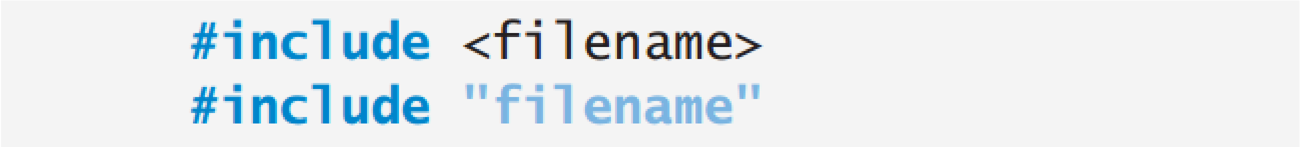
Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators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

using Declarations

- Next Sessions

Preprocessor directives

- Include
  - Causes a copy of a specified file to be included in place of the directive.

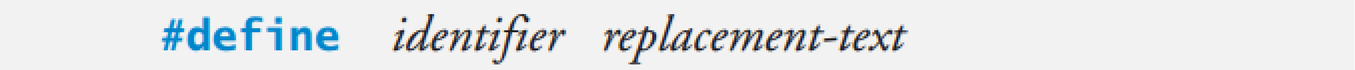


```
#include <filename>
#include "filename"
```

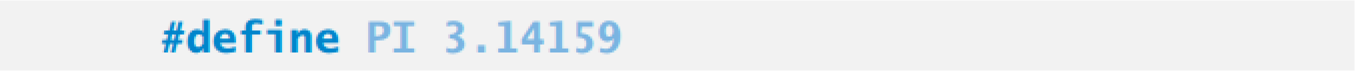
  - <> used for standard library files.
  - “” used for both local header files and standard library files. If local file not found compiler search in standard library files.

Preprocessor directives

- Symbolic constants



```
#define identifier replacement-text
```

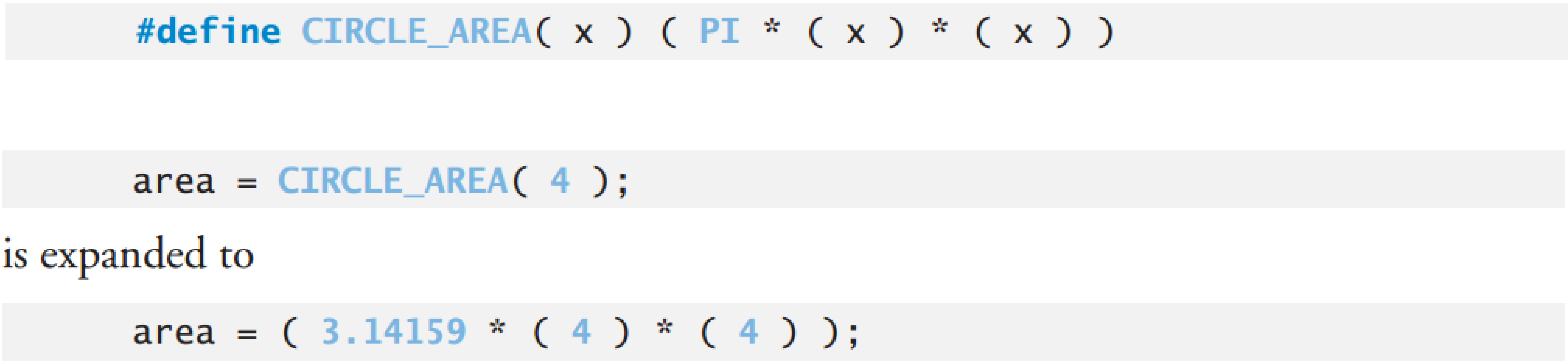


```
#define PI 3.14159
```

  - Using meaningful names for constants
  - Use constants just in files that declared there

Preprocessor directives

- Macros
  - Define functions



```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

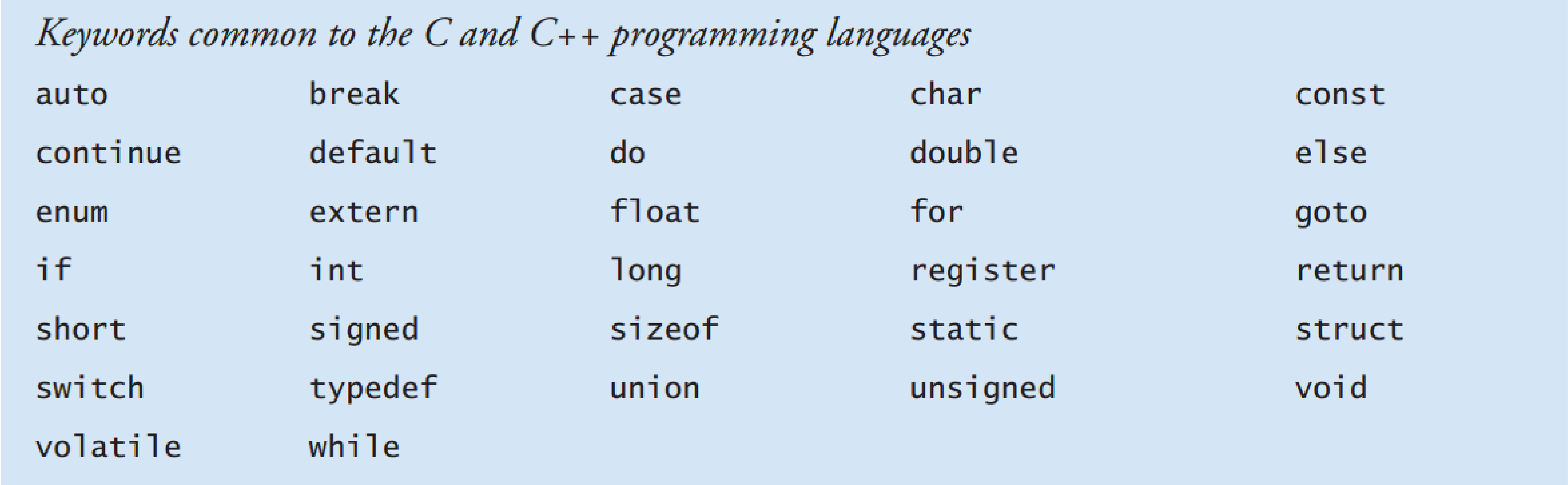
```
area = CIRCLE_AREA( 4 );
```

is expanded to

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

  - What difference between macros and functions ?

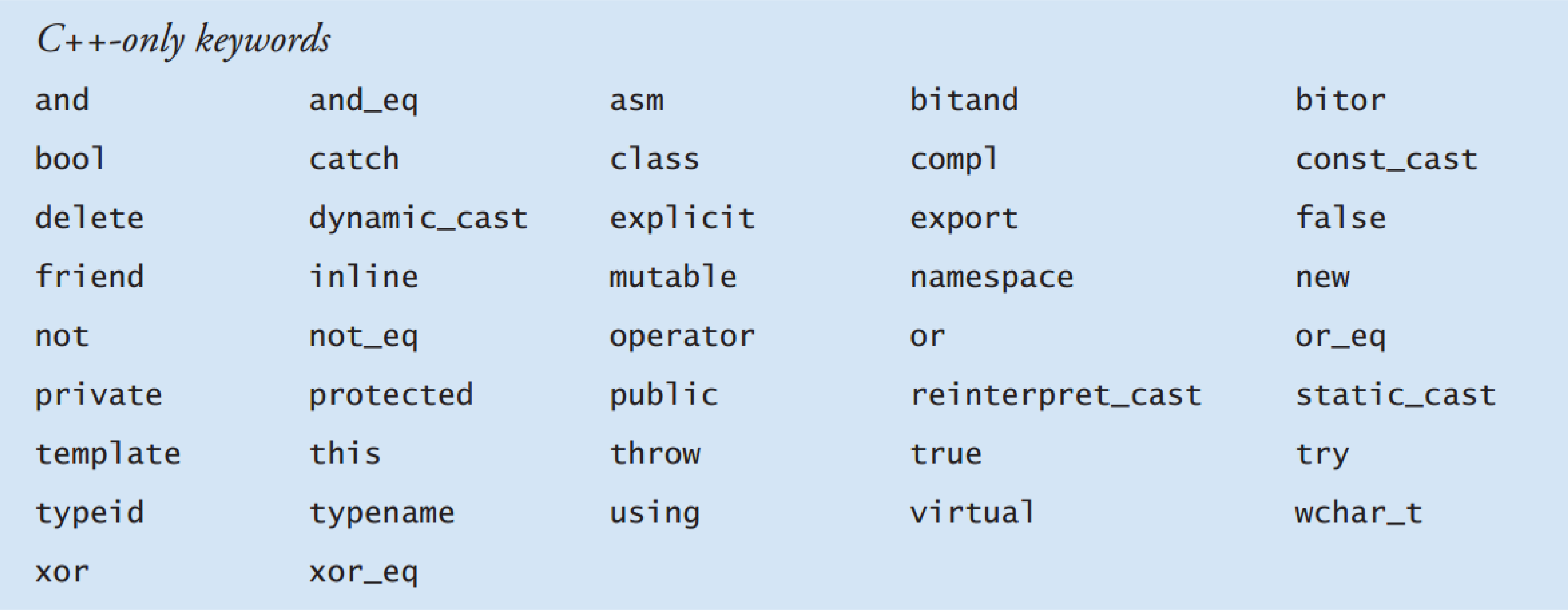
Keywords



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			

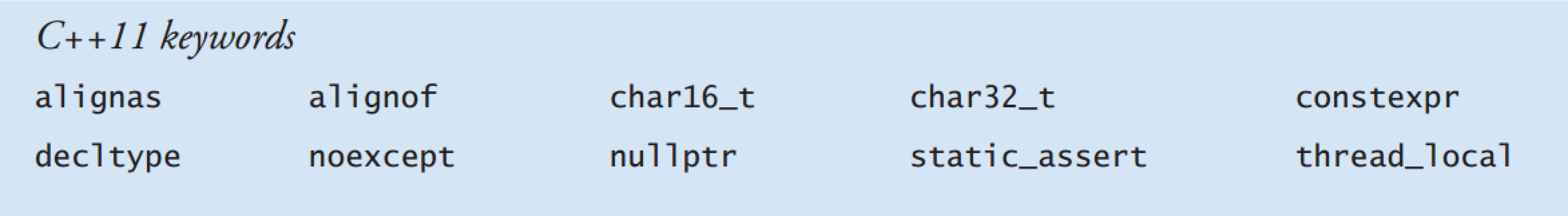
Keywords



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			

Keywords

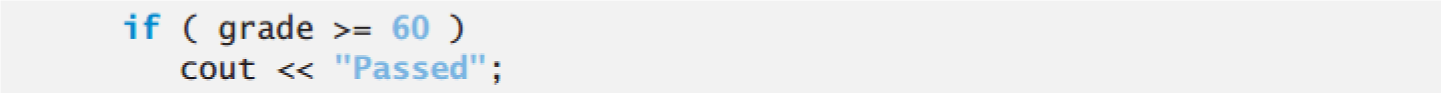


C++11 keywords

`alignas`	`alignof`	`char16_t`	`char32_t`	`constexpr`
`decltype`	`noexcept`	`nullptr`	`static_assert`	`thread_local`

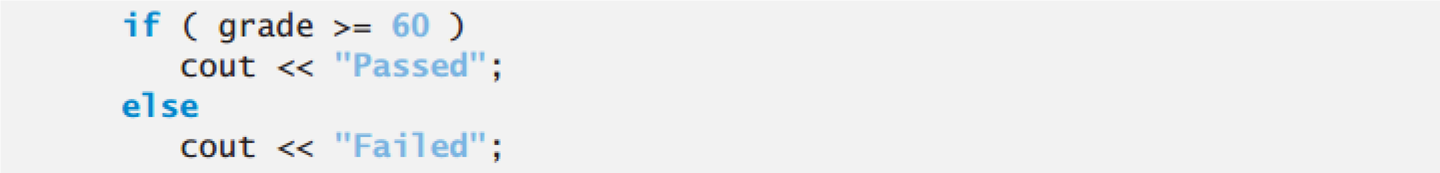
If Statement

- Single if



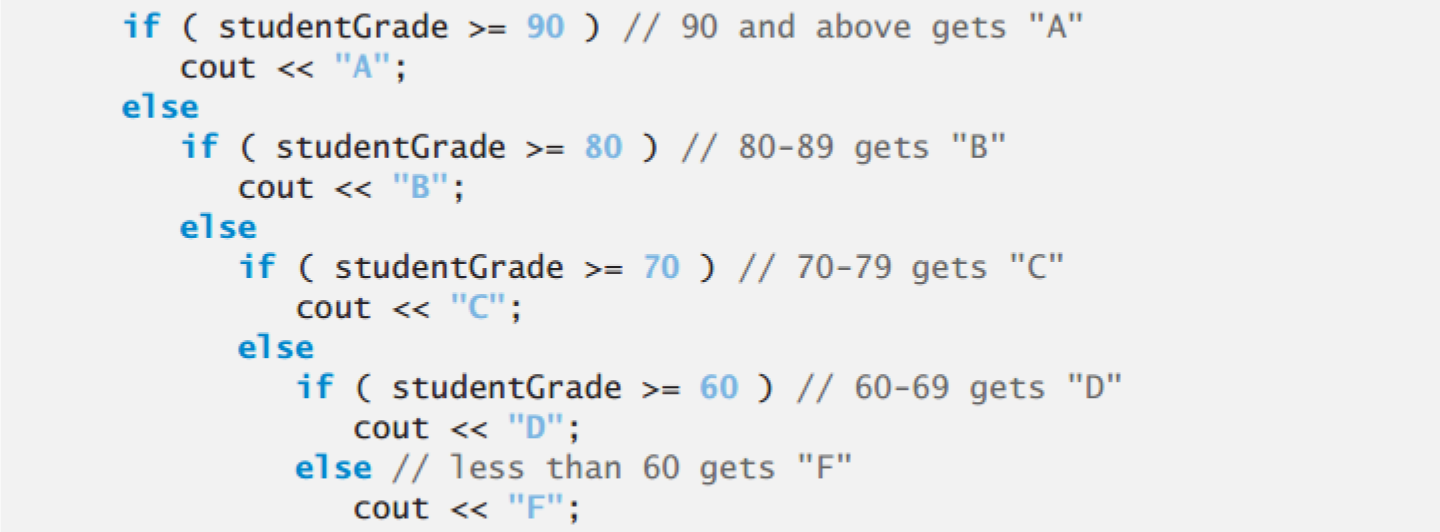
```
if ( grade >= 60 )
   cout << "Passed";
```

- If else



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

- Nested if



```
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";
```

If Statement

- What is output ?

```
int grade = 10;
if (grade >= 18)
    cout << "ok u are young!";
cout << "go to university";
```

*Dangling-**else*** Problem

- What is output ?

```
int x = 10, y = 4;
if (x > 5)
   if(y > 5)
       cout << "x and y are > 5";
else
   cout << "x is <= 5";
```

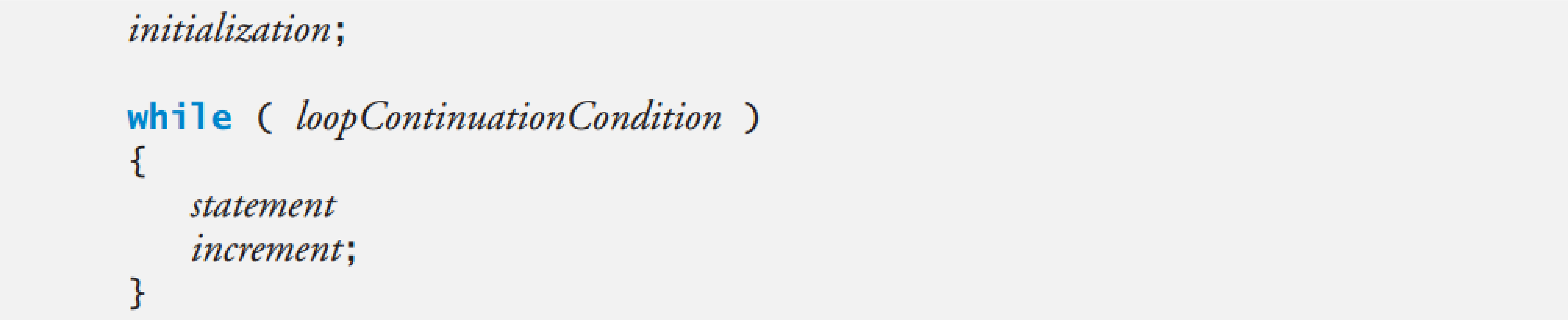
Conditional Operator (?:)

- By Example

```
char c = (age > 18) ? 't' : 'c';
cout << "you are " << ((age > 18) ? "young" : "teenager");
```

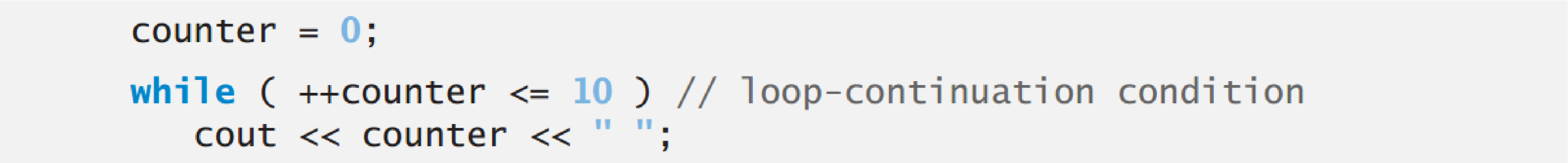
Loops

- While



```
initialization;

while ( loopContinuationCondition )
{
    statement
    increment;
}
```

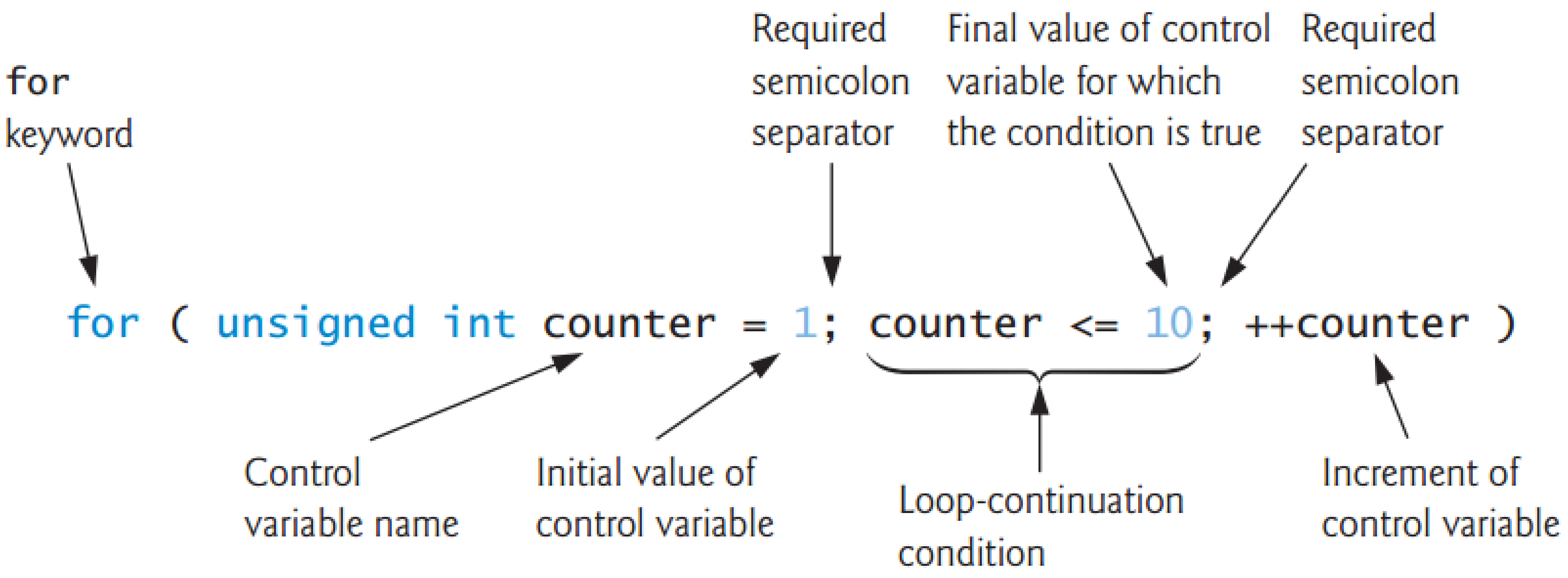


```
counter = 0;

while ( ++counter <= 10 ) // loop-continuation condition
   cout << counter << " ";
```

Loops

- For



```
for ( unsigned int counter = 1; counter <= 10; ++counter )
```

for keyword
Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
Control variable name
Initial value of control variable
Loop-continuation condition
Increment of control variable

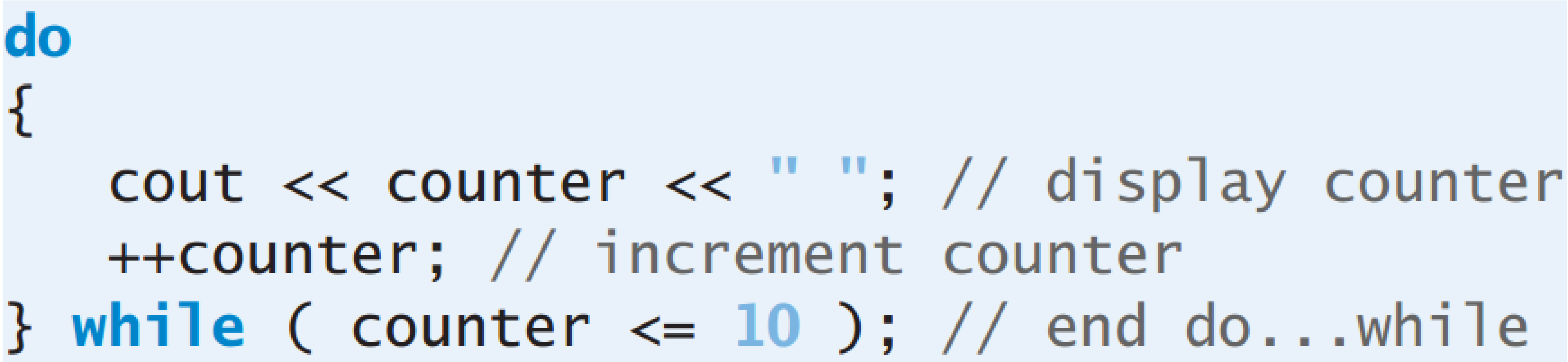
Loops

- For – multi control variables
  - What is output ?

```
for (int x = 1, y = 2; x <= 9 && y <= 10; x += 2, y += 2)
     printf("%d, %d\n", x, y);
```

Loops

- Do while



```
do
{
   cout << counter << " "; // display counter
   ++counter; // increment counter
} while ( counter <= 10 ); // end do...while
```

Nested Loops

- What is output ?

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        printf("Cefiro\n");
    }
}
```

Break Statement

- Break; exit from loop or switch

```
for (int i = 0; i < 10; i++)
{
     if (i == 5)
          break;
     cout << i << " ";
}
```

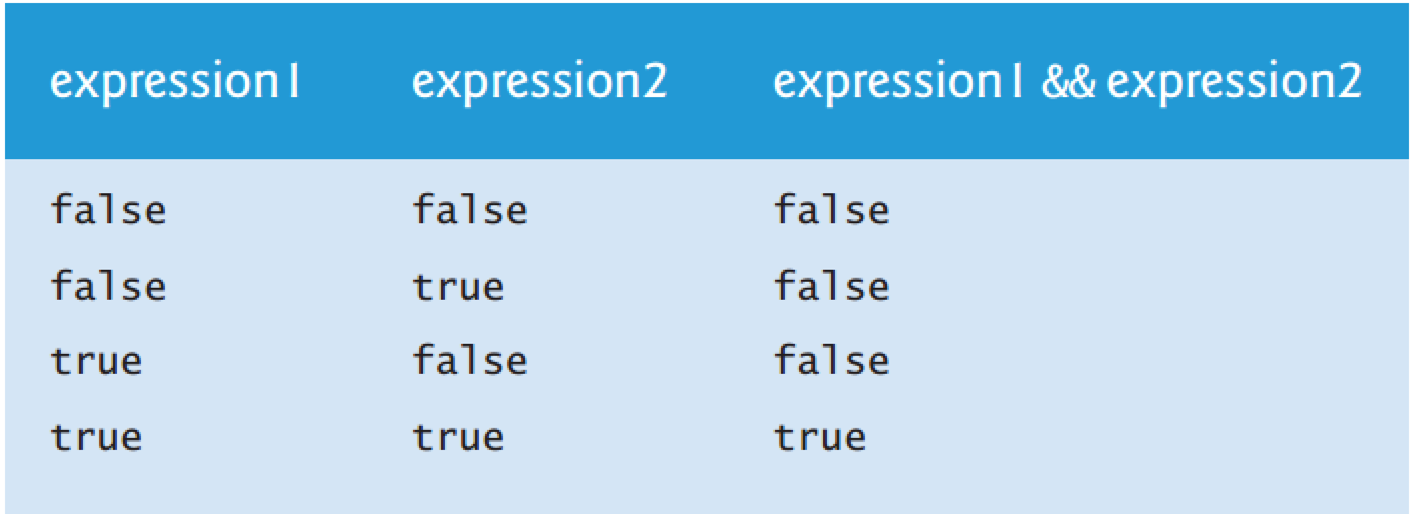
Continue Statement

- Continue; skip remaining statements in current loop

```
for (int i = 0; i < 10; i++)
{
     if (i == 5)
          continue;
     cout << i << " ";
}
```

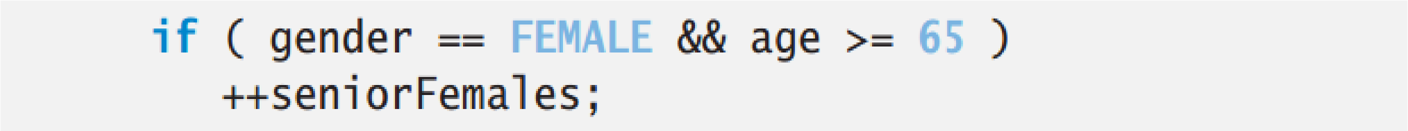
Logical Operators

- AND



expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

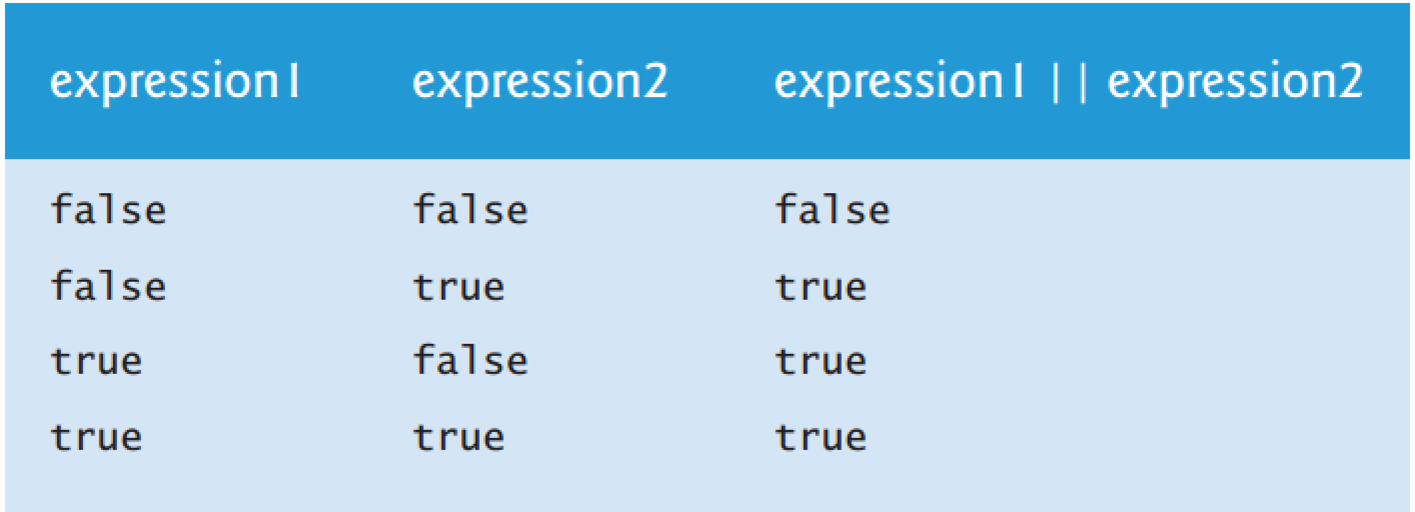
- Example



```
if ( gender == FEMALE && age >= 65 )
   ++seniorFemales;
```

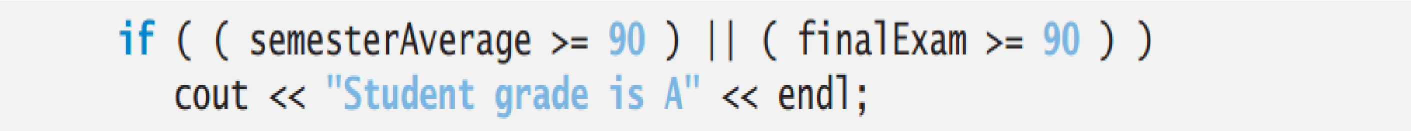
Logical Operators

- OR



expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

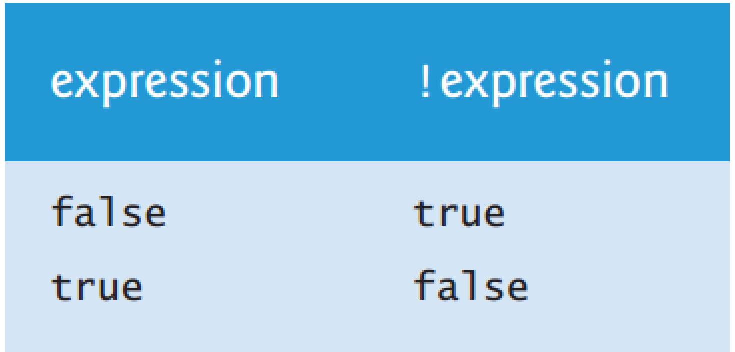
- Example



```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   cout << "Student grade is A" << endl;
```

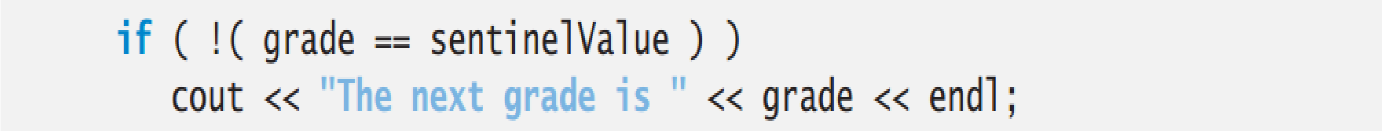
Logical Operators

- NOT



expression	!expression
false	true
true	false

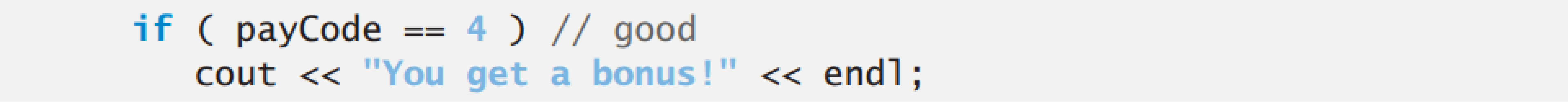
- Example



```
    if ( !( grade == sentinelValue ) )
       cout << "The next grade is " << grade << endl;
```

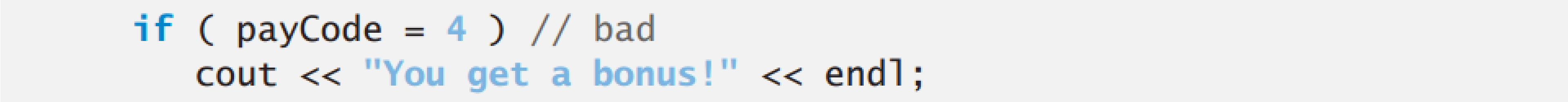
Confusing the == and =

- == is equality



```
    if ( payCode == 4 ) // good
       cout << "You get a bonus!" << endl;
```

- = is assignment

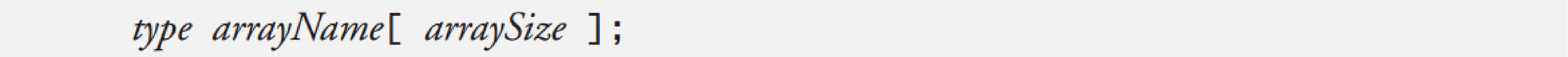


```
    if ( payCode = 4 ) // bad
       cout << "You get a bonus!" << endl;
```

  - What is output?
  - This is not a syntax error. But be careful.

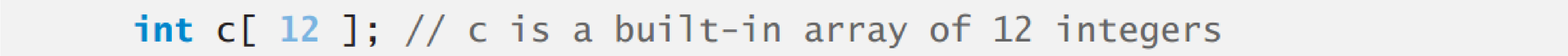
Built-in(fixed-size) Array

- To reserve some elements of one specifics type
- Declaration



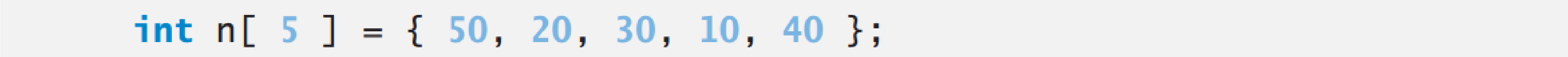
```
type arrayName[ arraySize ];
```

- array size must be a constant integer



```
int c[ 12 ]; // c is a built-in array of 12 integers
```

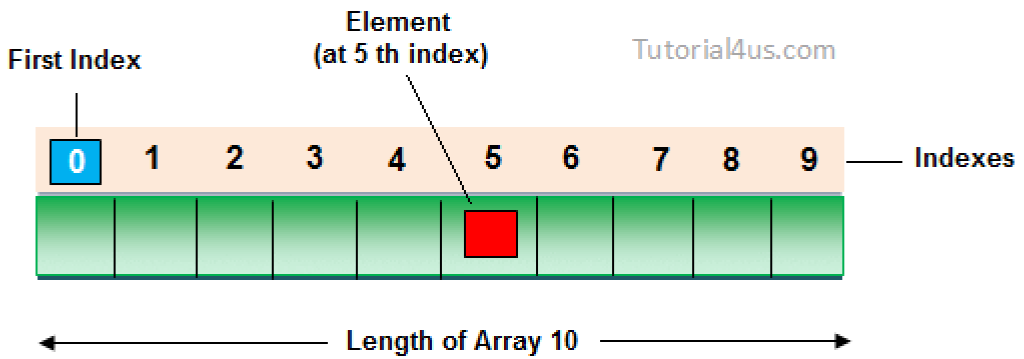
- Initializing Built-In Arrays



```
int n[ 5 ] = { 50, 20, 30, 10, 40 };
```

Built-in(fixed-size) Array

- Index of elements start **from 0 to arraysize - 1**



```
double array [10];
```

- Assign to an element of array
- Read value of an element of array

```
array[0] = 10.01;
cout << array[0];
```

Built-in(fixed-size) Array

- Built-in arrays have several limitations:
  - They *cannot be compared* using the relational and equality operators—you must use a loop to compare two built-in arrays element by element.
  - They *cannot be assigned* to one another
  - They *don’t know their own size*—a function that processes a built-in array typically receives *both* the built-in array’s *name* and its *size* as arguments.
  - They *don’t provide automatic bounds checking*—you must ensure that array-access expressions use subscripts that are within the built-in array’s bounds

Built-in(fixed-size) Array

- Prefer to not use built-in array, instead use pointers, array and vector class.

- But some time required use built in arrays. When ?
    - Programs that get command arguments

Functions

- This is better in develop large programs to construct it from small, simple pieces or components. This technique is called divide and conquer.

- How to divide C++ codes to small components??
  - Using Functions

Functions

- Several motivations for modularizing a program with functions :
  - One is the divide-and-conquer approach
  - Another is software reuse. For example, in earlier programs, we did not have to define how to read a line of text from the keyboard—C++ provides this capability via the getline function of the <string> header
  - To avoid repeating code
  - Also, dividing a program into meaningful functions makes the program easier to debug and maintain

Functions

```
return_type function_name( parameter1, parameter2,…)
{
        body of the function
}
```

Functions

```
int add_2number(int a, char b)
{
      int add = a + b;
      return add;
}
```

Void Type

- Void means nothing
  - In functions return nothing

```
void print (char* str)
{
        cout << str << endl;
}
```

- void*   what does it mean ?

Functions

- Calling Functions

```
int add_2number(int a, char b)
{
      int add = a + b;
      return add;
}

int main ()
{
      int x = 2;
      int y = 3;
      cout << "x + y = " << add_2numbers(x,y);
}
```