

CPP

TABLE OF CONTENTS

- ▷ Introduction
- ▷ Variables and types
 - ▽ Identifiers
 - ▶ List of identifiers
 - ▽ Fundamental Data-Types
 - ▽ Initialization Of Variables
 - ▽ Type deduction : auto and decltype
 - ▽ Introduction To Strings
- ▷ Constants
 - ▽ Literals
 - ▶ Integer Numerals
 - ▶ Floating Point Numerals
 - ▽ Character And String Literals
 - ▽ Other Literals
 - ▽ Typed Constant Expressions
 - ▽ Preprocessor Definitions
- ▷ Operators
 - ▽ Assignment
 - ▽ Arithmetic Operators
 - ▽ Compound Assignment
 - ▽ Increment & Decrement
 - ▽ Relational & Comparison Operators
 - ▽ Logical Operators
 - ▽ Conditional Ternary Operators
 - ▽ Comma Operator
 - ▽ Bitwise Operators
 - ▽ Explicit Type Casting Operator
 - ▽ sizeof
 - ▽ Precedence Of Operators
- ▷ Basic input / output
 - ▽ Standard Output (cout)
 - ▽ Standard Input (cin)
 - ▽ Cin and Strings

▽sstream

▷Statements and control flow

▽Range Based For Loop

▽Jump Statements

▷Functions

▽Pass By Reference

▽Inline Functions

▽Default Values In The Parameters

▽Declaration Of Functions

▷Overloads and templates

▽Overloaded Functions

▽Function Template

▽Non Type Template Arguments

▷Name visibility

▽Namespaces

▽Using Keyword

▽Namespace Aliasing

▽The std Namespace

▽Storage Classes

▷Arrays

▽Arrays As Parameters

▽Library Arrays

▷Character sequences

▽Initialization OF NULL Terminated Character Sequences

▽Strings and null-terminated character sequences

▷Pointers

▽Declaration

▽Pointer & Array

▽Pointer Arithmetic

▽Pointer & Constant

▽Pointers & String Literals

▽Pointers To Pointers

▽Void Pointers

▽Invalid & Null Pointers

▽Pointers To Functions

▷ Dynamic memory

- ▽ Operators new and new[]
- ▽ Operations delete & delete[]
- ▽ Dynamic Memory In C

▷ Data structures

- ▽ Data Structures
- ▽ Pointers To Structures
- ▽ Nesting Structures

▷ Other data types

- ▽ Type Aliases (typedef / using)
- ▽ Unions
- ▽ Anonymous Unions
- ▽ Enumerated Types (enum)
- ▽ Enumerated Types With enum Classes

▷ Classes - i

- ▽ Constructors
- ▽ Overloading Constructors
- ▽ Uniform Initialization
- ▽ Member Initialization In Constructors
- ▽ Pointers To Classes
- ▽ Interpretations
- ▽ Classed Defined With Struct & Union Keywords

▷ Classes - ii

- ▽ Overloading Operators
- ▽ The Keyword this
- ▽ Static Members
- ▽ Const Member Functions
- ▽ Class Templates
- ▽ Template Specialization

▷ Special members

- ▽ Default Constructor
- ▽ Destructor
- ▽ Copy Constructor
- ▽ Copy Assignment
- ▽ Move Constructor & Assignment

▽ Implicit Members

▷ Friendship and inheritance

 ▽ Friend Function

 ▽ Friend Classes

 ▽ Inheritance Between Classes

 ▽ What Is Inherited From The Base Class?

 ▽ Multiple Inheritance

▷ Polymorphism

 ▽ Pointer To Base Class

 ▽ Virtual Members

 ▽ Abstract Base Classes

▷ Type conversions

 ▽ Implicit Conversion

 ▽ Implicit Conversion With Classes

 ▽ Keyword Explicit

 ▽ Type-Casting

 ▽ Dynamic Cast

 ▽ Static_cast

 ▽ Reinterpret_cast

 ▽ Const_cast

 ▽ typeid

 ▽ <cstdint> / (stdint.h)

 ► Types

 ► Macros

 ► Limits of other types

 ► Functions like macros

▷ Exceptions

 ▽ Exception Specifications

 ▽ Standard Exceptions

▷ Preprocessor directives

 ▽ Macro Definitions

 ▽ Conditional Inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

 ▽ Line Control (#line)

 ▽ Error Directive (#error)

- ▽Source File Inclusion (#include)
- ▽Pragma Directive (#pragma)
- ▽Predefined macro names
- ▷Input/output with files
 - ▽Open A File
 - ▽Closing A File
 - ▽Text Files
 - ▽Checking State Flags
 - ▽Get and Put Stream Positioning
 - ▶tellg() and tellp()
 - ▶seekg() and seekp()
 - ▽Binary Files
 - ▽Buffers & Synchronization

INTRODUCTION

- ▷ Preprocessor directives must be specified in their own line and, because they are not statements, do not have to end with a semicolon (;).

- ▷

```
#include <iostream>
using namespace std;
int main() {
    cout<<"Hello World"=><endl; //Unqualified cout
    std::cout<<"Hello World"=><endl; //Qualified directly within std namespace
    return 0;
}
```

- ▷ cout is part of the standard library, and all the elements in the standard C++ library are declared within what is called a namespace: the namespace std.
- ▷ Explicit qualification is the only way to guarantee that name collisions never happen.

VARIABLES AND TYPES

▷ IDENTIFIERS

- ▽ A sequence of one or more letters, digits, or underscore characters (_)
- ▽ Identifiers shall always begin with a letter. They can also begin with an underline character (_), but such identifiers are -on most cases- considered reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere.
- ▽ List of identifiers :
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq
- ▽ The c++ is a case sensitive language ⇒ Example ≠ example ≠ EXAMPLE

▷ FUNDAMENTAL DATA-TYPES

- ▽ Character data type : They can represent a single character, such as 'A' or '\$'. The most basic type is char, which is a one-byte character. Other types are also provided for wider characters.
- ▽ Numerical integer type : They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be signed or unsigned, depending on whether they support negative values or not.

- ▽ Floating-point type : They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.
- ▽ Boolean type : The boolean type, known in C++ as `bool`, can only represent one of two states, true or false.

▽

Here is the complete list of fundamental types in C++:		
Group	Type names*	Notes on size / precision
Character types	<code>char</code>	Exactly one byte in size. At least 8 bits.
	<code>char16_t</code>	Not smaller than <code>char</code> . At least 16 bits.
	<code>char32_t</code>	Not smaller than <code>char16_t</code> . At least 32 bits.
	<code>wchar_t</code>	Can represent the largest supported character set.
Integer types (signed)	<code>signed char</code>	Same size as <code>char</code> . At least 8 bits.
	<code>signed short int</code>	Not smaller than <code>char</code> . At least 16 bits.
	<code>signed int</code>	Not smaller than <code>short int</code> . At least 16 bits.
	<code>signed long int</code>	Not smaller than <code>int</code> . At least 32 bits.
	<code>signed long long int</code>	Not smaller than <code>long</code> . At least 64 bits.
Integer types (unsigned)	<code>unsigned char</code>	
	<code>unsigned short int</code>	
	<code>unsigned int</code>	(same size as their signed counterparts)
	<code>unsigned long int</code>	
	<code>unsigned long long int</code>	
Floating-point types	<code>float</code>	
	<code>double</code>	Precision not less than <code>float</code>
	<code>long double</code>	Precision not less than <code>double</code>
Boolean type	<code>bool</code>	
Void type	<code>void</code>	no storage
Null pointer	<code>decltype(nullptr)</code>	

▽

```

Char data types :
Size of char 1 bytes
Size of char16_t 2 bytes
Size of char32_t 4 bytes
Size of wchar_t 4 bytes

Size of Signed Int data types :
Size of signed char 1 bytes
Size of signed short int 2 bytes
Size of signed int 4 bytes
Size of signed long int 8 bytes
Size of signed long long int 8 bytes

Size of Unsigned Int data types :
Size of unsigned char 1 bytes
Size of unsigned short int 2 bytes
Size of unsigned int 4 bytes
Size of unsigned long int 8 bytes
Size of unsigned long long int 8 bytes

Size of Floating-point data types :
Size of float 4 bytes
Size of double 8 bytes
Size of long double 16 bytes

Size of boolean data type :
Size of bool 1 bytes

Size of void type
Size of void 1 bytes

Size of decltype(nullptr) :
Size of decltype(nullptr) 8 bytes

```

- ▽ * The names of certain integer types can be abbreviated without their signed and int components - only the part not in italics is required to identify the type, the part in italics is optional. I.e., `signed short int` can be abbreviated as `signed short`, `short int`, or simply `short`; they all identify the same fundamental type.
- ▽ Only `char` has the exact size of 1 byte. Other fundamental data types do not have an exact size. They have a minimum and atmost sizes. This does not mean that these types are of an undetermined

size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run.



Size	Unique representable values		Notes
8-bit	256 = 2^8		
16-bit	65 536 = 2^{16}		
32-bit	4 294 967 296 = 2^{32} (~4 billion)		
64-bit	18 446 744 073 709 551 616		= 2^{64} (~18 billion billion)

▽ The types described above (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: void, which identifies the lack of type; and the type nullptr, which is a special type of pointer.

▷ INITIALIZATION OF VARIABLES

▽ In C++, there are three ways to initialize variables. They are all equivalent and are reminiscent of the evolution of the language over the years:

- ▶ The first one, known as c-like initialization (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized:
`type identifier = initial_value;`

For example,

```
int x = 0;
```

- ▶ A second method, known as constructor initialization (introduced by the C++ language), encloses the initial value between parentheses ():
`type identifier (initial_value);`

For example:

```
int x (0);
```

- ▶ A third method, known as uniform initialization, similar to the above, but using curly braces ({}) instead of parentheses (this was introduced by the revision of the C++ standard, in 2011):
`type identifier {initial_value};`

For example: `int x {0};`

▷ TYPE DEDUCTION : AUTO AND DECLTYPE

▽ When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use auto as the type specifier for the variable:

```
auto x=33;
```

▽ Variables that are not initialized can also make use of type deduction with the decltype specifier:

```
int foo=0;
decltype(foo) bar;
```

▷ INTRODUCTION TO STRINGS

▽ It is a compound type.

▽ Defined in header <string>

▽

```
1 // my first string
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string mystring;
9     mystring = "This is a string";
10    cout << mystring;
11    return 0;
12 }
```

▽ Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and change its value during execution.

▷ Note: inserting the endl manipulator ends the line (printing a newline character and flushing the stream).

CONSTANTS

▷ LITERALS

- ▽ They are used to express particular values within the source code of a program.
- ▽ Eg. int a=5; //here 5 is a literal constant
- ▽ Literal constants can be classified into: integer, floating-point, characters, strings, Boolean, pointers, and user-defined literals.

▽ Integer Numerals

- ▶ They are a simple succession of digits representing a whole number in decimal base.
- ▶ In addition to decimal numbers (those that most of us use every day), C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16) as literal constants. For octal literals, the digits are preceded with a 0 (zero) character. And for hexadecimal, they are preceded by the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75 //decimal  
0113 //octal  
0x4b //hexadecimal
```

- ▶ These literal constants have a type, just like variables. By default, integer literals are of type int. However, certain suffixes may be appended to an integer literal to specify a different integer type:

```
u or U for unsigned  
l or L for long  
ll or LL for long long
```

- ▶ Unsigned may be combined with any of the other two in any order to form unsigned long or unsigned long long.

- ▶

```
1 75          // int  
2 75u         // unsigned int  
3 75l         // long  
4 75ul        // unsigned long  
5 75lu        // unsigned long long
```

▽ Floating Point Numerals

- They express real values, with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character:

```
1 3.14159    // 3.14159
2 6.02e23    // 6.02 x 10^23
3 1.6e-19    // 1.6 x 10^-19
4 3.0        // 3.0
```

- The default type for floating-point literals is double.

Floating-point literals of type float or long double can be specified by adding one of the following suffixes:

f or F for float

l or L for long double

eg. 3.14158L //long double

6.02e23f //float

▷ CHARACTER AND STRING LITERALS

▽ character literal : 'x'

string literal : "hello"

▽ Character and string literals can also represent special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t).

These special characters are all of them preceded by a backslash character (\).

Escape code	Description
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\\	backslash (\)

- ▽ Internally, computers represent characters as numerical codes: most typically, they use one extension of the ASCII character encoding system
- The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are representable characters:

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

- ▽ Because most systems nowadays work with 8bit bytes, which can represent 256 different values, in addition to the 128 standard ASCII codes there are other 128 that are known as extended ASCII, which are platform- and locale-dependent. So there is more than one extended ASCII character set.

Standard 7-bit ASCII Table																theascii.com				
Dec	Hex	Oct	Binary	Char	Description	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char
0	0	0	0		Null character	32	20	40	1000000	space	64	40	100	1000000	@	96	60	140	1100000	
1	1	1	1		SOH Start of header	33	21	41	1000001	!	65	41	101	1000001	A	97	61	141	1100001	a
2	2	2	10		STX Start of text	34	22	42	1000010	"	66	42	102	1000010	B	98	62	142	1100010	b
3	3	3	11		ETX End of text	35	23	43	1000011	#	67	43	103	1000011	C	99	63	143	1100011	c
4	4	4	100		EOT End of transmission	36	24	44	1001000	\$	68	44	104	1000100	D	100	64	144	1100100	d
5	5	5	101		ENQ Enquiry	37	25	45	1001001	%	69	45	105	1000101	E	101	65	145	1100101	e
6	6	6	110		ACK Acknowledge	38	26	46	1001100	&	70	46	106	1000110	F	102	66	146	1100110	f
7	7	7	111		BELL Bell ring	39	27	47	1001111	*	71	47	107	1000111	G	103	67	147	1100111	g
8	8	8	1000		BS Backspace	40	28	48	1010000	(72	48	110	1001000	H	104	68	150	1101000	h
9	9	9	1001		HT Horizontal tab	41	29	49	1010001)	73	49	111	1001001	I	105	69	151	1101001	i
10	10	10	1010		LF Line feed	42	2A	50	1010100	*	74	4A	112	1001010	J	106	6A	152	1101010	j
11	OB	13	1011		VT Vertical tab	43	2B	51	1010111	+	75	4B	113	1001011	K	107	6B	153	1101011	k
12	OC	14	1100		FF Form feed	44	2C	54	1011000	-	76	4C	114	1001100	L	108	6C	154	1101100	l
13	OD	15	1101		CR Carriage return	45	2D	55	1011001	-	77	4D	115	1001101	M	109	6D	155	1101101	m
14	OE	16	1110		SO Shift out	46	2E	56	1011010	:	78	4E	116	1001110	N	110	6E	156	1101110	n
15	OF	17	1111		SI Shift in	47	2F	57	1011111	/	79	4F	117	1001111	O	111	6F	157	1101111	o
16	10	20	10000		DLE Data link escape	48	30	60	1100000	0	80	50	120	1010000	P	112	70	160	1110000	p
17	11	21	10001		DC1 Device control 1	49	31	61	1100001	1	81	51	121	1010001	Q	113	71	161	1110001	q
18	12	22	10010		DC2 Device control 2	50	32	62	1100010	2	82	52	122	1010010	R	114	72	162	1110010	r
19	13	23	10011		DC3 Device control 3	51	33	63	1100011	3	83	53	123	1010011	S	115	73	163	1110011	s
20	14	24	10100		DC4 Device control 4	52	34	64	1101000	4	84	54	124	1010100	T	116	74	164	1110100	t
21	15	25	10101		NAK Negative acknowledge	53	35	65	1101001	5	85	55	125	1010101	U	117	75	165	1110101	u
22	16	26	10110		SYN Synchronize	54	36	66	1101010	6	86	56	126	1010110	V	118	76	166	1110110	v
23	17	27	10111		ETB End transmission block	55	37	67	1101111	7	87	57	127	1010111	W	119	77	167	1110111	w
24	18	30	11000		CAN Cancel	56	38	68	1110000	8	88	58	130	1011000	X	120	78	170	1111000	x
25	19	31	11001		EM End of medium	57	39	69	1110001	9	89	59	131	1011001	Y	121	79	171	1111001	y
26	1A	32	11010		SUB Substitute	58	3A	70	1110100	:	90	5A	132	1011010	Z	122	7A	172	1111010	z
27	1B	33	11011		ESC Escape	59	3B	71	1110111	:	91	5B	133	1011011	!	123	7B	173	1111011	!
28	1C	34	11100		FS File separator	60	3C	74	1111000	<	92	5C	134	1011100	\	124	7C	174	1111100	\
29	1D	35	11101		GS Group separator	61	3D	75	1111011	=	93	5D	135	1011101	!	125	7D	175	1111101	!
30	1E	36	11110		RS Record separator	62	3E	76	1111100	>	94	5E	136	1011110	^	126	7E	176	1111110	^
31	1F	37	11111		US Unit separator	63	3F	77	1111111	?	95	5F	137	1011111	DEL	127	7F	177	1111111	DEL

- ▽ Characters can also be represented in literals using its numerical code by writing a backslash character (\) followed by the code expressed as an octal (base-8) or hexadecimal (base-16) number. For an octal value, the backslash is followed directly by the digits; while for hexadecimal, an x character is inserted

between the backslash and the hexadecimal digits themselves (for example: \x20 or \x4A).

▽ Several string literals can be concatenated to form a single string literal simply by separating them by one or more blank spaces, including tabs, newlines, and other valid blank characters. For example:

```
"this forms" "a single"      " string "
"of characters"
```

The above is a string literal equivalent to:

```
"this forms a single string of characters"
```

▽ Some programmers also use a trick to include long string literals in multiple lines: In C++, a backslash (\) at the end of line is considered a line-continuation character that merges both that line and the next into a single line. Therefore the following code:

```
x = "string expressed in \
two lines"
is equivalent to:
x = "string expressed in two lines"
```

▽ All the character literals and string literals described above are made of characters of type **char**. A different character type can be specified by using one of the following **prefixes**:

u for `char16_t`
U for `char32_t`
L for `wchar_t`

▽ Here the prefixes are case sensitive unlike others.

▽ Two additional prefixes also exist :

u8 for encoding the executable in utf8
R for raw string

▽ In raw strings, backslashes and single and double quotes are all valid characters; the content of the literal is delimited by an initial R"sequence(and a final)sequence", where sequence is any sequence of characters (including an empty sequence). The content of the string is what lies inside the parenthesis, ignoring the delimiting sequence itself. For example:

```
R"(string with \backslash)"  
R"$$(string with \backslash)$"
```

▷ OTHER LITERALS

- ▽ Three keyword literals exist in C++: `true`, `false` and `nullptr`:
`true` and `false` are the two possible values for variables of type `bool`.
`nullptr` is the `null` pointer value.

▷ TYPED CONSTANT EXPRESSIONS

- ▽ Sometimes, it is just convenient to give a name to a constant value:
`const double pi = 3.1415926;`
`const char tab='t'`

▷ PREPROCESSOR DEFINITIONS

- ▽ Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:
`#define identifier replacement`
- ▽ This replacement is performed by the preprocessor, and happens before the program is compiled, thus causing a sort of blind replacement: the validity of the types or syntax involved is not checked in any way.

OPERATORS

▷ ASSIGNMENT

▽ Assignment operations are expressions that can be evaluated. That means that the assignment itself has a value, and -for fundamental types- this value is the one assigned in the operation. For example:

```
y = 2 + (x = 5);
```

In this expression, y is assigned the result of adding 2 and the value of another assignment expression (which has itself a value of 5). It is roughly equivalent to:

```
x = 5;
```

```
y = 2 + x;
```

▽ The following expression is also valid in C++:

```
x = y = z = 5;
```

It assigns 5 to the all three variables: x, y and z; always from right-to-left.

▷ ARITHMETIC OPERATORS

▽ The five arithmetical operations supported by C++ are: addition, subtraction, multiplication, division, modulo

▷ COMPOUND ASSIGNMENT

▽ `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `%=` `^=` `|=`

▷ INCREMENT & DECREMENT

▽ Equivalent to `+=1` and `-=1`

▽ In the case that the increase operator is used as a prefix (`++x`) of the value, the expression evaluates to the final value of x, once it is already increased. On the other hand, in case that it is used as a suffix (`x++`), the value is also increased, but the expression evaluates to the value that x had before being increased.

▷ RELATIONAL & COMPARISON OPERATORS

▽ The result of such an operation is either true or false (i.e., a Boolean value).

▽ = > < ≥ ≤ ≠

▽ Of course, it's not just numeric constants that can be compared, but just any value, including, of course, variables.

▷ LOGICAL OPERATORS

▽ ! && ||

▽ When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in the last example `((5==5)|| (3>6))`, C++ evaluates first whether `5==5` is true, and if so, it never checks whether `3>6` is true or not. This is known as short-circuit evaluation, and works like this for these operators:

operator	short-circuit
&&	if the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated).
	if the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated).

▽ This is mostly important when the right-hand expression has side effects, such as altering values:

```
if ( (i<10) && (++i<n) ) { /* ... */ } // note that the condition increments i
```

▷ CONDITIONAL TERNARY OPERATORS

▽ `condition ? result1 : result2` //evaluates to result1 if condition is true

▷ COMMA OPERATOR

▽ The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

▷ BITWISE OPERATORS

▽ & | ^ ~ << >>

▽ Bitwise operators modify variables considering the bit patterns that represent the values they store.

▽

operator	asm equivalent	description
&	AND	Bitwise AND
	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

▷ EXPLICIT TYPE CASTING OPERATOR

▽ Precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;  
float f = 3.14;  
i = (int) f;
```

▽ Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

▷ SIZEOF

▽ This operator accepts one parameter, which can be either a type or a variable, and returns the size in bytes of that type or object.

▽ The value returned by sizeof is a compile-time constant, so it is always determined before program execution.

▷ PRECEDENCE OF OPERATORS



From greatest to smallest priority, C++ operators are evaluated in the following order:				
Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
		++ --	prefix increment / decrement	
3	Prefix (unary)	~ !	bitwise NOT / logical NOT	Right-to-left
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(type)	C-style type-casting	
		.* ->*	access pointer	
		* / %	multiply, divide, modulo	
		+ -	addition, subtraction	
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or		bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction		logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -= >>= <<= &= ^= =	assignment / compound assignment	Right-to-left
		? :	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

▽ When an expression has two operators with the same precedence level, grouping determines which one is evaluated first: either left-to-right or right-to-left.

▽ Parts of the expressions can be enclosed in parenthesis to override this precedence order, or to make explicitly clear the intended effect.

BASIC INPUT / OUTPUT

- ▷ C++ uses a convenient abstraction called streams to perform input and output operations in sequential media such as the screen, the keyboard or a file. A stream is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).
- ▷ Streams provided by standard library:
 - cin : standard input
 - cout : standard output
 - cerr : standard error(output)
 - clog : standard logging(output)
- ▷ STANDARD OUTPUT (cout)
 - ▽ Its a c++ stream object.
 - ▽ endl : Is a manipulator
 - ▽ \n : Is a character
 - ▽ The endl manipulator produces a newline character, exactly as the insertion of '\n' does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly fully buffered streams, and cout is (generally) not a fully buffered stream. Still, it is generally a good idea to use endl only when flushing the stream would be a feature and '\n' when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.
- ▷ STANDARD INPUT (cin)
 - ▽ The extraction operation on cin uses the type of the variable after the >> operator to determine how it interprets the characters read from the input; if it is an integer, the format



```
1 // stringstream
2 #include <iostream>
3 #include <string>
4 #include <sstream>
5 using namespace std;
6
7 int main ()
8 {
9     string mystr;
10    float price=0;
11    int quantity=0;
12
13    cout << "Enter price: ";
14    getline (cin,mystr);
15    stringstream(mystr) >> price;
16    cout << "Enter quantity: ";
17    getline (cin,mystr);
18    stringstream(mystr) >> quantity;
19    cout << "Total price: " << price*quantity << endl;
20    return 0;
21 }
```

```
Enter price: 22.25
Enter quantity: 7
Total price: 155.75
```

In this example, we acquire numeric values from the standard input indirectly: Instead of extracting numeric values directly from `cin`, we get lines from it into a string object (`mystr`), and then we extract the values from this string into the variables `price` and `quantity`. Once these are numerical values, arithmetic operations can be performed on them, such as multiplying them to obtain a total price.

With this approach of getting entire lines and extracting their contents, we separate the process of getting user input from its interpretation as data, allowing the input process to be what the user expects, and at the same time gaining more control over the transformation of its content into useful data by the program.

STATEMENTS AND CONTROL FLOW

- ▷ A for loop with no condition is equivalent to a loop with true as condition (i.e., an infinite loop). Eg. `for(int i=0; ;i++){ s1;}`
- ▷ We can use multiple expression in the for loop. As expressions, they can, however, make use of the comma operator (,): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

```
for ( n=0, i=100 ; n!=i ; ++n, --i ){  
    // whatever here ...  
}
```

- ▷ RANGE BASED FOR LOOP

▽ The for-loop has another syntax, which is used exclusively with ranges:

```
for ( declaration : range ) statement;
```

▽ This kind of for loop iterates over all the elements in range, where declaration declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end.

```
1 // range-based for loop  
2 #include <iostream>  
3 #include <string>  
4 using namespace std;  
5  
6 int main ()  
7 {  
8     string str {"Hello!"};  
9     for (char c : str)  
10    {  
11        cout << "[" << c << "]";  
12    }  
13    cout << '\n';  
14 }
```

[H] [e] [l] [1] [o] [!]

▽ Range based loops usually also make use of type deduction for the type of the elements with auto. Typically, the range-based loop above can also be written as:

```
1 for (auto c : str)  
2     cout << "[" << c << "]";
```

▷ JUMP STATEMENTS

- ▽ Break : break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end.
- ▽ Continue : The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.
- ▽ The Goto Statement : goto allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

```
1 // goto loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int n=10;
8     mylabel:
9     cout << n << ", ";
10    n--;
11    if (n>0) goto mylabel;
12    cout << "liftoff!\n";
13 }
```

- ▽ Switch : The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating if-else statements, but limited to constant expressions. It uses labels instead of blocks. Its most typical syntax is:

```
switch (expression)
{
    case constant1:
        group-of-statements-1;
        break;
    case constant2:
        group-of-statements-2;
        break;
    .
    .
    default:
        default-group-of-statements
}
```

FUNCTIONS

- ▷ If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement: `return 0; //this applies only to main`
- ▷ The values for main that are guaranteed to be interpreted in the same way on all platforms are:

value	description
0	The program was successful
EXIT_SUCCESS	The program was successful (same as above). This value is defined in header <cstdlib>.
EXIT_FAILURE	The program failed. This value is defined in header <cstdlib>.

`EXIT_SUCCESS` & `EXIT_FAILURE` are macros that expand to system-dependent integral expression that, when used with the function `exit({void exit(int status)})` signifies that the application succeeded or failed. Eg. `exit(EXIT_SUCCESS)`

- ▷ PASS BY REFERENCE



```
1 // passing parameters by reference
2 #include <iostream>
3 using namespace std;
4
5 void duplicate (int& a, int& b, int& c)
6 {
7     a*=2;
8     b*=2;
9     c*=2;
10 }
11
12 int main ()
13 {
14     int x=1, y=3, z=7;
15     duplicate (x, y, z);
16     cout << "x=" << x << ", y=" << y << ", z=" << z;
17     return 0;
18 }
```

- ▽ To gain access to its arguments, the function declares its parameters as references.
- ▽ In C++, references are indicated with an ampersand (&) following the parameter type.
- ▽ The parameters of the function becomes the aliases of the arguments passed by the calling function.
- ▽ On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for. The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant:

▽

```
1 string concatenate (const string& a, const string& b)
2 {
3     return a+b;
4 }
```

▷ INLINE FUNCTIONS

- ▽ Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.
- ▽ Preceding a function declaration with the `inline` specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

▽

```
1 inline string concatenate (const string& a, const string& b)
2 {
3     return a+b;
4 }
```

- ▽ `inline` is only specified in the function declaration, not when it is called.
- ▽ This specifier merely indicates the compiler that `inline` is preferred for this function, although the compiler is free to not inline it, and optimize otherwise.

▷ DEFAULT VALUES IN THE PARAMETERS

- ▽ In C++, functions can also have optional parameters, for which no arguments are required in the call, in such a way that, for example, a function with three parameters may be called with only two. For this, the function shall include a default value for its last parameter, which is used by the function when called with fewer arguments. For example:



```
1 // default values in functions
2 #include <iostream>
3 using namespace std;
4
5 int divide (int a, int b=2)
6 {
7     int r;
8     r=a/b;
9     return (r);
10}
11
12 int main ()
13 {
14     cout << divide (12) << '\n';
15     cout << divide (20,4) << '\n';
16     return 0;
17 }
```

▷ DECLARATION OF FUNCTIONS

▽ The parameter list does not need to include the parameter names, but only their types. Parameter names can nevertheless be specified, but they are optional, and do not need to necessarily match those in the function definition.



```
1 // declaring functions prototypes
2 #include <iostream>
3 using namespace std;
4
5 void odd (int x);
6 void even (int x);
7
8 int main()
9 {
10     int i;
11     do {
12         cout << "Please, enter number (0 to exit): ";
13         cin >> i;
14         odd (i);
15     } while (i!=0);
16     return 0;
17 }
18
19 void odd (int x)
20 {
21     if ((x%2)!=0) cout << "It is odd.\n";
22     else even (x);
23 }
24
25 void even (int x)
26 {
27     if ((x%2)==0) cout << "It is even.\n";
28     else odd (x);
29 }
```

Please, enter number (0 to exit): 9
It is odd.
Please, enter number (0 to exit): 6
It is even.
Please, enter number (0 to exit): 1030
It is even.
Please, enter number (0 to exit): 0
It is even.

OVERLOADS AND TEMPLATES

▷ OVERLOADED FUNCTIONS

▽ In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

```
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
    return (a*b);
}

double operate (double a, double b)
{
    return (a/b);
}

int main ()
{
    int x=5,y=2;
    double n=5.0,m=2.0;
    cout << operate (x,y) << '\n';
    cout << operate (n,m) << '\n';
    return 0;
}
```

- ▽ Two functions with the same name are generally expected to have - at least- a similar behavior.
- ▽ Note that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

▷ FUNCTION TEMPLATE

▽ Defining a function template follows the same syntax as a regular function, except that it is preceded by the template keyword and a series of template parameters enclosed in angle-brackets ◇:

template <template-parameters> function-declaration

▽ The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the class or typename keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

```
1 template <class SomeType>
2 SomeType sum (SomeType a, SomeType b)
3 {
4     return a+b;
5 }
```

- ▽ It makes no difference whether the generic type is specified with keyword `class` or keyword `typename` in the template argument list (they are 100% synonyms in template declarations).
- ▽ The generic type “`sometype`” can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.
- ▽ Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the function template, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:
`name <template-arguments> (function-arguments)`

```
x = sum<int>(10,20);
```

- ▽ When the generic type `T` is used as a parameter for a generic function, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets.
 Eg. `x=sum(10,20)` //`T` is deduced as `int` by compiler
`x=sum(10.5,20.5)` //`T` is deduced as `double` by compiler
- ▽ Templates are a powerful and versatile feature. They can have multiple template parameters, and the function can still use regular non-templated types. For example:

```

1 // function templates
2 #include <iostream>
3 using namespace std;
4
5 template <class T, class U>
6 bool are_equal (T a, U b)
7 {
8     return (a==b);
9 }
10
11 int main ()
12 {
13     if (are_equal(10,10.0))
14         cout << "x and y are equal\n";
15     else
16         cout << "x and y are not equal\n";
17     return 0;
18 }
```

x and y are equal

▷ NON TYPE TEMPLATE ARGUMENTS

- ▽ The template parameters can not only include types introduced by `class` or `typename`, but can also include expressions of a particular type:

NAME VISIBILITY

▷ NAMESAPCES

- ▽ But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.
- ▽ Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names.
- ▽ The syntax to declare a namespaces is:

```
namespace identifier
{
    named_entities
}
```

- ▽ Where identifier is any valid identifier and named_entities is the set of variables, types and functions that are included within the namespace. For example:

```
1 namespace myNamespace
2 {
3     int a, b;
4 }
```

- ▽ These variables can be accessed from within their namespace normally, with their identifier (either a or b), but if accessed from outside the myNamespace namespace they have to be properly qualified with the scope operator ::. For example, to access the previous variables from outside myNamespace they should be qualified like:

```
1 myNamespace::a
2 myNamespace::b
```

▽

```
1 // namespaces
2 #include <iostream>
3 using namespace std;
4
5 namespace foo
6 {
7     int value() { return 5; }
8 }
9
10 namespace bar
11 {
12     const double pi = 3.1416;
13     double value() { return 2*pi; }
14 }
15
16 int main () {
17     cout << foo::value() << '\n';
18     cout << bar::value() << '\n';
19     cout << bar::pi << '\n';
20     return 0;
21 }
```

5
6.2832
3.1416

- ▽ Namespaces can be split: Two segments of a code can be declared in the same namespace:

```
1 namespace foo { int a; }
2 namespace bar { int b; }
3 namespace foo { int c; }
```

▷ USING KEYWORD

- ▽ The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example :

```
1 // using
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8     int y = 10;
9 }
10
11 namespace second
12 {
13     double x = 3.1416;
14     double y = 2.7183;
15 }
16
17 int main () {
18     using first::x;
19     using second::y;
20     cout << x << '\n';
21     cout << y << '\n';
22     cout << first::y << '\n';
23     cout << second::x << '\n';
24     return 0;
25 }
```

5
2.7183
10
3.1416

- ▽ The keyword using can also be used as a directive to introduce an entire namespace:

```
1 // using
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8     int y = 10;
9 }
10
11 namespace second
12 {
13     double x = 3.1416;
14     double y = 2.7183;
15 }
16
17 int main () {
18     using namespace first;
19     cout << x << '\n';
20     cout << y << '\n';
21     cout << second::x << '\n';
22     cout << second::y << '\n';
23     return 0;
24 }
```

5
10
3.1416
2.7183

▽ `using` and `using namespace` have validity only in the same block in which they are stated or in the entire source code file if they are used directly in the global scope. For example, it would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks:

```
1 // using namespace example
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8 }
9
10 namespace second
11 {
12     double x = 3.1416;
13 }
14
15 int main () {
16 {
17     using namespace first;
18     cout << x << '\n';
19 }
20 {
21     using namespace second;
22     cout << x << '\n';
23 }
24 return 0;
25 }
```

5
3.1416

▷ NAMESPACE ALIASING

▽ Existing namespaces can be aliased with new names, with the following syntax:

```
namespace new_name = current_name;
```

▷ THE STD NAMESPACE

▽ All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the `std` namespace.

▽ Whether the elements in the `std` namespace are introduced with `using declarations` or are fully qualified on every use does not change the behavior or efficiency of the resulting program in any way. It is mostly a matter of style preference, although for projects mixing libraries, explicit qualification tends to be preferred.

▷ STORAGE CLASSES

▽ The storage for variables with global or namespace scope is allocated for the entire duration of the program. This is known as **static storage**, and it contrasts with the storage for local variables (those declared within a block). These use what is known as **automatic storage**. The storage for local variables is only available during the block in which they are declared; after that,

ARRAYS

- ▷ Declaration : type name [elements];
- ▷ The elements field within square brackets [], representing the number of elements in the array, must be a constant expression, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.
- ▷ Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).
- ▷ Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension.

▷ ARRAYS AS PARAMETERS

- ▽ In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument.
- ▽ To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
void procedure(int arg[])
```



```
1 // arrays as parameters
2 #include <iostream>
3 using namespace std;
4
5 void printarray (int arg[], int length) {
6     for (int n=0; n<length; ++n)
7         cout << arg[n] << ' ';
8     cout << '\n';
9 }
10
11 int main ()
12 {
13     int firstarray[] = {5, 10, 15};
14     int secondarray[] = {2, 4, 6, 8, 10};
15     printarray (firstarray,3);
16     printarray (secondarray,5);
17 }
```

5 10 15
2 4 6 8 10

⚙
Edit
&
Run

- ▽ In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

base_type[][][depth][depth]

Eg. void procedure(int myarray[][2][3])

- ▽ Notice that the first brackets [] are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.
- ▽ In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer.

▷ LIBRARY ARRAYS

- ▽ The arrays explained above are directly implemented as a language feature, inherited from the C language. They are a great feature, but by restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization.
- ▽ To overcome some of these issues with language built-in arrays, C++ provides an alternative array type as a standard container. It is a type template (a class template, in fact) defined in header <array>.
- ▽ They operate in a similar way to built-in arrays, except that they allow being copied (an actually expensive operation that copies the entire block of memory, and thus to use with care) and decay into pointers only when explicitly told to do so (by means of its member data).



language built-in array	container library array
<pre>#include <iostream> using namespace std; int main() { int myarray[3] = {10,20,30}; for (int i=0; i<3; ++i) ++myarray[i]; for (int elem : myarray) cout << elem << '\n'; }</pre>	<pre>#include <iostream> #include <array> using namespace std; int main() { array<int,3> myarray {10,20,30}; for (int i=0; i<myarray.size(); ++i) ++myarray[i]; for (int elem : myarray) cout << elem << '\n'; }</pre>

CHARACTER SEQUENCES

- ▷ Because strings are, in fact, sequences of characters, we can represent them also as plain arrays of elements of a character type.
- ▷ By convention, the end of strings represented in character sequences is signalled by a special character: the null character, whose literal value can be written as '\0' (backslash, zero).

▷ INITIALIZATION OF NULL TERMINATED CHARACTER SEQUENCES

- ▽ Sequences of characters enclosed in double-quotes ("") are literal constants. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

▽

```
1 char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 char myword[] = "Hello";
```

- ▽ Because string literals are regular arrays, they have the same restrictions as these, and cannot be assigned values. Expressions (once myword has already been declared as above), such as:

```
1 myword = "Bye";
2 myword[] = "Bye";
```

would **not** be valid, like neither would be:

```
myword = { 'B', 'y', 'e', '\0' },
```

This is because arrays cannot be assigned values. Note, though, that each of its elements can be assigned a value individually. For example, this would be correct:

```
1 myword[0] = 'B';
2 myword[1] = 'y';
3 myword[2] = 'e';
4 myword[3] = '\0';
```

▷ STRINGS AND NULL-TERMINATED CHARACTER SEQUENCES

- ▽ Plain arrays with null-terminated sequences of characters are the typical types used in the C language to represent strings (that is why they are also known as C-strings). In C++, even though the standard library defines a specific type for strings (class `string`), still, plain arrays with null-terminated sequences of characters (C-strings) are a natural way of representing strings

POINTERS

▷ The actual address of a variable in memory cannot be known before runtime.

▷ DECLARATION

▽ Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.

▽ The size in memory of a pointer depends on the platform where the program runs.

▷ POINTER & ARRAY

▽ The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int myarray[20];
int * mypointer;
mypointer=myarray;//valid statement
```

▽ The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything. Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot.



```
1 // more pointers
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int numbers[5];
8     int * p;
9     p = numbers; *p = 10;
10    p++; *p = 20;
11    p = &numbers[2]; *p = 30;
12    p = numbers + 3; *p = 40;
13    p = numbers; *(p+4) = 50;
14    for (int n=0; n<5; n++)
15        cout << numbers[n] << " ";
16    return 0;
17 }
```

10, 20, 30, 40, 50,

▽ [] brackets are a dereferencing operator known as offset operator. They dereference the variable they follow just as *

does, but they also add the number between brackets to the address being dereferenced.

▽

```
1 a[5] = 0;           // a [offset of 5] = 0
2 * (a+5) = 0;       // pointed to by (a+5) = 0
```

▷ POINTER ARITHMETIC

- ▽ Only addition and subtraction operations are allowed, They behave according to the size of the data type to which they point.
- ▽ Postfix operators, such as increment and decrement, have higher precedence than prefix operators, such as the dereference operator (*). Therefore, the following expression:
 $*p++$ is equivalent to $*(p++)$.
- ▽ Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
1 *p++    // same as *(p++): increment pointer, and dereference unincremented address
2 *++p    // same as *(++p): increment pointer, and dereference incremented address
3 ++*p    // same as ++(*p): dereference pointer, and increment the value it points to
4 (*p)++ // dereference pointer, and post-increment the value it points to
```

▽

```
*p++ = *q++;
```

Because $++$ has a higher precedence than $*$, both p and q are incremented, but because both increment operators ($++$) are used as postfix and not prefix, the value assigned to $*p$ is $*q$ before both p and q are incremented. And then both are incremented. It would be roughly equivalent to:

```
1 *p = *q;
2 ++p;
3 ++q;
```

▷ POINTER & CONSTANT

- ▽ Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough

with qualifying the type pointed to by the pointer as `const`. For example:



```
1 int x;
2 int y = 10;
3 const int * p = &y;
4 x = *p;           // ok: reading p
5 *p = x;          // error: modifying p, which is const-qualified
```

- ▽ Note also, that the expression `&y` is of type `int*`, but this is assigned to a pointer of type `const int*`. This is allowed: a pointer to non-`const` can be implicitly converted to a pointer to `const`. But not the other way around! As a safety feature, pointers to `const` are not implicitly convertible to pointers to non-`const`.
- ▽ One of the use cases of pointers to `const` elements is as function parameters: a function that takes a pointer to non-`const` as parameter can modify the value passed as argument, while a function that takes a pointer to `const` as parameter cannot.
- ▽ `const data_type * pointers` point to constant content they cannot modify, but they are not constant themselves: i.e., the pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.
- ▽ Pointers can also be themselves `const`. And this is specified by appending `const` to the pointed type (after the asterisk):

```
1 int x;
2     int *      p1 = &x;    // non-const pointer to non-const int
3 const int *      p2 = &x;    // non-const pointer to const int
4     int * const p3 = &x;    // const pointer to non-const int
5 const int * const p4 = &x;    // const pointer to const int
```

- ▽ The `const` qualifier can either precede or follow the pointed type, with the exact same meaning:

```
1 const int * p2a = &x;    //      non-const pointer to const int
2 int const * p2b = &x;    // also non-const pointer to const int
```

▷ POINTERS & STRING LITERALS

- ▽ String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type `const char` (as literals, they can never be modified). For example:

```
const char * foo = "hello";
```

▷ POINTERS TO POINTERS

- ▽ The syntax simply requires an asterisk (*) for each level of indirection in the declaration of the pointer:



```
1 char a;
2 char * b;
3 char ** c;
4 a = 'z';
5 b = &a;
6 c = &b;
```

▷ VOID POINTERS

- ▽ void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

- ▽ This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters.

- ▽ They have a great limitation: the data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

- ▽ One of its possible uses may be to pass generic parameters to a function. For example:

```
1 // increaser                                         y, 1603
2 #include <iostream>
3 using namespace std;
4
5 void increase (void* data, int psize)
6 {
7   if ( psize == sizeof(char) )
8     { char* pchar; pchar=(char*)data; ++(*pchar); }
9   else if (psize == sizeof(int) )
10    { int* pint; pint=(int*)data; ++(*pint); }
11 }
12
13 int main ()
14 {
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << '\n';
20   return 0;
21 }
```

▷ INVALID & NULL POINTERS

- ▽ Pointers can actually point to any address, including addresses that do not refer to any valid element. In contrast to pointing to a variable or to an element of an array.

- ▽ Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements of an array:

```
1 int * p;           // uninitialized pointer (local variable)
2
3 int myarray[10];
4 int * q = myarray+20; // element out of bounds
```

▽ Neither p nor q point to addresses known to contain a value, but none of the above statements causes an error. In C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not. What can cause an error is to dereference such a pointer (i.e., actually accessing the value they point to). Accessing such a pointer causes undefined behavior, ranging from an error during runtime to accessing some random value.

▽ But, sometimes, a pointer really needs to explicitly point to nowhere, and not just an invalid address. For such cases, there exists a special value that any pointer type can take: the null pointer value. This value can be expressed in C++ in two ways: either with an integer value of zero, or with the nullptr keyword:

```
1 int * p = 0;
2 int * q = nullptr;
```

- ▽ All null pointers compare equal to all other null pointers.
- ▽ Do not confuse null pointers with void pointers! A null pointer is a value that any pointer can take to represent that it is pointing to "nowhere", while a void pointer is a type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer, and the other to the type of data it points to.

▷ POINTERS TO FUNCTIONS

▽ C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```
1 // pointer to functions
2 #include <iostream>
3 using namespace std;
4
5 int addition (int a, int b)
6 { return (a+b); }
7
8 int subtraction (int a, int b)
9 { return (a-b); }
10
11 int operation (int x, int y, int (*functocall)(int,int))
12 {
13     int g;
14     g = (*functocall)(x,y);
15     return (g);
16 }
17
18 int main ()
19 {
20     int m,n;
21     int (*minus)(int,int) = subtraction;
22
23     m = operation (7, 5, addition);
24     n = operation (20, m, minus);
25     cout <<n;
26     return 0;
27 }
```

8

▽ In the example above, minus is a pointer to a function that has two parameters of type int. It is directly initialized to point to the function subtraction:

int (minus)(int,int) = subtraction;*

DYNAMIC MEMORY

▷ OPERATORS NEW AND NEW[]

▽ Returns the address of the beginning of the new block of memory allocated.

▽

```
pointer = new type  
pointer = new type [number_of_elements]
```

▽ The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

▽ C++ provides two standard mechanisms to check if the allocation was successful:

▶ One is by handling exceptions. Using this method, an exception of type `bad_alloc` is thrown when the allocation fails. If this exception is thrown and it is not handled by a specific handler, the program execution is terminated. This exception method is the method used by default by `new`.

▶ The other method is known as `nothrow`, and what happens when it is used is that when a memory allocation fails, instead of throwing a `bad_alloc` exception or terminating the program, the pointer returned by `new` is a *null pointer*, and the program continues its execution normally.

This method can be specified by using a special object called `nothrow`, declared in header `<new>`, as argument for `new`:

```
foo=new (nothrow) int[5];
```

▽ This `nothrow` method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations.

▷ OPERATIONS DELETE & DELETE[]

- ▽ The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

▷ DYNAMIC MEMORY IN C

- ▽ It uses functions malloc, calloc, realloc and free, defined in the header <cstdlib>.

- ▽ Malloc : `void* malloc (size_t size); //size_t is an unsigned integral type and size if the size of memory block`
 - ▶ Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.
 - ▶ Always returns a pointer of type void* and null pointer if allocation fails.
 - ▶ The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.
 - ▶ If size is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.

- ▽ Calloc : `void* calloc (size_t num, size_t size); //num is the no of elements to allocate, size if the size of each element, size_t is an unsigned integral type`

- ▶ Allocates a block of memory for an array of num elements, each of them size bytes long, and initializes all its bits to zero.
- ▶ On success, a pointer to the memory block allocated by the function.
- ▶ The type of this pointer is always void*, which can be cast to the desired type of data pointer in order to be dereferenceable.
- ▶ If the function failed to allocate the requested block of memory, a null pointer is returned.
- ▶ The effective result is the allocation of a zero-initialized memory block of (num*size) bytes.
- ▶ If size is zero, the return value depends on the particular library implementation (it may or may not be a null pointer), but the returned pointer shall not be dereferenced.

DATA STRUCTURES

▷ DATA STRUCTURES

▽ A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures can be declared in C++ using the following syntax:

```
struct type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    ...  
} object_names;
```

▽ Where `type_name` is a name for the structure type, `object_name` can be a set of valid identifiers for objects that have the type of this structure. Within braces {}, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

▽ Declaring Objects :



```
1 struct product {  
2     int weight;  
3     double price;  
4 } ;  
5  
6 product apple;  
7 product banana, melon;
```

```
1 struct product {  
2     int weight;  
3     double price;  
4 } apple, banana, melon;
```

▽ The members of an object can be accessed directly using a “.”.

▽ Because structures are types, they can also be used as the type of arrays to construct tables or databases of them.

▷ POINTERS TO STRUCTURES

▽ Structures can be pointed to by its own type of pointers.

▽ The arrow operator (`→`) is a dereference operator that is used exclusively with pointers to objects that have members. This operator serves to access the member of an object directly from its address.

▷ NESTING STRUCTURES

▽ Structures can also be nested in such a way that an element of a structure is itself another structure.

OTHER DATA TYPES

▷ TYPE ALIASES (TYPEDEF / USING)

▽ A type alias is a different name by which a type can be identified. In C++, any valid type can be aliased so that it can be referred to with a different identifier.

▽ Two ways :

► `typedef`(Inherited from c language)

`typedef existing_type new_type_name ;`

where existing type is any type, Either fundamental or compound.

► `using`(defined in c++)

`using new_type_name = existing_type ;`

▽ Both aliases defined with `typedef` and aliases defined with `using` are semantically equivalent. The only difference being that `typedef` has certain limitations in the realm of templates that `using` has not. Therefore, `using` is more generic, although `typedef` has a longer history and is probably more common in existing code.

▷ UNIONS

▽ Unions allow one portion of memory to be accessed as different data types. Its declaration and use is similar to the one of structures, but its functionality is totally different:

```
union type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

▽ This creates a new union type, identified by `type_name`, in which all its member elements occupy the same physical space in memory. The size of this type is the one of the largest member element.

For example:

```
1 union mytypes_t {  
2     char c;  
3     int i;  
4     float f;  
5 } mytypes;
```

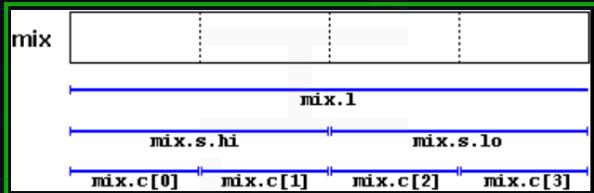
Each of the members(`c,i,f`) is of a different data type. But since all of them are referring to the same location in memory, the

modification of one of the members will affect the value of all of them. It is not possible to store different values in them in a way that each is independent of the others.

- ▽ One of the uses of a union is to be able to access a value either in its entirety or as an array or structure of smaller elements.

For example:

```
1 union mix_t {  
2     int l;  
3     struct {  
4         short hi;  
5         short lo;  
6     } s;  
7     char c[4];  
8 } mix;
```



- ▽ The exact alignment and order of the members of a union in memory depends on the system, with the possibility of creating portability issues.

▷ ANONYMOUS UNIONS

- ▽ When unions are members of a class (or structure), they can be declared with no name. In this case, they become anonymous unions, and its members are directly accessible from objects by their member names. For example, see the differences between these two structure declarations:

structure with regular union	structure with anonymous union
struct book1_t { char title[50]; char author[50]; union { float dollars; int yen; } price; } book1;	struct book2_t { char title[50]; char author[50]; union { float dollars; int yen; }; } book2;

- ▽ Accessing :

```
1 book1.price.dollars  
2 book1.price.yen
```

```
1 book2.dollars  
2 book2.yen
```

▷ ENUMERATED TYPES (ENUM)

▽ Enumerated types are types that are defined with a set of custom identifiers, known as enumerators, as possible values. Objects of these enumerated types can take any of these enumerators as value.

▽ Syntax :

```
enum type_name {  
    value1,  
    value2,  
    value3,  
    :  
} object_names;
```

▽ This creates the type `type_name`, which can take any of `value1`, `value2`, `value3`, ... as value. Objects (variables) of this type can directly be instantiated as `object_names`.

▽ Eg.

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

This declaration includes no other type, neither fundamental nor compound, in its definition. To say it another way, somehow, this creates a whole new data type from scratch without basing it on any other existing type. The possible values that variables of this new type `color_t` may take are the enumerators listed within braces.

```
1 colors_t mycolor;  
2  
3 mycolor = blue;  
4 if (mycolor == green) mycolor = red;
```

▽ Values of enumerated types declared with `enum` are implicitly convertible to an integer type. In fact, the elements of such an `enum` are always assigned an integer numerical equivalent internally, to which they can be implicitly converted to. If it is not specified otherwise, the integer value equivalent to the first possible value is 0, the equivalent to the second is 1, to the third is 2, and so on... Therefore, in the data type `colors_t` defined above, `black` would be equivalent to 0, `blue` would be equivalent to 1, `green` to 2, and so on...

▽ A specific integer value can be specified for any of the possible values in the enumerated type. And if the constant value that follows it is itself not given its own value, it is automatically assumed to be the same value plus one. For example:

▽ Here january is represented by 1 and december by 12.

```
1 enum months_t { january=1, february, march, april,
2                               may, june, july, august,
3                               september, october, november, december} y2k;
```

▷ ENUMERATED TYPES WITH ENUM CLASSES

▽ In C++, it is possible to create real enum types that are neither implicitly convertible to int and that neither have enumerator values of type int, but of the enum type itself, thus preserving type safety. They are declared with enum class (or enum struct) instead of just enum:

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```

Each of the enumerator values of an enum class type needs to be scoped into its type (this is actually also possible with enum types, but it is only optional). For example:

```
1 Colors mycolor;
2
3 mycolor = Colors::blue;
4 if (mycolor == Colors::green) mycolor = Colors::red;
```

▽ Enumerated types declared with enum class also have more control over their underlying type; it may be any integral data type, such as char, short or unsigned int, which essentially serves to determine the size of the type. This is specified by a colon and the underlying type following the enumerated type. For example:

```
enum class EyeColor : char {blue, green, brown};
```

Here, EyeColor is a distinct type with the same size of a char (1 byte).

CLASSES - I

- ▷ Like data structures, they can contain data members, but they can also contain functions as members.
- ▷ Classes are defined using either keyword `class` or keyword `struct`, with the following syntax:

```
class class_name {  
    access_specifier_1:  
        member1;  
    access_specifier_2:  
        member2;  
    ...  
} object_names;
```

- ▷ The body of the declaration can contain members, which can either be data or function declarations, and optionally access specifiers.
- ▷ An access specifier is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights for the members that follow them:
 - ▽ `Private` : members of a class are accessible only from within other members of the same class (or from their "friends").
 - ▽ `Protected` : members are accessible from other members of the same class (or from their "friends"), but also from members of their derived classes.
 - ▽ `Public` : members are accessible from anywhere where the object is visible.

- ▷ By default, all members of a class declared with the `class` keyword have private access for all its members.

- ▷ Eg.

```
1 // classes example  
2 #include <iostream>  
3 using namespace std;  
4  
5 class Rectangle {  
6     int width, height;  
7 public:  
8     void set_values (int,int);  
9     int area() {return width*height;}  
10 },  
11  
12 void Rectangle::set_values (int x, int y) {  
13     width = x;  
14     height = y;  
15 }  
16  
17 int main () {  
18     Rectangle rect;  
19     rect.set_values (3,4);  
20     cout << "area: " << rect.area();  
21     return 0;  
22 }
```

- ▷ Here the scope resolution operator is used in the definition of function to define a member of a class outside the class itself.
- ▷ The only difference between defining a member function completely within the class definition or to just include its declaration in the function and define it later outside the class, is that in the first case the function is automatically considered an inline member function by the compiler, while in the second it is a normal (not-inline) class member function. This causes no differences in behavior, but only on possible compiler optimizations.
- ▷ Classes allow programming using object-oriented paradigms: Data and functions are both members of the object, reducing the need to pass and carry handlers or other state variables as arguments to functions, because they are part of the object whose member is called.

- ▷ CONSTRUCTORS

- ▽ A class can include a special function called its constructor, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.
- ▽ This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.

```

1 // example: class constructor
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7 public:
8     Rectangle (int,int);
9     int area () {return (width*height);}
10 };
11
12 Rectangle::Rectangle (int a, int b) {
13     width = a;
14     height = b;
15 }
16
17 int main () {
18     Rectangle rect (3,4);
19     Rectangle rectb (5,6);
20     cout << "rect area: " << rect.area() << endl;
21     cout << "rectb area: " << rectb.area() << endl;
22     return 0;
23 }
```

rect area: 12
rectb area: 30



- ▽ Constructors cannot be called explicitly as if they were regular member functions. They are only executed once, when a new object of that class is created.

▽ Constructors never return values, they simply initialize the object.

▷ OVERLOADING CONSTRUCTORS

▽ A constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types.



```
1 // overloading class constructors
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7 public:
8     Rectangle ();
9     Rectangle (int,int);
10    int area (void) {return (width*height);}
11 };
12
13 Rectangle::Rectangle () {
14     width = 5;
15     height = 5;
16 }
17
18 Rectangle::Rectangle (int a, int b) {
19     width = a;
20     height = b;
21 }
22
23 int main () {
24     Rectangle rect (3,4);
25     Rectangle rectb;
26     cout << "rect area: " << rect.area() << endl;
27     cout << "rectb area: " << rectb.area() << endl;
28     return 0;
29 }
```

rect area: 12
rectb area: 25

*
Edit
&
Run

▽ The default constructor : The default constructor is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments. Empty parentheses cannot be used to call the default constructor. This is because the empty set of parentheses would make of rectc a function declaration instead of an object declaration: It would be a function that takes no arguments and returns a value of type Rectangle.



```
1 Rectangle rectb; // ok, default constructor called
2 Rectangle rectc(); // oops, default constructor NOT called
```

▷ UNIFORM INITIALIZATION

▽ The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as functional form. But constructors can also be called with other syntaxes:

- ▶ First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument): `class_name object_name = initialization_value;`
- ▶ More recently, C++ introduced the possibility of constructors to be called using uniform initialization, which essentially is the

same as the functional form, but using braces ({}) instead of parentheses (()): class_name object_name { value, value, ... } or class_name object_name = {value, value, ... }

▽ Different ways of initialization :

```
1 // classes and uniform initialization
2 #include <iostream>
3 using namespace std;
4
5 class Circle {
6     double radius;
7 public:
8     Circle(double r) { radius = r; }
9     double circum() {return 2*radius*3.14159265;}
10 };
11
12 int main () {
13     Circle foo (10.0);    // functional form
14     Circle bar = 20.0;   // assignment init.
15     Circle baz {30.0};  // uniform init.
16     Circle qux = {40.0}; // POD-like
17
18     cout << "foo's circumference: " << foo.circum() << '\n';
19     return 0;
20 }
```

Edit & Run

▽ An advantage of uniform initialization over functional form is that, unlike parentheses, braces cannot be confused with function declarations, and thus can be used to explicitly call default constructors.

```
1 Rectangle rectb; // default constructor called
2 Rectangle rectc(); // function declaration (default constructor NOT called)
3 Rectangle rectd{}; // default constructor called
```

▽ Uniform Initialization has preference for initializer_list as its type.

Initializer List

This type is used to access the values in a C++ initialization list, which is a list of elements of type const T.

Objects of this type are automatically constructed by the compiler from initialization list declarations, which is a list of comma-separated elements enclosed in braces:

```
auto il = { 10, 20, 30 }; // the type of il is an initializer_list
```

Notice though that this template class is not implicitly defined and the header <initializer_list> shall be included to access it, even if the type is used implicitly.

initializer_list objects are automatically constructed as if an array of elements of type T was allocated, with each of the elements in the list being copy-initialized to its corresponding element in the array, using any necessary non-narrowing implicit conversions.

The initializer_list object refers to the elements of this array without containing them: copying an initializer_list object produces another object referring to the same underlying elements, not to new copies of them (reference semantics).

The lifetime of this temporary array is the same as the initializer_list object.

Constructors taking only one argument of this type are a special kind of constructor, called *initializer-list constructor*. Initializer-list constructors take precedence over other constructors when the initializer-list constructor syntax is used:

```
1 struct myclass {
2     myclass (int,int);
3     myclass (initializer_list<int>);
4     /* definitions ... */
5 };
6
7 myclass foo {10,20}; // calls initializer_list ctor
8 myclass bar (10,20); // calls first constructor
```

▷ MEMBER INITIALIZATION IN CONSTRUCTORS

- ▽ When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members.

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) {}
```

- ▽ For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default, but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed.
- ▽ Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor), but in some other cases, default-construction is not even possible (when the class does not have a default constructor). In these cases, members shall be initialized in the member initialization list.

For example:

```
1 // member initialization
2 #include <iostream>
3 using namespace std;
4
5 class Circle {
6     double radius;
7 public:
8     Circle(double r) : radius(r) {}
9     double area() {return radius*radius*3.14159265;}
10 };
11
12 class Cylinder {
13     Circle base;
14     double height;
15 public:
16     Cylinder(double r, double h) : base (r), height(h) {}
17     double volume() {return base.area() * height;}
18 };
19
20 int main () {
21     Cylinder foo (10,20);
22
23     cout << "foo's volume: " << foo.volume() << '\n';
24     return 0;
25 }
```

foo's volume: 6283.19

- ▽ In this example, class Cylinder has a member object whose type is another class (base's type is Circle). Because objects of class Circle can only be constructed with a parameter, Cylinder's constructor needs to call base's constructor, and the only way to do this is in the member initializer list.

- ▽ These initializations can also use uniform initializer syntax, using braces {} instead of parentheses ():

```
Cylinder::Cylinder (double r, double h) : base{r}, height{h} { }
```

▷ POINTERS TO CLASSES

- ▽ Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer. For example:

`Rectangle * prect;`

is a pointer to an object of class Rectangle.

▷ INTERPRETATIONS

▽

expression	can be read as
<code>*x</code>	pointed to by x
<code>&x</code>	address of x
<code>x.y</code>	member y of object x
<code>x->y</code>	member y of object pointed to by x
<code>(*x).y</code>	member y of object pointed to by x (equivalent to the previous one)
<code>x[0]</code>	first object pointed to by x
<code>x[1]</code>	second object pointed to by x
<code>x[n]</code>	(n+1)th object pointed to by x

▷ CLASSED DEFINED WITH STRUCT & UNION KEYWORDS

- ▽ Classes can be defined not only with keyword `class`, but also with keywords `struct` and `union`.

- ▽ The keyword `struct`, generally used to declare plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword `class`. The only difference between both is that members of classes declared with the keyword `struct` have public access by default, while members of classes declared with the keyword `class` have private access by default. For all other purposes both keywords are equivalent in this context.

- ▽ Conversely, the concept of unions is different from that of classes declared with `struct` and `class`, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold member functions. The default access in union classes is public.

CLASSES - II

▷ OVERLOADING OPERATORS

▽ C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes. Here is a list of all the operators that can be overloaded:

Overloadable operators													
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>	
<=>	==	!=	<=	>=	++	--	%	&	^	!			
~	&=	^=	=	&&		%=	[]	()	,	->*	->	new	
delete	new[]	delete[]											

▽ Operators are overloaded by means of operator functions, which are regular functions with special names: their name begins by the operator keyword followed by the operator sign that is overloaded.

The syntax is:

```
type operator sign (parameters) { /*... body ...*/ }
```

▽ Eg. Overloading + operator for cartesian vectors

```
1 // overloading operators example
2 #include <iostream>
3 using namespace std;
4
5 class CVector {
6 public:
7     int x,y;
8     CVector () {};
9     CVector (int a,int b) : x(a), y(b) {}
10    CVector operator + (const CVector&);
11 };
12
13 CVector CVector::operator+ (const CVector& param) {
14     CVector temp;
15     temp.x = x + param.x;
16     temp.y = y + param.y;
17     return temp;
18 }
19
20 int main () {
21     CVector foo (3,1);
22     CVector bar (1,2);
23     CVector result;
24     result = foo + bar;
25     cout << result.x << ',' << result.y << '\n';
26     return 0;
27 }
```

▽ The function operator+ of class CVector overloads the addition operator (+) for that type. Once declared, this function can be called either implicitly using the operator, or explicitly using its functional name:

```
c = a + b;
```

```
c = a.operator+ (b);
```

- ▽ The parameter expected for a member function overload for operations such as operator+ is naturally the operand to the right hand side of the operator. This is common to all binary operators (those with an operand to its left and one operand to its right).
- ▽ Summary of the parameters needed for each of the different operators than can be overloaded (please, replace @ by the operator in each case):

Expression	Operator	Member function	Non-member function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & < > == != <= >= << >> && ,	A::operator@(B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= = <=>= []	A::operator@(B)	-
a(b,c...)	()	A::operator()(B,C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE()	-

Where a is an object of class A, b is an object of class B and c is an object of class C. TYPE is just any type (that operators overloads the conversion to type TYPE).

- ▽ Some operators may be overloaded in two forms: either as a member function or as a non-member function.
- ▽ When any operator is overloaded as non-member functions; In this case, the operator function takes an object of the proper class as first argument.

```

1 // non-member operator overloads
2 #include <iostream>
3 using namespace std;
4
5 class CVector {
6 public:
7     int x,y;
8     CVector () {}
9     CVector (int a, int b) : x(a), y(b) {}
10 };
11
12
13 CVector operator+ (const CVector& lhs, const CVector& rhs) {
14     CVector temp;
15     temp.x = lhs.x + rhs.x;
16     temp.y = lhs.y + rhs.y;
17     return temp;
18 }
19
20 int main () {
21     CVector foo (3,1);
22     CVector bar (1,2);
23     CVector result;
24     result = foo + bar;
25     cout << result.x << ',' << result.y << '\n';
26     return 0;
27 }
```

▷ THE KEYWORD THIS

- ▽ The keyword this represents a pointer to the object whose member function is being executed. It is used within a class's member function to refer to the object itself.
- ▽ One of its uses can be to check if a parameter passed to a member function is the object itself. For example:

```
1 // example on this
2 #include <iostream>
3 using namespace std;
4
5 class Dummy {
6 public:
7     bool isitme (Dummy& param);
8 };
9
10 bool Dummy::isitme (Dummy& param)
11 {
12     if (&param == this) return true;
13     else return false;
14 }
15
16 int main () {
17     Dummy a;
18     Dummy* b = &a;
19     if ( b->isitme(a) )
20         cout << "yes, &a is b\n";
21     return 0;
22 }
```

yes, &a is b

- ▽ It is also frequently used in operator overloaded member functions that return objects by reference. Following with the examples on cartesian vector seen before, its operator= function could have been defined as:

```
1 CVector& CVector::operator= (const CVector& param)
2 {
3     x=param.x;
4     y=param.y;
5     return *this;
6 }
```

▷ STATIC MEMBERS

- ▽ A class can contain static members, either data or functions.
- ▽ A static data member of a class is also known as a "class variable", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., its value is not different from one object of this class to another.
- ▽ It may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```

1 // static members in classes
2 #include <iostream>
3 using namespace std;
4
5 class Dummy {
6     public:
7         static int n;
8         Dummy () { n++; }
9     };
10
11 int Dummy::n=0;
12
13 int main () {
14     Dummy a;
15     Dummy b[5];
16     cout << a.n << '\n';
17     Dummy * c = new Dummy;
18     cout << Dummy::n << '\n';
19     delete c;
20     return 0;
21 }
```

6
7

▽ In fact, static members have the same properties as non-member variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, they cannot be initialized directly in the class, but need to be initialized somewhere outside it. As in the previous example:

int Dummy::n=0;

▽ Because it is a common variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

*cout << a.n;
cout << Dummy::n;*

▽ Classes can also have static member functions. These represent the same: members of a class that are common to all object of that class, acting exactly as non-member functions but being accessed like members of the class. Because they are like non-member functions, they cannot access non-static members of the class (neither member variables nor member functions). They neither can use the keyword *this*.

▷ CONST MEMBER FUNCTIONS

▽ When an object of a class is qualified as a const object:

const MyClass myobject;

The access to its data members from outside the class is restricted to read-only, as if all its data members were *const* for those accessing them from outside the class. Note though, that the


```

1 // const objects
2 #include <iostream>
3 using namespace std;
4
5 class MyClass {
6     int x;
7 public:
8     MyClass(int val) : x(val) {}
9     const int& get() const {return x;}
10 };
11
12 void print (const MyClass& arg) {
13     cout << arg.get() << '\n';
14 }
15
16 int main() {
17     MyClass foo (10);
18     print(foo);
19
20     return 0;
21 }
```

10

▽ Member functions can be overloaded on their constness: i.e., a class may have two member functions with identical signatures except that one is const and the other is not: in this case, the const version is called only when the object is itself const, and the non-const version is called when the object is itself non-const.

▽

```

1 // overloading members on constness
2 #include <iostream>
3 using namespace std;
4
5 class MyClass {
6     int x;
7 public:
8     MyClass(int val) : x(val) {}
9     const int& get() const {return x;}
10    int& get() {return x;}
11 };
12
13 int main() {
14     MyClass foo (10);
15     const MyClass bar (20);
16     foo.get() = 15;           // ok: get() returns int&
17 // bar.get() = 25;          // not valid: get() returns const int&
18     cout << foo.get() << '\n';
19     cout << bar.get() << '\n';
20
21     return 0;
22 }
```

15

20

▷ CLASS TEMPLATES

▽ Defination :

```

1 template <class T>
2 class mypair {
3     T values [2];
4 public:
5     mypair (T first, T second)
6     {
7         values[0]=first; values[1]=second;
8     }
9 };
```

▽ Call :

```
mpair<int> myobject (115, 36);
```

▽ In case that a member function is defined outside the defintion of the class template, it shall be preceded with the template <...> prefix:

▽ Syntax :

```
template <> class mycontainer <char> { ... };
```

▽ The empty `<>` signifies that all types are known and no template arguments are required for this specialization.

The `<char>` specialization parameter itself identifies the type for which the template class is being specialized (`char`).

▽

```
1 template <class T> class mycontainer { ... };  
2 template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

▽ When we declare specializations for a template class, we must also define all its members, even those identical to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

SPECIAL MEMBERS

- ▷ Special member functions are member functions that are implicitly defined as member of classes under certain circumstances. There are six:

Member function	typical form for class C:
Default constructor	C::C();
Destructor	C::~C();
Copy constructor	C::C (const C&);
Copy assignment	C& operator= (const C&);
Move constructor	C::C (C&&);
Move assignment	C& operator= (C&&);

- ▷ DEFAULT CONSTRUCTOR

- ▽ The default constructor is the constructor called when objects of a class are declared, but are not initialized with any arguments.
- ▽ If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor.

Therefore, after declaring a class like this:

```
1 class Example {
2     public:
3     int total;
4     void accumulate (int x) { total += x; }
5 };
```

The compiler assumes that this class has a default constructor.

Declaring objects of this class :

Example ex;

- ▽ But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor, and no longer allows the declaration of new objects of that class without arguments.

```
1 class Example2 {
2     public:
3     int total;
4     Example2 (int initial_value) : total(initial_value) { };
5     void accumulate (int x) { total += x; };
6 };
```

Declaring objects of this class :

Example ex{100}; //valid

Example ex; //Invalid

Therefore, if objects of this class need to be constructed without arguments, the proper default constructor also needs to be declared in the class.



```
1 // classes and default constructors                                bar's content: Example
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 class Example3 {
7     string data;
8 public:
9     Example3 (const string& str) : data(str) {}
10    Example3() {}
11    const string& content() const {return data;}
12 };
13
14 int main () {
15     Example3 foo;
16     Example3 bar ("Example");
17
18     cout << "bar's content: " << bar.content() << '\n';
19     return 0;
20 }
```

▷ DESTRUCTOR

- ▽ They are responsible for the necessary cleanup needed by a class when its lifetime ends (If there is dynamic allocation).
- ▽ it would be very useful to have a function called automatically at the end of the object's life in charge of releasing the allocated dynamic memory. To do this, we use a destructor. A destructor is a member function very similar to a default constructor: it takes no arguments and returns nothing, not even void. It also uses the class name as its own name, but preceded with a tilde sign (~).

```
1 // destructors                                         bar's content: Example
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 class Example4 {
7     string* ptr;
8 public:
9     // constructors:
10    Example4() : ptr(new string) {}
11    Example4 (const string& str) : ptr(new string(str)) {}
12    // destructor:
13    ~Example4 () {delete ptr;}
14    // access content:
15    const string& content() const {return *ptr;}
16 };
17
18 int main () {
19     Example4 foo;
20     Example4 bar ("Example");
21
22     cout << "bar's content: " << bar.content() << '\n';
23     return 0;
24 }
```

▷ COPY CONSTRUCTOR

- ▽ When an object is passed a named object of its own type as argument, its copy constructor is invoked in order to construct a copy.
- ▽ A copy constructor is a constructor whose first parameter is of type reference to the class itself (possibly const qualified) and which can be invoked with a single argument of this type. For

example, for a class `MyClass`, the copy constructor may have the following signature:

```
MyClass::MyClass (const MyClass&);
```

- ▽ If a class has no custom copy nor move constructors (or assignments) defined, an implicit copy constructor is provided. This copy constructor simply performs a copy of its own members.

For example, for a class such as:

```
1 class MyClass {  
2     public:  
3         int a, b; string c;  
4 };
```

An implicit copy constructor is automatically defined. The definition assumed for this function performs a shallow copy, roughly equivalent to:

```
MyClass::MyClass(const MyClass& x) : a(x.a), b(x.b), c(x.c) {}
```

- ▽ shallow copies only copy the members of the class themselves.
- ▽ For classes that contains pointers of which it handles its storage. For that class, performing a shallow copy means that the pointer value is copied, but not the content itself; This means that both objects (the copy and the original) would be sharing a single string object (they would both be pointing to the same object), and at some point (on destruction) both objects would try to delete the same block of memory, probably causing the program to crash on runtime. This can be solved by defining the following custom *copy constructor* that performs a deep copy:

```
1 // copy constructor: deep copy  
2 #include <iostream>  
3 #include <string>  
4 using namespace std;  
5  
6 class Example5 {  
7     string* ptr;  
8 public:  
9     Example5 (const string& str) : ptr(new string(str)) {}  
10    ~Example5 () {delete ptr;}  
11    // copy constructor:  
12    Example5 (const Example5& x) : ptr(new string(x.content())) {}  
13    // access content:  
14    const string& content() const {return *ptr;}  
15 };  
16  
17 int main () {  
18     Example5 foo ("Example");  
19     Example5 bar = foo;  
20  
21     cout << "bar's content: " << bar.content() << '\n';  
22     return 0;  
23 }
```

bar's content: Example

▷ COPY ASSIGNMENT

- ▽ Objects are not only copied on construction, when they are initialized: They can also be copied on any assignment operation. See the difference:

```
1 MyClass foo;
2 MyClass bar (foo);      // object initialization: copy constructor called
3 MyClass baz = foo;      // object initialization: copy constructor called
4 foo = bar;              // object already initialized: copy assignment called
```

- ▽ The copy assignment operator is an overload of operator= which takes a value or reference of the class itself as parameter. The return value is generally a reference to *this (although this is not required). For example, for a class MyClass, the copy assignment may have the following signature:

`MyClass& operator= (const MyClass&);`

- ▽ The copy assignment operator is also a special function and is also defined implicitly if a class has no custom copy nor move assignments (nor move constructor) defined.

- ▽ The implicit version performs a shallow copy which is not suitable for classes with pointers to objects they handle its storage.

- ▽ Not only the class incurs the risk of deleting the pointed object twice, but the assignment creates memory leaks by not deleting the object pointed by the object before the assignment. These issues could be solved with a copy assignment that deletes the previous object and performs a deep copy:

```
1 Example5& operator= (const Example5& x) {
2     delete ptr;           // delete currently pointed string
3     ptr = new string (x.content()); // allocate space for new string, and copy
4     return *this;
5 }
6
```

▷ MOVE CONSTRUCTOR & ASSINGMENT

- ▽ Unlike copying, the content is actually transferred from one object (the source) to the other (the destination): the source loses that content, which is taken over by the destination. This moving only happens when the source of the value is an unnamed object.

▽ Unnamed objects are objects that are temporary in nature, and thus haven't even been given a name. Typical examples of unnamed objects are return values of functions or type-casts.

▽ The move constructor is called when an object is initialized on construction using an unnamed temporary. Likewise, the move assignment is called when an object is assigned the value of an unnamed temporary:

```
1 MyClass fn();           // function returning a MyClass object
2 MyClass foo;            // default constructor
3 MyClass bar = foo;      // copy constructor
4 MyClass baz = fn();     // move constructor
5 foo = bar;              // copy assignment
6 baz = MyClass();         // move assignment
```

▽ Both the value returned by fn and the value constructed with MyClass are unnamed temporaries. In these cases, there is no need to make a copy, because the unnamed object is very short-lived and can be acquired by the other object when this is a more efficient operation.

▽ The move constructor and move assignment are members that take a parameter of type rvalue reference to the class itself:

```
1 MyClass (MyClass&&);        // move-constructor
2 MyClass& operator= (MyClass&&); // move-assignment
```

▽ An rvalue reference is specified by following the type with two ampersands (&&). As a parameter, an rvalue reference matches arguments of temporaries of this type.

▽ The concept of moving is most useful for objects that manage the storage they use, such as objects that allocate storage with new and delete. In such objects, copying and moving are really different operations:

- Copying from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B.
- Moving from A to B means that the memory already allocated to A is transferred to B without allocating any new storage.



```
1 // move constructor/assignment
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 class Example6 {
7     string* ptr;
8 public:
9     Example6 (const string& str) : ptr(new string(str)) {}
10    ~Example6 () {delete ptr;}
11    // move constructor
12    Example6 (Example6&& x) : ptr(x.ptr) {x.ptr=nullptr;}
13    // move assignment
14    Example6& operator= (Example6&& x) {
15        delete ptr;
16        ptr = x.ptr;
17        x.ptr=nullptr;
18        return *this;
19    }
20    // access content:
21    const string& content() const {return *ptr;}
22    // addition:
23    Example6 operator+(const Example6& rhs) {
24        return Example6(content()+rhs.content());
25    }
26 };
27
28
29 int main () {
30     Example6 foo ("Exam");
31     Example6 bar = Example6("ple"); // move-construction
32
33     foo = foo + bar; // move-assignment
34
35     cout << "foo's content: " << foo.content() << '\n';
36     return 0;
37 }
```

▽ Compilers already optimize many cases that formally require a move-construction call in what is known as Return Value Optimization. Most notably, when the value returned by a function is used to initialize an object. In these cases, the move constructor may actually never get called.

▷ IMPLICIT MEMBERS

▽ The six special members functions described above are members implicitly declared on classes under certain circumstances:

Member function	implicitly defined:	default definition:
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy constructor and no copy nor move assignment	moves all members

▽ Each class can select explicitly which of these members exist with their default definition or which are deleted by using the keywords `default` and `delete`, respectively. The syntax is either one of:

```
function_declaration = default;
function_declaration = delete;
```


FRIENDSHIP AND INHERITANCE

▷ FRIEND FUNCTION

- ▽ In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends".
- ▽ Friends are functions or classes declared with the friend keyword.



```
1 // friend functions
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7 public:
8     Rectangle() {}
9     Rectangle (int x, int y) : width(x), height(y) {}
10    int area() {return width * height;}
11    friend Rectangle duplicate (const Rectangle&);
12 };
13
14 Rectangle duplicate (const Rectangle& param)
15 {
16     Rectangle res;
17     res.width = param.width*2;
18     res.height = param.height*2;
19     return res;
20 }
21
22 int main () {
23     Rectangle foo;
24     Rectangle bar (2,3);
25     foo = duplicate (bar);
26     cout << foo.area() << '\n';
27     return 0;
28 }
```

24

- ▽ Function `duplicate` is not considered a member of class `Rectangle`. It simply has access to its private and protected members without being a member.
- ▽ Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.

▷ FRIEND CLASSES



```
1 // friend class
2 #include <iostream>
3 using namespace std;
4
5 class Square;
6
7 class Rectangle {
8     int width, height;
9 public:
10    int area () {
11        return (width * height); }
12    void convert (Square a);
13 };
14
15 class Square {
16     friend class Rectangle;
17 private:
18     int side;
19 public:
20     Square (int a) : side(a) {}
21 };
22
23 void Rectangle::convert (Square a) {
24     width = a.side;
25     height = a.side;
26 }
27
28 int main () {
29     Rectangle rect;
30     Square sqr (4);
31     rect.convert(sqr);
32     cout << rect.area();
33     return 0;
34 }
```

16

- ▽ Empty declaration of class Square is necessary because class Rectangle uses Square (as a parameter in member convert), and Square uses Rectangle (declaring it a friend).
- ▽ Friendships are never corresponded unless specified: In our example, Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle. Therefore, the member functions of Rectangle can access the protected and private members of Square but not the other way around.
- ▽ Another property of friendships is that they are not transitive: The friend of a friend is not considered a friend unless explicitly specified.

▷ INHERITANCE BETWEEN CLASSES

- ▽ Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a base class and a derived class: The derived class inherits the members of the base class, on top of which it can add its own members.

▽ The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:

```
class derived_class_name: public base_class_name  
{ /* ... */ };
```

▽ The public access specifier may be replaced by any one of the other access specifiers (protected or private). This access specifier limits the most accessible level for the members inherited from the base class: The members with a less restrictive level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

▽

```
1 // derived classes  
2 #include <iostream>  
3 using namespace std;  
4  
5 class Polygon {  
6     protected:  
7         int width, height;  
8     public:  
9         void set_values (int a, int b)  
10            { width=a; height=b; }  
11    };  
12  
13 class Rectangle: public Polygon {  
14     public:  
15         int area ()  
16            { return width * height; }  
17    };  
18  
19 class Triangle: public Polygon {  
20     public:  
21         int area ()  
22            { return width * height / 2; }  
23    };  
24  
25 int main () {  
26     Rectangle rect;  
27     Triangle trgl;  
28     rect.set_values (4,5);  
29     trgl.set_values (4,5);  
30     cout << rect.area() << '\n';  
31     cout << trgl.area() << '\n';  
32     return 0;  
33 }
```

20
10

▽ Access :

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

▽ If no access level is specified for the inheritance, the compiler assumes private for classes declared with keyword class and public for those declared with struct.

▷ WHAT IS INHERITED FROM THE BASE CLASS?

- ▽ In principle, a publicly derived class inherits access to every member of a base class except:
 - its constructors and its destructor
 - its assignment operator members (`operator=`)
 - friends
 - its private members
- ▽ Even though access to the constructors and destructor of the base class is not inherited as such, they are automatically called by the constructors and destructor of the derived class
- ▽ Unless otherwise specified, the constructors of a derived class calls the default constructor of its base classes (i.e., the constructor taking no arguments). Calling a different constructor of a base class is possible, using the same syntax used to initialize member variables in the initialization list:
`derived_constructor_name (parameters) : base_constructor_name
(parameters) {...}`



```
1 // constructors and derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Mother {
6 public:
7     Mother ()
8     { cout << "Mother: no parameters\n"; }
9     Mother (int a)
10    { cout << "Mother: int parameter\n"; }
11 };
12
13 class Daughter : public Mother {
14 public:
15     Daughter (int a)
16     { cout << "Daughter: int parameter\n\n"; }
17 };
18
19 class Son : public Mother {
20 public:
21     Son (int a) : Mother (a)
22     { cout << "Son: int parameter\n\n"; }
23 };
24
25 int main () {
26     Daughter kelly(0);
27     Son bud(0);
28
29     return 0;
30 }
```

Mother: no parameters
Daughter: int parameter

Mother: int parameter
Son: int parameter

▷ MULTIPLE INHERITANCE

- ▽ A class may inherit from more than one class by simply specifying more base classes, separated by commas, in the list of a class's base classes (i.e., after the colon).

POLYMORPHISM

▷ POINTER TO BASE CLASS

▽ One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.



```
1 // pointers to base class
2 #include <iostream>
3 using namespace std;
4
5 class Polygon {
6 protected:
7     int width, height;
8 public:
9     void set_values (int a, int b)
10    { width=a; height=b; }
11 };
12
13 class Rectangle: public Polygon {
14 public:
15     int area()
16     { return width*height; }
17 };
18
19 class Triangle: public Polygon {
20 public:
21     int area()
22     { return width*height/2; }
23 };
24
25 int main () {
26     Rectangle rect;
27     Triangle trgl;
28     Polygon * ppoly1 = &rect;
29     Polygon * ppoly2 = &trgl;
30     ppoly1->set_values (4,5);
31     ppoly2->set_values (4,5);
32     cout << rect.area() << '\n';
33     cout << trgl.area() << '\n';
34     return 0;
35 }
```

20
10

▽ But because the type of both ppoly1 and ppoly2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.

▷ VIRTUAL MEMBERS

▽ A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the **virtual** keyword:



```
1 // pure virtual members can be called          20
2 // from the abstract base class             10
3 #include <iostream>
4 using namespace std;
5
6 class Polygon {
7 protected:
8     int width, height;
9 public:
10    void set_values (int a, int b)
11        { width=a; height=b; }
12    virtual int area() =0;
13    void printarea()
14        { cout << this->area() << '\n'; }
15 };
16
17 class Rectangle: public Polygon {
18 public:
19    int area (void)
20        { return (width * height); }
21 };
22
23 class Triangle: public Polygon {
24 public:
25    int area (void)
26        { return (width * height / 2); }
27 };
28
29 int main () {
30    Rectangle rect;
31    Triangle trgl;
32    Polygon * ppolyl = &rect;
33    Polygon * ppoly2 = &trgl;
34    ppolyl->set_values (4,5);
35    ppoly2->set_values (4,5);
36    ppolyl->printarea();
37    ppoly2->printarea();
38    return 0;
39 }
```

▽ Virtual members and abstract classes grant C++ polymorphic characteristics, most useful for object-oriented projects.



```
1 // dynamic allocation and polymorphism          20
2 #include <iostream>                           10
3 using namespace std;
4
5 class Polygon {
6 protected:
7     int width, height;
8 public:
9    Polygon (int a, int b) : width(a), height(b) {}
10   virtual int area (void) =0;
11   void printarea()
12       { cout << this->area() << '\n'; }
13 };
14
15 class Rectangle: public Polygon {
16 public:
17    Rectangle(int a,int b) : Polygon(a,b) {}
18    int area()
19        { return width*height; }
20 };
21
22 class Triangle: public Polygon {
23 public:
24    Triangle(int a,int b) : Polygon(a,b) {}
25    int area()
26        { return width*height/2; }
27 };
28
29 int main () {
30    Polygon * ppolyl = new Rectangle (4,5);
31    Polygon * ppoly2 = new Triangle (4,5);
32    ppolyl->printarea();
33    ppoly2->printarea();
34    delete ppolyl;
35    delete ppoly2;
36    return 0;
37 }
```

TYPE CONVERSIONS

▷ IMPLICIT CONVERSION

- ▽ Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;  
int b;  
b=a;
```

Here, the value of `a` is promoted from `short` to `int` without the need of any explicit operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow the conversions between numerical types (`short` to `int`, `int` to `float`, `double` to `int` ...), to or from `bool`, and some pointer conversions.

- ▽ Converting to `int` from some smaller integer type, or to `double` from `float` is known as promotion, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., `-1` becomes the largest value representable by the type, `-2` the second largest, ...).

The conversions from/to `bool` consider `false` equivalent to zero (for numeric types) and to null pointer (for pointer types); `true` is equivalent to all other values and is converted to the equivalent of 1.

If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes undefined behavior.

Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is implementation-specific (and may not be portable).

- ▽ Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.
- ▽ For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:
 - Null pointers can be converted to pointers of any type
 - Pointers to any type can be converted to void pointers.
 - Pointer upcast: pointers to a derived class can be converted to a pointer of an accessible and unambiguous base class, without modifying its const or volatile qualification.

▷ IMPLICIT CONVERSION WITH CLASSES

- ▽ Implicit conversions can be controlled by means of three member functions:
 - Single-argument constructors: allow implicit conversion from a particular type to initialize an object.
 - Assignment operator: allow implicit conversion from a particular type on assignments.
 - Type-cast operator: allow implicit conversion to a particular type.

▽

```

1 // implicit conversion of classes:
2 #include <iostream>
3 using namespace std;
4
5 class A {};
6
7 class B {
8 public:
9   // conversion from A (constructor):
10  B (const A& x) {}
11  // conversion from A (assignment):
12  B& operator= (const A& x) {return *this;}
13  // conversion to A (type-cast operator)
14  operator A() {return A();}
15 };
16
17 int main ()
18 {
19   A foo;
20   B bar = foo;    // calls constructor
21   bar = foo;     // calls assignment
22   foo = bar;     // calls type-cast operator
23   return 0;
24 }
```

- ▽ The type-cast operator uses a particular syntax: it uses the operator keyword followed by the destination type and an empty set of parentheses. Notice that the return type is the destination type and thus is not specified before the operator keyword.

▷ KEYWORD EXPLICIT

▽ On a function call, C++ allows one implicit conversion to happen for each argument. This may be somewhat problematic for classes, because it is not always what is intended. For example, if we add the following function to the last example:

```
void fn (B arg) {}
```

This function takes an argument of type B, but it could as well be called with an object of type A as argument:

```
fn (foo);
```

▽ It can be prevented by marking the affected constructor with the explicit keyword:

```
1 // explicit:  
2 #include <iostream>  
3 using namespace std;  
4  
5 class A {};  
6  
7 class B {  
8 public:  
9     explicit B (const A& x) {}  
10    B& operator= (const A& x) {return *this;}  
11    operator A() {return A();}  
12 };  
13  
14 void fn (B x) {}  
15  
16 int main ()  
17 {  
18     A foo;  
19     B bar (foo);  
20     bar = foo;  
21     foo = bar;  
22  
23 // fn (foo); // not allowed for explicit ctor.  
24     fn (bar);  
25  
26     return 0;  
27 }
```

▽ Additionally, constructors marked with explicit cannot be called with the assignment-like syntax; In the above example, bar could not have been constructed with: B bar = foo;

▽ Type-cast member functions (those described in the previous section) can also be specified as explicit. This prevents implicit conversions in the same way as explicit-specified constructors do for the destination type.

▷ TYPE-CASTING

▽ C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion, known in C++ as type-casting. There exist two main syntaxes for generic type-casting: functional and c-like:

```
1 double x = 10.3;  
2 int y;  
3 y = int (x);    // functional notation  
4 y = (int) x;    // c-like cast notation
```

- ▽ These operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that -while being syntactically correct- can cause runtime errors.

```

1 // class type-casting
2 #include <iostream>
3 using namespace std;
4
5 class Dummy {
6     double i,j;
7 };
8
9 class Addition {
10    int x,y;
11 public:
12    Addition (int a, int b) { x=a; y=b; }
13    int result() { return x+y; }
14 };
15
16 int main () {
17    Dummy d;
18    Addition * padd;
19    padd = (Addition*) &d;
20    cout << padd->result();
21    return 0;
22 }
```

This code compiles without errors. Unrestricted explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or some other unexpected results.

- ▽ In order to control these types of conversions between classes, we have four specific casting operators: dynamic_cast, reinterpret_cast, static_cast and const_cast. Their format is to follow the new type enclosed between angle-brackets (\diamond) and immediately after, the expression to be converted between parentheses.

```

dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const cast <new type> (expression)
```

▷ DYNAMIC CAST

- ▽ dynamic_cast can only be used with pointers and references to classes (or with void*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.
- ▽ This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion.
- ▽ But dynamic_cast can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual

members) if -and only if- the pointed object is a valid complete object of the target type. For example:

```
1 // dynamic_cast
2 #include <iostream>
3 #include <exception>
4 using namespace std;
5
6 class Base { virtual void dummy() {} };
7 class Derived: public Base { int a; };
8
9 int main () {
10     try {
11         Base * pba = new Derived;
12         Base * pbb = new Base;
13         Derived * pd;
14
15         pd = dynamic_cast<Derived*>(pba);
16         if (pd==0) cout << "Null pointer on first type-cast.\n";
17
18         pd = dynamic_cast<Derived*>(pbb);
19         if (pd==0) cout << "Null pointer on second type-cast.\n";
20
21     } catch (exception& e) {cout << "Exception: " << e.what();}
22     return 0;
23 }
```

Null pointer on second type-cast.

Compatibility note: This type of `dynamic_cast` requires Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This needs to be enabled for runtime type checking using `dynamic_cast` to work properly with these types.

▽ The code above tries to perform two dynamic casts from pointer objects of type `Base*` (`pba` and `pbb`) to a pointer object of type `Derived*`, but only the first one is successful. Notice their respective initializations:

```
1 Base * pba = new Derived;
2 Base * pbb = new Base;
```

▽ Even though both are pointers of type `Base*`, `pba` actually points to an object of type `Derived`, while `pbb` points to an object of type `Base`. Therefore, when their respective type-casts are performed using `dynamic_cast`, `pba` is pointing to a full object of class `Derived`, whereas `pbb` is pointing to an object of class `Base`, which is an incomplete object of class `Derived`.

▽ When `dynamic_cast` cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead.

▽ `dynamic_cast` can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a `void*` pointer.

▷ `STATIIC_CAST`

▽ `static_cast` can perform conversions between pointers to related classes, not only upcasts (from pointer-to-derived to pointer-to-base), but also downcasts (from pointer-to-base to pointer-to-derived)

▽ No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, it does not incur the overhead of the type-safety checks of `dynamic_cast`.

▽

```
1 class Base {};
2 class Derived: public Base {};
3 Base * a = new Base;
4 Derived * b = static_cast<Derived*>(a);
```

This would be valid code, although `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

▽ Therefore, `static_cast` is able to perform with pointers to classes not only the conversions allowed implicitly, but also their opposite conversions.

▽ `static_cast` is also able to perform all conversions allowed implicitly (not only those with pointers to classes), and is also able to perform the opposite of these. It can :

- ▶ Convert from `void*` to any pointer type. In this case, it guarantees that if the `void*` value was obtained by converting from that same pointer type, the resulting pointer value is the same.
- ▶ Convert integers, floating-point values and enum types to enum types.

▽ Additionally, `static_cast` can also perform the following:

- ▶ Explicitly call a single-argument constructor or a conversion operator.

- ▶ Convert to rvalue references.
- ▶ Convert enum class values into integers or floating-point values.
- ▶ Convert any type to void, evaluating and discarding the value.

▷ REINTERPRET_CAST

- ▽ reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
- ▽ It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as intptr_t), is guaranteed to be able to be cast back to a valid pointer.
- ▽ The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable.

For example:

```
1 class A { /* ... */ };
2 class B { /* ... */ };
3 A * a = new A;
4 B * b = reinterpret_cast<B*>(a);
```

This code compiles, although it does not make much sense, since now b points to an object of a totally unrelated and likely incompatible class. Dereferencing b is unsafe.

▷ CONST_CAST

- ▽ This type of casting manipulates the constness of the object pointed by a pointer, either to be set or to be removed. For example, in order to pass a const pointer to a function that expects a non-const argument:

```
1 // const_cast
2 #include <iostream>
3 using namespace std;
4
5 void print (char * str)
6 {
7     cout << str << '\n';
8 }
9
10 int main () {
11     const char * c = "sample text";
12     print ( const_cast<char *> (c) );
13     return 0;
14 }
```

sample text

▽ The example above is guaranteed to work because function print does not write to the pointed object. Note though, that removing the constness of a pointed object to actually write to it causes undefined behavior.

▷ TYPEID

▽ typeid allows to check the type of an expression:

```
typeid (expression)
```

▽ This operator returns a reference to a constant object of type type_info that is defined in the standard header <typeinfo>. A value returned by typeid can be compared with another value returned by typeid using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

▽

```
1 // typeid
2 #include <iostream>
3 #include <typeinfo>
4 using namespace std;
5
6 int main () {
7     int * a,b;
8     a=0; b=0;
9     if (typeid(a) != typeid(b))
10    {
11         cout << "a and b are of different types:\n";
12         cout << "a is: " << typeid(a).name() << '\n';
13         cout << "b is: " << typeid(b).name() << '\n';
14     }
15     return 0;
16 }
```

a and b are of different types:
a is: int *
b is: int

▽ When typeid is applied to classes, typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object :

```
1 // typeid, polymorphic class
2 #include <iostream>
3 #include <typeinfo>
4 #include <exception>
5 using namespace std;
6
7 class Base { virtual void f(){} };
8 class Derived : public Base {};
9
10 int main () {
11     try {
12         Base* a = new Base;
13         Base* b = new Derived;
14         cout << "a is: " << typeid(a).name() << '\n';
15         cout << "b is: " << typeid(b).name() << '\n';
16         cout << "*a is: " << typeid(*a).name() << '\n';
17         cout << "*b is: " << typeid(*b).name() << '\n';
18     } catch (exception& e) { cout << "Exception: " << e.what() << '\n'; }
19     return 0;
20 }
```

a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived

- ▽ The string returned by member name of type_info depends on the specific implementation of your compiler and library.
 - ▽ If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeid throws a bad_typeid exception.
- ▷ <CSTDINT> / (STDINT.H)
- ▽ This header defines a set of integral type aliases with specific width requirements, along with macros specifying their limits and macro functions to create values of these types.
- ▽ Types :
- ▶ The following are typedefs of fundamental integral types or extended integral types.

signed type	unsigned type	description
intmax_t	uintmax_t	Integer type with the maximum width supported.
int8_t	uint8_t	Integer type with a width of exactly 8, 16, 32, or 64 bits.
int16_t	uint16_t	For signed types, negative values are represented using 2's complement. No padding bits.
int32_t	uint32_t	
int64_t	uint64_t	Optional: These typedefs are not defined if no types with such characteristics exist.*
int_least8_t	uint_least8_t	
int_least16_t	uint_least16_t	Integer type with a minimum of 8, 16, 32, or 64 bits.
int_least32_t	uint_least32_t	No other integer type exists with lesser size and at least the specified width.
int_least64_t	uint_least64_t	
int_fast8_t	uint_fast8_t	
int_fast16_t	uint_fast16_t	Integer type with a minimum of 8, 16, 32, or 64 bits.
int_fast32_t	uint_fast32_t	At least as fast as any other integer type with at least the specified width.
int_fast64_t	uint_fast64_t	
intptr_t	uintptr_t	Integer type capable of holding a value converted from a void pointer and then be converted back to that type with a value that compares equal to the original pointer. Optional: These typedefs may not be defined in some library implementations.*

- ▶ Some of these typedefs may denote the same types. Therefore, function overloads should not rely on these being different.
- ▶ Some types are optional (and thus, with no portability guarantees). A particular library implementation may also define additional types with other widths supported by its system. In any case, if either the signed or the unsigned version is defined, both the signed and unsigned versions are defined.

▽ Macros :

- ▶ Limits of cstdint types

Macro	description	defined as
INTMAX_MIN	Minimum value of <code>intmax_t</code>	$-(2^{63}-1)$, or lower
INTMAX_MAX	Maximum value of <code>intmax_t</code>	$2^{63}-1$, or higher
UINTMAX_MAX	Maximum value of <code>uintmax_t</code>	$2^{64}-1$, or higher
INTN_MIN	Minimum value of exact-width signed type	Exactly $-2^{(N-1)}$
INTN_MAX	Maximum value of exact-width signed type	Exactly $2^{(N-1)-1}$
UINTN_MAX	Maximum value of exact-width unsigned type	Exactly 2^N-1
INT_FASTN_MIN	Minimum value of minimum-width signed type	$-(2^{(N-1)}-1)$, or lower
INT_FASTN_MAX	Maximum value of minimum-width signed type	$2^{(N-1)}-1$, or higher
UINT_FASTN_MAX	Maximum value of minimum-width unsigned type	2^N-1 , or higher
INTPTR_MIN	Minimum value of <code>intptr_t</code>	$-(2^{15}-1)$, or lower
INTPTR_MAX	Maximum value of <code>intptr_t</code>	$2^{15}-1$, or higher
UINTPTR_MAX	Maximum value of <code>uintptr_t</code>	$2^{16}-1$, or higher

- Where N is one in 8, 16, 32, 64, or any other type width supported by the library.
- Only the macros corresponding to types supported by the library are defined.

▽ Limits of other types :

► Limits of other standard integral types:

Macro	description	defined as
SIZE_MAX	Maximum value of <code>size_t</code>	$2^{64}-1$, or higher
PTRDIFF_MIN	Minimum value of <code>ptrdiff_t</code>	$-(2^{16}-1)$, or lower
PTRDIFF_MAX	Maximum value of <code>ptrdiff_t</code>	$2^{16}-1$, or higher
SIG_ATOMIC_MIN	Minimum value of <code>sig_atomic_t</code>	if <code>sig_atomic_t</code> is signed: -127, or lower if <code>sig_atomic_t</code> is unsigned: 0
SIG_ATOMIC_MAX	Maximum value of <code>sig_atomic_t</code>	if <code>sig_atomic_t</code> is signed: 127, or higher if <code>sig_atomic_t</code> is unsigned: 255, or higher
WCHAR_MIN	Minimum value of <code>wchar_t</code>	if <code>wchar_t</code> is signed: -127, or lower if <code>wchar_t</code> is unsigned: 0
WCHAR_MAX	Maximum value of <code>wchar_t</code>	if <code>wchar_t</code> is signed: 127, or higher if <code>wchar_t</code> is unsigned: 255, or higher
WINT_MIN	Minimum value of <code>wint_t</code>	if <code>wint_t</code> is signed: -32767, or lower if <code>wint_t</code> is unsigned: 0
WINT_MAX	Maximum value of <code>wint_t</code>	if <code>wint_t</code> is signed: 32767, or higher if <code>wint_t</code> is unsigned: 65535, or higher

▽ Functions like macros :

- These function-like macros expand to integer constants suitable to initialize objects of the types above:

Macro	description
INTMAX_C	expands to a value of type <code>intmax_t</code>
UINTMAX_C	expands to a value of type <code>uintmax_t</code>
INTN_C	expands to a value of type <code>int_leastN_t</code>
UINTN_C	expands to a value of type <code>uint_leastN_t</code>

- eg. `INTMAX_C(2012)` // expands to 2012LL or similar

EXCEPTIONS

- ▷ Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.
- ▷ To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.
- ▷ An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

```
1 // exceptions
2 #include <iostream>
3 using namespace std;
4
5 int main () {
6     try
7     {
8         throw 20;
9     }
10    catch (int e)
11    {
12        cout << "An exception occurred. Exception Nr. " << e << '\n';
13    }
14    return 0;
15 }
```

An exception occurred. Exception Nr. 20

- ▷ A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.
- ▷ The syntax for catch is similar to a regular function with one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught by that handler.
- ▷ Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed.
- ▷ If an ellipsis (...) is used as the parameter of catch, that handler will catch any exception no matter what the type of the

exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers:

```
1 try {  
2     // code here  
3 }  
4 catch (int param) { cout << "int exception"; }  
5 catch (char param) { cout << "char exception"; }  
6 catch (...) { cout << "default exception"; }
```

- ▷ After an exception has been handled the program, execution resumes after the try-catch block, not after the throw statement!.
- ▷ It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:

```
1 try {  
2     try {  
3         // code here  
4     }  
5     catch (int n) {  
6         throw;  
7     }  
8 }  
9 catch (...) {  
10    cout << "Exception occurred";  
11 }
```

- ▷ EXCEPTION SPECIFICATIONS

- ▽ Older code may contain dynamic exception specifications. They are now deprecated in C++, but still supported. A dynamic exception specification follows the declaration of a function, appending a `throw specifier` to it. For example:

`double myfunction (char param) throw (int);`

- ▽ This declares a function called `myfunction`, which takes one argument of type `char` and returns a value of type `double`. If this function throws an exception of some type other than `int`, the function calls `std::unexpected` instead of looking for a handler or calling `std::terminate`.

- ▽ If this `throw specifier` is left empty with no type, this means that `std::unexpected` is called for any exception. Functions with no `throw specifier` (regular functions) never call `std::unexpected`, but follow the normal path of looking for their exception handler.



```
1 int myfunction (int param) throw(); // all exceptions call unexpected
2 int myfunction (int param);           // normal exception handling
```

▷ STANDARD EXCEPTIONS

▽ The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called `std::exception` and is defined in the `<exception>` header. This class has a virtual member function called `what` that returns a null-terminated character sequence (of type `char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.



```
1 // using standard exceptions                                         My exception happened.
2 #include <iostream>
3 #include <exception>
4 using namespace std;
5
6 class myexception: public exception
7 {
8     virtual const char* what() const throw()
9     {
10         return "My exception happened";
11     }
12 } myex;
13
14 int main () {
15     try
16     {
17         throw myex;
18     }
19     catch (exception& e)
20     {
21         cout << e.what() << '\n';
22     }
23     return 0;
24 }
```

▽ All exceptions thrown by components of the C++ Standard library throw exceptions derived from this exception class. These are:

exception	description
<code>bad_alloc</code>	thrown by <code>new</code> on allocation failure
<code>bad_cast</code>	thrown by <code>dynamic_cast</code> when it fails in a dynamic cast
<code>bad_exception</code>	thrown by certain dynamic exception specifiers
<code>bad_typeid</code>	thrown by <code>typeid</code>
<code>bad_function_call</code>	thrown by empty <code>function</code> objects
<code>bad_weak_ptr</code>	thrown by <code>shared_ptr</code> when passed a bad <code>weak_ptr</code>

▽ Also deriving from `exception`, header `<exception>` defines two generic exception types that can be inherited by custom exceptions to report errors:

exception	description
<code>logic_error</code>	error related to the internal logic of the program
<code>runtime_error</code>	error detected during runtime

▽ Eg.

PREPROCESSOR DIRECTIVES

▷ These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

▷ MACRO DEFINITIONS

▽ To define preprocessor macros we can use #define. Its syntax is:

```
#define identifier replacement
```

▽ This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ proper, it simply replaces any occurrence of identifier by replacement.

▽ #define can work also with parameters to define function macros:

```
#define getmax(a,b) a>b?a:b
```

▽ Defined macros are not affected by block structure. A macro lasts until it is undefined with the #undef preprocessor directive:

```
1 #define TABLE_SIZE 100
2 int table1[TABLE_SIZE];
3 #undef TABLE_SIZE
4 #define TABLE_SIZE 200
5 int table2[TABLE_SIZE];
```

▽ Function macro definitions accept two special operators (# and ##) in the replacement sequence:

► The operator #, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):

```
#define str(x) #x
cout << str(test);
is equivalent to : cout << "test";
```

► The operator ## concatenates two arguments leaving no blank spaces between them:

```
#define glue(a,b) a##b
glue(c,out)<<endl; is equivalent to : cout<<endl;
```

▽ Preprocessor replacements happen before any C++ syntax check.

▷ CONDITIONAL INCLUSIONS (#IFDEF, #IFNDEF, #IF, #ENDIF, #ELSE AND #ELIF)

- ▽ These directives allow to include or discard part of the code of a program if a certain condition is met.
- ▽ #ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is.

```
1 #ifdef TABLE_SIZE  
2 int table[TABLE_SIZE];  
3 #endif
```

- ▽ #ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined.

```
1 #ifndef TABLE_SIZE  
2 #define TABLE_SIZE 100  
3 #endif  
4 int table[TABLE_SIZE];
```

- ▽ The #if, #else and #elif (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows #if or #elif can only evaluate constant expressions, including macro expressions.

```
1 #if TABLE_SIZE>200  
2 #undef TABLE_SIZE  
3 #define TABLE_SIZE 200  
4  
5 #elif TABLE_SIZE<50  
6 #undef TABLE_SIZE  
7 #define TABLE_SIZE 50  
8  
9 #else  
10 #undef TABLE_SIZE  
11 #define TABLE_SIZE 100  
12 #endif  
13  
14 int table[TABLE_SIZE];
```

- ▽ The behavior of #ifdef and #ifndef can also be achieved by using the special operators defined and !defined respectively in any #if or #elif directive:

```
1 #if defined ARRAY_SIZE  
2 #define TABLE_SIZE ARRAY_SIZE  
3 #elif !defined BUFFER_SIZE  
4 #define TABLE_SIZE 128  
5 #else  
6 #define TABLE_SIZE BUFFER_SIZE  
7 #endif
```

▷ LINE CONTROL (#LINE)

- ▽ When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a

line number, so it is easier to find the code generating the error.

▽ The `#line` directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

```
#line number "filename"
```

▽ Where `number` is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.

"`filename`" is an optional parameter that allows to redefine the file name that will be shown. For example:

```
1 #line 20 "assigning variable"  
2 int a?;
```

▷ ERROR DIRECTIVE (#ERROR)

▽ This directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter:

```
1 #ifndef __cplusplus  
2 #error A C++ compiler is required!  
3 #endif
```

▽ This example aborts the compilation process if the macro name `__cplusplus` is not defined (this macro name is defined by default in all C++ compilers).

▷ SOURCE FILE INCLUSION (#INCLUDE)

▽ When the preprocessor finds an `#include` directive it replaces it by the entire content of the specified header or file. There are two ways to use `#include`:

```
1 #include <header>  
2 #include "file"
```

▽ ◇ is used to include headers provided by the implementation, such as the headers that compose the standard library (`iostream`, `string`, ...). Whether the headers are actually files or exist in some other form is implementation-defined, but in any case they shall be properly included with this directive.

▽ `""` includes a file. The file is searched for in an implementation-defined manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a header inclusion, just as if the quotes (`"")` were replaced by angle-brackets (`<>`).

▷ PRAGMA DIRECTIVE (#PRAGMA)

▽ This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with `#pragma`.

▽ If the compiler does not support a specific argument for `#pragma`, it is ignored - no syntax error is generated.

▷ PREDEFINED MACRO NAMES

▽ The following macro names are always defined (they all begin and end with two underscore characters, `__`):

macro	value
<code>__LINE__</code>	Integer value representing the current line in the source code file being compiled.
<code>__FILE__</code>	A string literal containing the presumed name of the source file being compiled.
<code>__DATE__</code>	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
<code>__TIME__</code>	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
<code>__cplusplus</code>	An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler: <ul style="list-style-type: none"> • 199711L: ISO C++ 1998/2003 • 201103L: ISO C++ 2011 Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above.
<code>__STDC_HOSTED__</code>	1 if the implementation is a <i>hosted implementation</i> (with all standard headers available) 0 otherwise.

▽ The following macros are optionally defined, generally depending on whether a feature is available:

macro	value
<code>__STDC__</code>	In C: if defined to 1, the implementation conforms to the C standard. In C++: Implementation defined.
<code>__STDC_VERSION__</code>	In C: <ul style="list-style-type: none"> • 199401L: ISO C 1990, Amendment 1 • 199901L: ISO C 1999 • 201112L: ISO C 2011 In C++: Implementation defined.
<code>__STDC_MB_MIGHT_NEQ_WC__</code>	1 if multibyte encoding might give a character a different value in character literals
<code>__STDC_ISO_10646__</code>	A value in the form <code>yyyymmL</code> , specifying the date of the Unicode standard followed by the encoding of <code>wchar_t</code> characters
<code>__STDCPP_STRICT_POINTER_SAFETY__</code>	1 if the implementation has <i>strict pointer safety</i> (see <code>get_pointer_safety</code>)
<code>__STDCPP_THREADS__</code>	1 if the program can have more than one thread

INPUT/OUTPUT WITH FILES

- ▷ C++ provides the following classes to perform output and input of characters to/from files:
 - ▽ `ofstream`: Stream class to write on files
 - ▽ `ifstream`: Stream class to read from files
 - ▽ `fstream`: Stream class to both read and write from/to files.
- ▷ These classes are derived directly or indirectly from the classes `istream` and `ostream`.

- ▷

```
1 // basic file operations
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main () {
7     ofstream myfile;
8     myfile.open ("example.txt");
9     myfile << "Writing this to a file.\n";
10    myfile.close();
11    return 0;
12 }
```

[file example.txt]
Writing this to a file.

This code creates a file called `example.txt` and inserts a sentence into it

- ▷ OPEN A FILE

- ▽ The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file. An open file is represented within a program by a stream (i.e., an object of one of these classes) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

- ▽ In order to open a file with a stream object we use its member function `open`:

```
open (filename, mode);
```

Where `filename` is a string representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

<code>ios::in</code>	Open for input operations.
<code>ios::out</code>	Open for output operations.
<code>ios::binary</code>	Open in binary mode.
<code>ios::ate</code>	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
<code>ios::app</code>	All output operations are performed at the end of the file, appending the content to the current content of the file.
<code>ios::trunc</code>	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

▽ All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:

```
1 ofstream myfile;
2 myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

▽ Each of the open member functions of classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
<code>ofstream</code>	<code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>
<code>fstream</code>	<code>ios::in ios::out</code>

- ▽ For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).
- ▽ For `fstream`, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.
- ▽ File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).
- ▽ These three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member.

Therefore, we could also have declared the previous `myfile` object and conduct the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

- ▽ To check if a file stream was successful opening a file, you can do it by calling to member `is_open`. This member function returns a bool value of true in the case that indeed the stream object is

associated with an open file, or false otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

▷ CLOSING A FILE

▽ close member is called to close a file. It flushes associated buffers and closes the file specified.

eg. myfile.close();

▽ Once this member function is called, the stream object can be reused to open another file, and the file is available again to be opened by other processes.

▽ In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

▷ TEXT FILES

▽ ios::binary flag is not included in the opening mode.

▽ These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

▽

```
1 // writing on a text file
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main () {
7     ofstream myfile ("example.txt");
8     if (myfile.is_open())
9     {
10         myfile << "This is a line.\n";
11         myfile << "This is another line.\n";
12         myfile.close();
13     }
14     else cout << "Unable to open file";
15     return 0;
16 }
```

[file example.txt]
This is a line.
This is another line.



```
1 // reading a text file
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5 using namespace std;
6
7 int main () {
8     string line;
9     ifstream myfile ("example.txt");
10    if (myfile.is_open())
11    {
12        while ( getline (myfile,line) )
13        {
14            cout << line << '\n';
15        }
16        myfile.close();
17    }
18
19    else cout << "Unable to open file";
20
21    return 0;
22 }
```

This is a line.
This is another line.

▽ The value returned by `getline` is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other error occurred.

▷ CHECKING STATE FLAGS

▽ The following member functions exist to check for specific states of a stream (all of them return a bool value):

<code>bad()</code>	Returns <code>true</code> if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
<code>fail()</code>	Returns <code>true</code> in the same cases as <code>bad()</code> , but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
<code>eof()</code>	Returns <code>true</code> if a file open for reading has reached the end.
<code>good()</code>	It is the most generic state flag: it returns <code>false</code> in the same cases in which calling any of the previous functions would return <code>true</code> . Note that <code>good</code> and <code>bad</code> are not exact opposites (<code>good</code> checks more state flags at once).

▽ The member function `clear()` can be used to reset the flags.

▷ GET AND PUT STREAM POSITIONING

▽ All `i/o streams` objects keep internally -at least- one internal position:

- ▶ `ifstream`, like `istream`, keeps an internal get position with the location of the element to be read in the next input operation.
- ▶ `ofstream`, like `ostream`, keeps an internal put position with the location where the next element has to be written.

- ▶ Finally, `fstream`, keeps both, the get and the put position, like `iostream`.
- ▽ These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:
 - ▶ `tellg()` and `tellp()`
 - ▼ These two member functions with no parameters return a value of the member type `streampos`, which is a type representing the current get position (in the case of `tellg`) or the put position (in the case of `tellp`).
 - ▶ `seekg()` and `seekp()`
 - ▼ These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:

```
seekg ( position );
seekp ( position );
```

Using this prototype, the stream pointer is changed to the absolute position `position` (counting from the beginning of the file). The type for this parameter is `streampos`, which is the same type as returned by functions `tellg` and `tellp`.

The other form for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the get or put position is set to an offset value relative to some specific point determined by the parameter `direction`. `offset` is of type `streamoff`. And `direction` is of type `seekdir`, which is an enumerated type that determines the point from where `offset` is counted from, and that can take any of the following values:

<code>ios::beg</code>	offset counted from the beginning of the stream
<code>ios::cur</code>	offset counted from the current position
<code>ios::end</code>	offset counted from the end of the stream

► Eg.

```
1 // obtaining file size                                         size is: 40 bytes.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main () {
7     streampos begin,end;
8     ifstream myfile ("example.bin", ios::binary);
9     begin = myfile.tellg();
10    myfile.seekg (0, ios::end);
11    end = myfile.tellg();
12    myfile.close();
13    cout << "size is: " << (end-begin) << " bytes.\n";
14    return 0;
15 }
```

- **streampos** is a specific type used for buffer and file positioning and is the type returned by `file.tellg()`. Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.
- These stream positioning functions use two particular types: `streampos` and `streamoff`. These types are also defined as member types of the `stream` class:

Type	Member type	Description
<code>streampos</code>	<code>ios::pos_type</code>	Defined as <code>fpos<mbstate_t></code> . It can be converted to/from <code>streamoff</code> and can be added or subtracted values of these types.
<code>streamoff</code>	<code>ios::off_type</code>	It is an alias of one of the fundamental integral types (such as <code>int</code> or <code>long long</code>).

- Each of the member types above is an alias of its non-member equivalent (they are the exact same type). It does not matter which one is used. The member types are more generic, because they are the same on all stream objects (even on streams using exotic types of characters), but the non-member types are widely used in existing code for historical reasons.

▷ BINARY FILES

- ▽ For binary files, reading and writing data with the extraction and insertion operators (`<<` and `>>`) and functions like `getline` is not efficient, since we do not need to format any data and data is likely not formatted in lines.
- ▽ File streams include two member functions specifically designed to read and write binary data sequentially: `write` and `read`. The first one (`write`) is a member function of `ostream` (inherited by `ofstream`). And `read` is a member function of `istream` (inherited by `ifstream`). Objects of class `fstream` have both. Their prototypes

are:

```
write ( memory_block, size );
read ( memory_block, size );
```

- ▽ Where `memory_block` is of type `char*` (pointer to `char`), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The `size` parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.



```
1 // reading an entire binary file
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main () {
7     streampos size;
8     char * memblock;
9
10    ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
11    if (file.is_open())
12    {
13        size = file.tellg();
14        memblock = new char [size];
15        file.seekg (0, ios::beg);
16        file.read (memblock, size);
17        file.close();
18
19        cout << "the entire file content is in memory";
20
21        delete[] memblock;
22    }
23    else cout << "Unable to open file";
24    return 0;
25 }
```

the entire file content is in memory

- ▽ First, the file is open with the `ios::ate` flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member `tellg()`, we will directly obtain the size of the file.

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

`memblock = new char[size];`

Right after that, we proceed to set the get position at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:

```
1 file.seekg (0, ios::beg);
2 file.read (memblock, size);
3 file.close();
```

▷ BUFFERS & SYNCHRONIZATION

- ▽ When we operate with file streams, these are associated to an internal buffer object of type `streambuf`. This buffer object may

