The c++11 programming language

from:
Stroustrup's books,
the « Working Draft, Standard for Programming Language C++, Document Number: N3337, Date: 2012-01-16, Revises: N3291 »
and web sites such as wikipedia, stackoverflow, cplusplus-development, stroustrup, cprogramming, cppreference, cplusplus, ...

Valerie Roy[1]

[1]MINES ParisTech
firstname.name@mines-paristech.fr

2015-2016

General introduction on the C++ language *by* B. Stroustrup

Why a modern c++ language ? by B. Stroustrup

c++ is :

- a **general-purpose** programming language
- allowing **type-rich**, **lightweight** abstractions
- particularly suited for **resource-constrained** applications
- but programmer **must takes the time** to master the language

the c++ language and the way pogrammers use it have **dramatically improved** over the years

modern C++ (such as c++11) is a **far better tool** for writing quality software than were previous versions

better programming styles and techniques, more elegant, correct, maintainable, and efficient code, ...

An emphasis on stability

because **billions** of lines of c++ are deployed world-wide : c++ puts emphasis on **stability**

thus **standards-conforming** code you write today will still work a couple of decades from now (1985 and 1995 c++ code still works)

however if you stick to **older styles**, you will be writing **lower-quality** and **worse-performing** code

you will **do better** writing software with **modern c++**

International Standard of the c++ language

International Standard of the c++ language

the **International Standard** specifies **requirements** for **implementations** of the **c++ programming language** and it **defines** the C++ language

c++ is a **general purpose** programming language **based** on the **C** programming language as described in *ISO[a]/IEC[b] 9899 :1999 Programming languages*

a. International Organization for Standardization
b. International Electrotechnical Commission

the **C standard library** is a subset of the **C++ standard library**

C++ programming language standards ANSI (American National Standards Institute) :

name	date	standard
C++98	1998	ISO/IEC 14882 :1998
C++03	2003	ISO/IEC 14882 :2003
C++TR1	2007	ISO/IEC TR 19768 :2007
C++11	2011	ISO/IEC 14882 :2011
C++14	2014	ISO/IEC 14882 :2014(E)
C++17	2017	

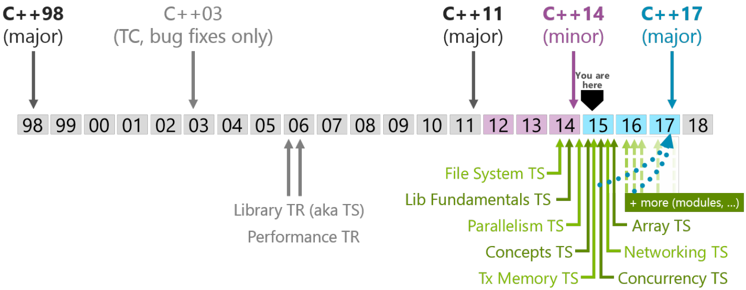


FIGURE: C++ milestones

TS (Technical Specifications)

C++ standardization

c++11 refers to the **2011 version** of the c++ programming language

most of the compilers do not implement by default the new c++11 standard

to **compile** your source files using the **c++11 standard** : you (may) have to **pass** an **option** to the **compiler**.

With the g++ compiler on linux :

```
g++ -std=c++11 FILE.cpp
```

in conclusion, this course will focus on c++11 : when a feature only exists in c++11 a *in c++11 only*will (try to) warn you programmers ?

if you have no idea how to respond to questions like :

- *What's a compiler ?*
- *What's a for-loop ?*
- *What's a type ?*

if you are wondering :

Why bother testing ?

if you a think that :

All languages are basically the same ; you simply have to know the syntax

if are confident that :

there is a single language that is ideal for every task

listen carefully to this course ...

modern c++11 features

What features does c++11 offer above c++98 ?

a machine model suitable for modern computers with lots of concurrency

language and standard-library facilities for doing systems-level concurrent programming (using multicores)

- general and uniform initialization ({})
- a simpler **`for-statement`**
- a new non-const reference type (**`&&`**) and **move** semantics
- (very) basic **Unicode** support **`char32_t`**, **`char64_t`**
- lambda functions
- general constant expressions (**`constexpr`**)
- control over class defaults (**`deleted`** and **`defaulted`** functions)
- variadic templates (i.e. with an unknown number of arguments)
- user-defined literals (**`operator""`**)

- Regular expression handling (in standard-library <regex>)
- resource management pointers (**`unique_ptr`**, **`shared_ptr`**, **`weak_ptr`**)
- random numbers
- improved containers (including hash tables)

those libraries and language features exist to support programming techniques for developing quality software

Basic Source Character Set

to **represent** a **character** inside a **text file** a **code** must be **associated** to the **character**

The **basic set** of **characters** of a **c++ program** consists of **96 characters** :

- the **space** character
- the **control** characters (horizontal and vertical tab, new-line, ...)
- the following 91 **graphical** characters :

  a b c d e f g h i j k l m n o p q r s t u v w x y z

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

  0 1 2 3 4 5 6 7 8 9

  _ { } [ ] # ( ) < > \% : ; . ? * + - / ^ & | ~ ! = , \ " '

US-ASCII (American Standard Code for Information Interchange [1]) aka **UTF8-1**

this **set** is the **US-ASCII** character-encoding scheme (American Standard Code for Information Interchange)

it **pairs** each **character** with a **code pattern**

it is based on the **English alphabet**

it is a **7-bits code** points

it comprises **128** ($2^7 = 128$) **code points** in the **range 0** to **hexa 7F**

it contains **non-printing control characters** (null, backspace, tabulations, ...)

1. see http ://www.iana.org/assignments/character-sets/character-sets.xhtml

USASCII code chart

Bits				$b_7 b_6 b_5$ →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

FIGURE: The 7-bits ASCII character set - encoding from 0 (00 in hexa, NUL) to 127 (7F in hexa, DEL)

every c++ implementation has to support

- characters from the basic source character set
- a way to name other characters (the universal character \uffff or \Uffffffff)

BUT the mapping from characters (in your file), to source characters (used at compile time) is **implementation defined**

```
#include <iostream>
using namespace std;
int main() {
  cout << "\u00A4" << '␣' << "\u00A6" << '␣' << "\u00A8" << '␣' << "\u00B4" << '␣'
       << "\u00B8" << '␣' << "\u00BC" << '␣' << "\u00BD" << '␣' << "\u00BE" << '␣'
       << endl;
}
$$ g++ file.cpp
$$ ./a.out
?
```

be careful with **character not** in the **basic source character** set

Unicode

the **request** was to **design** an **encoding** scheme to **represent simultaneously characters** of all the **languages**

some **languages** (such as Chinese or Japanese) having a **much more larger set** of **characters** to **encode**

encoding scheme sometime **needs** more **bytes** (2, 4, 8, ...) to be **represented**

to know what the c++ standard say about character set, see

- *ISO/IEC 10646-1 :1993, Information technology —Universal Multiple-Octet Coded Character Set (UCS) — Part 1 : Architecture and Basic Multilingual Plane* a programming language = lexical units + syntaxic rules

Lexical units

header names (after a **`#include`** directive), **identifiers** (including the **keywords**), **numbers**, **character**, **string** literals, **operators**, **punctuators**, ...

Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

there are 10 new keywords *in c++11 only* : alignas, alignof, char16_t, char32_t, constexpr, decltype, noexcept, nullptr, static_assert, thread_local

Syntaxic rules

they compose the grammar of the language

the standard gives the meaning of the rules

source files and programs

a file is unit of storage

A **file** is the traditional **unit** of **storage** and the traditional **unit** of **compilation**.

we do **not** take into account here systems that do **not** store, compile, and present C++ programs to the programmer as sets of files

the discussion here will **concentrate** on systems that employ the traditional use of **files**

having a complete program in one file is usually **impossible** (in particular for the code for the standard libraries)

for realistically sized applications, having **all** of the user's own code in a **single** file is both **impractical** and **inconvenient**

The way a **program** is **organized** into **files** :

- help you emphasize your **program logical structure**
- help a **human reader** understand the program
- help the **compiler** enforce that **logical** structure.

(where) a file is a unit of compilation

all of the **file** must be **recompiled** whenever a **change** has been made to it or to something on which it depends

(even for a moderately sized program), the **amount** of **time** spent **recompiling** can be **significantly reduced** by **partitioning** the program into files of suitable size

when a user **presents** a **source** file to the **compiler**, the file is **preprocessed** :

- **macro processing** is done (`ifndef, define, end, ...`)
- `#include` directives bring in headers

the result of **preprocessing** is called a **translation unit**

the **translation unit** is what the **compiler works on** and what the C++ **language rules** describe

linking together separately compiled parts

to **enable separate compilation**, the programmer must supply **declarations** providing the **type** information needed to **analyze** a translation unit **in isolation** from the rest of the program

the **linker** is the program that **binds** together the separately compiled parts (it detects many kinds of inconsistencies)

the linker detects many kinds of **inconsistencies**

the memory a program can address[2]

every **application** running on your **operating system** has its **unique** address space

your application sees it as a **continuous block** of memory

the size of the memory your program *sees* is the size of the memory your program (computer) *can address* (using pointers)

on a 32-bit processor, a program can address almost 2^{32} bytes of (RAM) memory

a question : ***what*** *happens when a process (the execution of a program) want to access more memory than your machine physically has available as RAM ?*

2. http ://www.cprogramming.com/tutorial/virtual_memory_and_heaps.html

the **virtual memory** (the process' address space)

it uses a **virtual address** mecanism where **part** of the **hard disk** can be **mapped** together with **real** memory

the **process** does not know whether the address is **physically stored** in RAM or on the hard disk

the **operating system** maintains a **table** (mapping virtual addresses to their correspondent physical addresses)

in each process, the virtual memory available to that process (its address space) is typically organized in several segments :

- a **text** section (where the actual code is kept)
- a **data** section (where global static variables are kept)
- a **stack** (used to store memory for function arguments, return values, and automatic variables)
- the **heap** (used for dynamic allocation)

the segments

```
+-----------+
|           |
|   text    | (fixed size)
|           |
+-----------+
|           |
|   data    | (fixed size)
|           |
+-----------+
|   stack   | | growth
+-----------+ V
|           |
|   free    |
|           |
+-----------+ ^
|   heap    | | growth
+-----------+
```

the size of the **text** and **data** segments are **known** at **compile-time**

the size of the **stack** and **heap** segments **grows** and **shrinks** during program execution

all static variables are global (the notion of variables' scope is enforced by the compiler)

variables are memory addresses (the compiler keeps track of which addresses are used by each subprogram), functions have addresses, ...

the **heap memory** (dynamic memory allocation)

allocation of dynamic memory is decided by the programmer (**`malloc`**, **`new`**, **`new[]`**, ... allocate and return a memory address)

realease of dynamic memory is done by the programmer (**`free`**, **`delete`**, **`delete[]`** release a given memory address)

the dynamic memory allocation is time consuming

When the allocation is invoked :

- it looks for a free memory block that fits the size of your request (with a smart algorithm - not returning the first that fit the size and not scanning the whole the memory)
- when found, it marks it as reserved
- it returns a pointer to that location

prefer automatic memory (allocated on the stack) whenever possible for not too large allocation

A c++ example

```
+-----------------------+
| int i;                |
| int main () {         |
|   int j;              |
|   int k = 12;         |  code segment
|   char* pc = new char;|
|   *pc = 'y';          |  <-- YOU ARE HERE
|   delete pc;          |
| }                     |
+-----------------------+
| i : 0                 |
|                       |  data segment
+-----------------------+
| pc : XXX              |
| k  : 12               |  stack segment
| j  : ?                |
+-----------------------+
|        free           |
+-----------------------+
| XXX: 'y'              |  heap segment
+-----------------------+
```

i is a static and global integer variable zero-initialized (by default)

j is an uninitialized automatic integer variable, pushed on the stack segment (you don't know itsvalue)

k is an automatic integer variable, pushed on the stack segment with the value **12**

new char; has requested from the heap the address of a memory the size of a character (**XXX**)

pc is an automatic pointer to character, pushed on the stack segment with the value of the character address **XXX**

***pc** is piece of memory the size of the character at the address **pc**, we initialize it with the character **'y'**

When you are going something wrong

another question : ***what*** *happens when a program tries to access a memory that it is not allowed to access (or in a way that is not allowed) ?*

you get a **segmentation fault** and the operation is not done

it keeps you from corrupting the memory

but you are clearly doing something **wrong** with memory

A c++ example

```
+-----------------------+
| int main () {         |
|   int* j;             |  code segment
|   *j  = 12;           |  <-- a big problem here !!!
| }                     |
+-----------------------+
|    (data segment)     |
+-----------------------+
| j  : ?                |
|                       |
|        (stack segment)|
+-----------------------+
|        free           |
+-----------------------+
|     (heap segment)    |
+-----------------------+
```

`j` is an uninitialized automatic *pointer to an integer* variable, pushed on the stack segment : **you don't know its value but it surely has one**

`*j` try to access the memory *the size of an integer* at the address `j` that is wrongly initialized

a segmentation fault occurs

Notice that : the mecanism of virtual memory prevents you from accessing the memory space of another process memory

thus the segmentation fault occurs in case of accessing your own memory in an improper way

for example : you are trying to write to non-writable space, or to access the part of the virtual address space that is not mapped to physical one, ...

The `main` function

The unique entry point of a program is the `main` function

A **program** shall contain a **global** and **unique function** called **`main`**

It is the **designated start** of the **program**

It **must have** a **return type** of type **`int`** (*otherwise its **type** is **implementation-defined***)

The function **`main`** cannot be **overloaded**

The unique entry point of a program is the **main** function

The function **main** cannot be declared **inline**

The function **main** cannot be declared **static**

The function **main** cannot be called within a program

two **definitions** of **main** are **allowed** :

```
int main () { /* ... */ }
```

```
int main (int argc, char* argv []) { /* ... */ }
```

Passing **command-line arguments** to `main`

```
int main (int argc, char* argv []) { /* ... */ }
```

`(argc-1)` is the **number** of **arguments passed** to the **program**

they are **supplied** :

- in **`argv[0]`** to **`argv[argc-1]`**
- as **character strings** (*pointers to the first character of null-terminated strings*)
- **`argv[0]`** is the name used to invoke the program

```
#include <iostream>
using namespace std;
int main (int argc, char* argv[]) {
  cout << "program␣name:␣" << argv[0] << endl;
  cout << "number␣of␣arguments:␣" << argc−1
       << endl;
  cout << "arguments:␣";
  for (int i = 1; i < argc; ++i)
    cout << argv[i] << "␣";
  cout << endl;
  return 0;
}
```

```
$$ a.out
program name: a.out
number of arguments: 0
arguments:
$ echo $?
0
```

```
$$ a.out 1 Hello 12.3f
program name: a.out
number of arguments: 3
arguments: 1 Hello 12.3f
```

Uniform initialization : initializer

expressing initialization

an initialization determines the **initial** value of a variable

there are four syntaxic forms to express initialization in c++ :

```
int  i  {12};
int  j  = {12};
int  k = 12;
int  l  (12);
```

the first one uses the **{ }**-initializer and exists *in c++11 only*

the second one does exist in older version of c++ but in *in c++11 only* is behaves the same as first one

the **{ }**-initializer can be used in **every context**

it is strongly recommended by B.J. Stroustrup : *Prefer the -initializer syntax for declarations with a **named** type*

{ }-initializer *does not allow* **narrowing** conversion

the *in c++11 only* **{ }-initializer** *does not allow* **narrowing** (it **warns** you)

a value of some type (integer, float, double, ...) **cannot be converted** in another type (integer, float, double, ...) that cannot hold its value

```
#include <limits>
int main () {
  // the biggest int i can store in my program is 2147483648
  std::numeric_limits<int>::max();
   int i = 2147483647; // OK
   int j = 2147483648; // SILENT narrowing conversion !!
   int k {2147483647}; // OK
   int l {2147483648}; // WARNED narrowing conversion
}
```

i is 2147483647
j is -2147483648
k is 2147483647
l is -2147483648

inside {} there is a narrowing conversion of **`2147483648l`** from **`long int`** to **`int`**

the same with different numeric type : { }-initializer *does not allow* **narrowing conversion**

a floating-point value **cannot be converted** to an integer type (you get a warning)

```
int main () {
  float f = 3.14;
  int i = f;   // SILENT narrowing conversion
  int j {f};   // WARNED narrowing conversion
}
```

inside {} there is a narrowing conversion of **`3.14`** from **`double`** to **`int`**

if you really need to do narrowing conversion, **tells** the compiler **you know** what you are doing with a **cast**

```
int main () {
  float f = 3.4;
  int i = static_cast<int>(f);
}
```

depending on the size of the floating-point and integer (on your computer), an integer value might not be **converted** exactely to a floating-point type [a]

a. for example, when the number of bits integers are coded on (not counting the sign bit) is greather that the number of digits in the mantissa, it might happen the **empty initialization-list** {} indicates a **default value**

```
int i {};               // i becomes 0
double d {};            // d becomes 0.0
char* p {};             // p becomes nullptr
char c {};              // c becomes '\0'
vector<int> v {};       // v becomes the empty vector
string s {};            // s becomes ""
```

Fundamental types in c++ :

- booleans
- character types
- integer types
- floating-point types
- prefixed and postfixed types

Types

Every **identifier** has a **type** associated with it [a]

a. Notice that an object can have several types

the **type** determines :

- what **operations** can be **applied** to the identifier
- how the operations are **interpreted**

```
#include <iostream>
using namespace std;
int main () {
  float f {1.0};
  float g {3.0};
  cout << f/g << endl;
}
$$ g++ file.cpp
$$ ./a.out
0.333333
```

```
#include <iostream>
using namespace std;
int main () {
  int i {1};
  int j {3};
  cout << i/j << endl;
}
$$ g++ file.cpp
$$ ./a.out
0
```

boolean

dealing with boolean in c++ : the type `bool`

values of **type** `bool` are `true` or `false`

the operations on boolean values are : **and** (`&&`), **or** (`||`), **not** (`!`)

```
#include <iostream>
int main () {
  bool b1 {true},
       b2 {not b1},
       b3 {! b2},
       b4 {b1 or b2},
       b5 {b1 || b2},
       b6 {b1 and b3},
       b7 {b1 && b3};
  std::cout << std::boolalpha
            << b1 << "␣" << b2 << "␣" << b3 << "␣" << b4 << "␣"
            << b5 << "␣" << b6 << "␣" << b7 << "␣" << std::endl;
}
$$ ./a.out
true false true ...
```

`bool` is the type returned by conditions

`bool` is the **type** of a **condition** in an **`if`** or in an **iteration statement**

values of type **`bool`** can participate in **integral promotions** and conversely

by definition, **`true`** has the value **`1`** when converted to an integer and **`false`** has the value **`0`**

and conversely, integers can be implicitly converted to **`bool`** values (nonzero integers convert to **`true`** and **`0`** converts to **`false`**)

if you use the {}-initializer (*in c++11 only*), you prevent narrowing : you must be explicit

```
int main () {
  int i {1};
  bool b {i != 0};
}
```

a pointer can be implicitly converted to a bool (a non-null pointer converts to **`true`**, pointers with the value **`nullptr`** convert to **`false`**)

```
bool b = p;                // SILENT NARROWING to true or false
bool b2 {p != nullptr };   // EXPLICIT test against nullptr
if (p) {                   // equivalent (and prefered) to p != nullptr
}
```

implementation-defined aspects of type `bool` are in <limits>

it tells you the **numeric limits** of a type i.e the **minimal** and the **maximal** element

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

int main () {
  cout << "bool "
       << (int)(numeric_limits<bool>::min()) << " "
       << (int)(numeric_limits<bool>::max()) << endl;
}
```

```
$$ ./a.out
bool 0 1
```

integral types

Integer literal (**decimal**, **octal**, **hexadecimal** integers)

it is a **sequence** of **digits without period** (a **prefix** can specify its **base**)

a **decimal integer literal begins** with a **digit other** than **0**

an **octal integer literal begins** with the digit **0** (decimal digits from 0 to 7)

```
#include <iostream>
int main () {
  int i {014};
  std::cout << i << " " << std::oct
            << i << std:: endl;
}
$$ ./a.out
12 14
```

a **hexadecimal integer literal begins** with **0x** or **0X** (decimal digits and letters **a/A** through **f/F**)

```
#include <iostream>
int main () {
  int i {0xA};
  std::cout << i << " " << std::hex
            << i << std:: endl;
}
$$ ./a.out
10 a
```

Integer literal (**signed**, **unsigned**, **long**, **long long**)

an integral is **signed** by **default**

a **suffix** can specify its **type**

unsigned-suffix : **u U**

```
#include <iostream>
int main () {
  std::cout << 12u << "␣" << sizeof(12U) << std:: endl;
}
$$ ./a.out
12 4
```

long-suffix : **l L**

```
#include <iostream>
int main () {
  std::cout << 12l << "␣" << sizeof(12L) << std:: endl;
}
$$ ./a.out
12 8
```

long-long-suffix : **ll** and **LL**

```
#include <iostream>
int main () {
  std::cout << 12ll << "␣" << sizeof(12LL) << std:: endl;
}
$$ ./a.out
12 8
```

Integer types

five standard `signed` integer types :

- `signed char`
- `short` / `short int`
- `int`
- `long` / `long int`
- `long long` / `long long int`

each **type provides at least as much storage** as those **preceding it** :
`signed char` $\leq$ `short int` $\leq$ `int` $\leq$ `long int` $\leq$ `long long int`

```
int main () {
  short i {};
  short int j {};
  unsigned short int l {};
  long k {};
  long int m {};
  unsigned long int n {};
  long long t {};
  long long int h {};
  unsigned long long int f {}
}
```

for **each `signed integer` type**, there **exists** an **`unsigned integer` type**

each of which **occupies** the **same amount** of **storage** as the **corresponding `signed integer`** type

Integer types

Plain ints have the **natural size** of the **architecture** of the **runtime** (execution) environment

large enough to **contain** any **value** in the **range** of **`INT_MIN`** and **`INT_MAX`** (defined in **`<climits>`**)

```
#include <climits>
#include <iostream>
int main () {
  std::cout << "INT_MIN = " << INT_MIN << std::endl;
  std::cout << "INT_MAX = " << INT_MAX << std::endl;
}
$$ g++ -std=c++11 file.cpp
$$ ./a.out
INT_MIN = -2147483648
INT_MAX = 2147483647
$$
```

Integer overflow

be **careful** with **integer overflow** (**master** what you are **doing**)

```
#include<iostream>
int main () {
  using namespace std;
  unsigned int i {1};
  while (i > 0)
    i = i+1;
  cout << i-1 << " + 1 " << " = " << i << endl;
}
$$ ./a.out
4294967295 + 1 = 0
```

Maximum and minimum values of integral types : <limits>

```
#include <limits>
#include <iostream>
int main () {
  using namespace std;
  cout << "int ["
       << numeric_limits<int>::min() << ",␣"
       << numeric_limits<int>::max() << ",␣" << endl;
  cout << "unsigned␣int["
       << numeric_limits<unsigned int>::min() << ",␣"
       << numeric_limits<unsigned int>::max() << ",␣" << endl;
  cout << "long␣int␣["
       << numeric_limits<long int>::min() << ",␣"
       << numeric_limits<long int>::max() << "]" << endl;
  cout << "unsigned␣long␣int␣["
       << numeric_limits<unsigned long int>::min() << ",␣"
       << numeric_limits<unsigned long int>::max() << "]" << endl;
}
```

```
int               [−2147483648, 2147483647]
unsigned int      [0, 4294967295,]
long int          [−9223372036854775808, 9223372036854775807]
unsigned long int [0, 18446744073709551615]
```

floating-point literals and floating-point types

floating point literal

a **floating point** is an **approximation** of a **real number**

```
#include <iostream>
int main () {
  std::cout << 12. << std::endl
            << 12.3 << std::endl
            << 12.3e-7 << std::endl
            << 12e2 << std::endl
            << .7E-10 << std::endl;
}
$$ ./a.out
12
12.3
1.23e-06
1200
7e-11
```

a **floating point literal** is **composed** of :

- an **integer part**
- a **decimal point**
- a **fraction part**
- an **optional integer exponent introduced** by **`e`** or **`E`**

Floating point types

three types : **`float`**, **`double`**, and **`long double`**

```
#include <iostream>
int main () {
   float f {12.};
  std::cout << f << "␣" << sizeof(f) << std:: endl;
  double d {12.};
  std::cout << d << "␣" << sizeof(d) << std:: endl;
  long double l {12.};
  std::cout << l << "␣" << sizeof(l) << std:: endl;
}
$$ ./a.out
2 4
12 8
12 16
```

each **type provides at least as much storage** as those **preceding it** :
`float` $\leq$ **`double`** $\leq$ **`long double`**

Floating point literal

a **suffix** can specify its **type**

`f` and **`F`** for **`float`**

```
#include <iostream>
int main () {
  std::cout << 12.f << "␣" << sizeof(12.f) << std:: endl;
}
$$ ./a.out
12 4
```

without suffix the default is **`double`**

```
#include <iostream>
int main () {
  std::cout << 12. << "␣" << sizeof(12.) << std:: endl;
}

$$ ./a.out
12 8
```

`l` and **`L`** for **`long double`**

```
#include <iostream>
int main () {
  std::cout << 12.l << "␣" << sizeof(12.L) << std:: endl;
}
$$ ./a.out
12 16
```

Maximum and minimum values of floating point types <limits>

```
#include <limits>
#include <iostream>

int main () {
  using namespace std;
  cout << "float ["
    << (float)(numeric_limits<float>::min()) << ", "
    << (float)(numeric_limits<float>::max()) << "]" << endl;
  cout << "double["
    << (double)(numeric_limits<double>::min()) << ", "
    << (double)(numeric_limits<double>::max()) << "]" << endl;
  cout << "long double["
    << (long double)(numeric_limits<long double>::min()) << ", "
    << (long double)(numeric_limits<long double>::max()) << "]" << endl;
}
```

```
$$ g++ -std=c++11 file.cpp
$$ ./a.out
float       [1.17549e-38, 3.40282e+38]
double      [2.22507e-308, 1.79769e+308]
long double [3.3621e-4932, 1.18973e+4932]
$$
```

void

void

The **void** type is an **incomplete type** that has an **empty set** of **values**

used as **return type** for **functions** that **do not return** a value

(optional) used as a **parameter** for **functions** that do **not take** any **argument**

```
void foo(void) {}

int main () {
  foo ();
}
```

it is the **base type** for **pointers** to **objects** of **unknown type**

```
int main () {
  void* p1 {};
  void* p2 { nullptr }; // in c++ 11 only
}
```

character literals and character types

Dealing with character literals in c++

a character literal in c++ is **one** or **more** characters enclosed in **single quotes** ('a', ...)

```
#include <iostream>
int main () {
  std::cout << 'a' << '\t' << 'tt';
}
```

you get a warning (multi-character character constant for 'tt')

*in c++11 only*the characters enclosed in **single quotes** may be **optionally preceded** by **u**, **U**, or **L** (**otherwise** it is an **ordinary character literal**)

```
#include <iostream>
int main () {
  std::cout << sizeof('a') << sizeof(u'a') << sizeof(U'a') << sizeof(L'a');
}
```

```
1 2 4 4
```

you can see that **'a'**, **u'a'**, **U'a'** and **L'a'** have **different sizes**

Types of the character literal in c++11

an ordinary character literal has type :**`char`**

a character literal that **begins** with the letter :

- **`u`** has **type `char16_t`** *in c++11 only*
- **`U`** has **type `char32_t`** *in c++11 only*
- **`L`** (aka wide-character) has **type `wchar_t`**

```
#include <iostream>
int main () {
  std::cout << sizeof(char) << sizeof(char16_t) << sizeof(char32_t) << sizeof(wchar_t);
}
```

```
1 2 4 4
```

the two new character types *in c++11 only* : **`char16_t`** and **`char32_t`** are designed to **deal with character encoding**

what does the c++ standard say about character types ?

the type `char` **shall be large enough** to store **characters** from the **Basic source character set**

thus `char` are almost universally considered 8-bit long type (that can hold $2^8 = 256$ values)

with n **bits** :

unsigned type range	$[0 \text{ to } 2^n - 1[$	from 0 to 255
signed type range	$[-2^{n-1}, 2^{n-1} - 1]$	from -128 to $+127$

it is **implementation-defined** whether a `char` is signed or not

$\Leftrightarrow$ it is **not safe** to **assume** that `char` can hold **more than 127 characters**

what does the c++ standard say about character types ?

Characters can be **explicitly declared `unsigned`** or **`signed`**

remember that **plain `char`** values outside $[0, 127[$ lead to **portability** problems :

```
#include <iostream>
int main () {
  using namespace std;
  signed char c1 {127};   // OK
  signed char c2 {128};   // ERROR (warning)
  unsigned char c3 {128};// OK
  char c4 {128};          // NOT PORTABLE
  std::cout << int(c1) << int(c2) << int(c3) << int(c4);
}
```

```
127 −128 128 −128
```

plain **`char`**, **`signed char`**, and **`unsigned char`** are three distinct types but occupy the **same amount** of **storage**

Maximum and minimum values of character types <limits>

```
#include <limits>
#include <iostream>
int main () {
  using namespace std;
  cout << "char["
       << (int)(numeric_limits<char>::min()) << ", "
       << (int)(numeric_limits<char>::max()) << "]" << endl;
  cout << "unsigned char ["
       << (int)(numeric_limits<unsigned char>::min()) << ", "
       << (int)(numeric_limits<unsigned char>::max()) << "]" << endl;
  cout << "char16_t["
       << (long int)(numeric_limits<char16_t>::min()) << ", "
       << (long int)(numeric_limits<char16_t>::max()) << "]" << endl;
  cout << "char32_t["
       << (long int)(numeric_limits<char32_t>::min()) << ", "
       << (long int)(numeric_limits<char32_t>::max()) << "]" << endl;
}
```

```
char          [-128, 127]
unsigned char [0,    255]
char16_t      [0,    65535]
char32_t      [0,    4294967295]
```

strings and raw-strings *in c++11 only*

sometime, you have to write string literals with a *specific* uses of backslash

for example you are using an escape sequence inside a string (or you are writing a regular expression) but you don't want C++ to interpret it

```
#include<iostream>
int main () {
  const char* s1 = "\t\"Hello\n\tWorld !\" \n";         // interpreted
  const char* s2 = "\\t \\\" Hello\\n\\tWorld !\\\"\\ n";   // NOT interpreted
  std::cout << s1 << std::endl;
  std::cout << s2 << std::endl;
}
```

```
        "Hello
         World !"
\t\"Hello\n\tWorld !\"\n

```

A **raw** string literal is a string literal where a **backslash** is just a **backslash**, a double quote is just a double quote, ...

```
#include<iostream>
int main () {
  const char* s1 = "\t\"Hello\n\tWorld !\" \n";     // interpreted
  const char* s2 = R"(\t\"Hello\n\tWorld !\"\n)";   // NOT interpreted
}
```

Pointers

what is a pointer ?

a pointer is the address of an object allocated somewhere (in the stack, the heap or the static store)

for a **type `T`**, **`T*`** is the type **pointer to `T`**

in an expression :

- **&** is the **address of** *operator* (it returns the address of an object)
- ***** is the **object pointed by** *operator* (it returns the object at the given address)

the **implementation** of **pointers** is directly **bound** to the **addressing mechanisms** of the **machine**

in c++11 only **`nullptr`** is the null pointer

```
int main () {
  float* f { nullptr };   // a null pointer to a float
  int i {};               // an integer in the stack with the value 0
  int* pi {&i};           // pi is set to the address of i
  (*pi) = 12;             // the integer i is modified
}
```

in c++11 only `nullptr`

the literal **`nullptr`** represents the null pointer i.e. a pointer that **does not** point to an object

it can be assigned to any pointer type (but only to pointer)

there is only one **`nullptr`** shared by all pointer types, rather than a null pointer for each pointer type

before **`nullptr`**, zero **0** was used as a notation for the null pointer

```
int main () {
  float* pf {};  // nullptr by default
  char* pc { nullptr };  // ok nullptr is explicit
  int* pi {0};  // ok (will be nullptr)
}
```

`nullptr` is a value of type **`std::nullptr_t`** from the **`c cstddef`** library

```
#include <cstddef>
int main () {
  std::nullptr_t p { nullptr };
}
```

Bad use of pointers

```
#include <iostream>
int main () {
  using namespace std;
  float * pf;
  cout << *pf << endl;
}
```

`pf` is an object of type pointer to **`float`** allcoated in the stack

`pf` is **not initialized**

its **value** can be **anything**

but it **still represents** the **address** of an **object**

`*pf` is the **object** in **memory `pf` points to**

because **`pf`** is not a **legal adress** you will get a **memory error** (segmentation fault)

Arrays

arrays

an array is the fundamental way of representing a sequence of objects in memory

it is the solution for simple fixed-length sequences of objects of a given type in memory

an array can be allocated statically, on the stack, and on the free store

```
int t1 [10];                  // 10 ints statically allocated
int main () {
  float t2 [20];              // 20 floats on the stack
  char* p {new char[40]};     // 40 chars on the free store
}
```

array is a low-level facility (should be used inside implementations of higher-level) data structures (**`string`**, **`vector`**, ...)

arrays are low-level facilities

you cannot initialize or assign one array with another (not even of exactly the same type)

```
int main () {
  int t1 [5];
  int t2 [5] = t1 ;  // ERROR array must be initialized with a {}
  t1 = t2 ;          // ERROR invalid array assignment
}
```

you cannot pass arrays by value

avoid arrays as function arguments

prefer more reliable resource handles (string, vector)

examples of global arrays

```
bool t1 [12];
const float t2 [] = { 2.4, 4e12, .3, −5. };
const float t2 [] { 2.4, 4e12, .3, −5. }; (only in c++11)
const int N = 13;
int t3 [N];
int M = 20;
int t4 [M]; // ERROR M is not a compile−time constant
int main () {
  t2[0] = .12; // ERROR t2 is const
}
```

- `t1` : **global array** of 12 false-initialized **boolean** [a]
- `t2` : **global array** of 4 initialized **constant floats**
- `t3` : array of **`N=13`** zero-initialized integers
- `t4` : a **compile-time error** because **`M`** is **not constant**
  - **assignment** of the **first element** of **`tab2`** causes a **compile-time because** constant **cannot** be **changed**

---

a. static global variables are zero-initialized

array initialization-list

an array can be initialized by a list of values

```
int t1[] { 0, 1, 2, 3, 4 };
int t2[] { 9, 8, 7, 4, 3};
```

an array can be declared without a specific size but with an initializer list

when needed the size is calculated by counting the elements of the initializer list

`t1` and **`t2`** are of type **`int[5]`**

it is an error to give surplus elements in an initializer

```
int t2[5] { 9, 8, 7, 4, 3, 2, 1};
```

but if you supply too few elements, 0 is used for the rest

```
int t2[8] { 9, 8, 7, 4};
```

is equivalent to :

```
int t2[8] { 9, 8, 7, 4, 0, 0, 0, 0};
```

Example of **Arrays**

three ways of initializing a **global** empty array of ints

```
int t[3];
```

```
int t[3] {};
```

```
int t[] {0, 0, 0};
```

in c++11 only

```
int t[3] {};
```

```
int t[] {0, 0, 0};
```

three ways of initializing a **global** array of zero-initialized pointers to integer

```
int* p [4];
```

```
int* p [4] {};
```

```
int* p [4] { nullptr , nullptr , nullptr , nullptr };
```

initializing an array of chars

```
char t [6] {'h', 'e', 'l', 'l', 'o'}; // '\0'
```

```
char t [] {"hello"}; // '\0'
```

arrays as **global** variables or constants

in **`T`** *tab* **`[`** *expr* **`]`**, for **global variables** or **constant**, **expr `must be`** a **compile**-**time constant**

Arrays as local variables or constants

local arrays are **declared** the same way global arrays are

a **local array** is **allocated** in the **stack**

we do not need to know the size of the array at compile-time

```
int main () {
  bool t1 [12];
  const float t2 [] { 2.4, 4e12, .3, -5. };
  const int N = 13;
   int t3 [N];
   int M = 20;
   int t4 [M];     // OK (local array on the stack)
  tab2[0] = 12.; // ERROR (at compile-time)
}
```

How are they initialized ?

Arrays in memory

in **memory**, an **array** is a **contiguous zone**

in **memory**, an array **`tab`** of objects occupies a **contiguous zone** of **size** : **`sizeof(tab)`** (*in number of bytes - unsigned char*)

this **zone** is **large enough** to **contain `n` object** of **type `T`** of **size** : $\mathbf{n} \times \mathit{sizeof}(T)$

```
int main () {
  bool tab1 [12];
  const float tab2 [] { 2.4, 4e12, .3, −5. };
  const int N = 13;
   int tab3 [N];
   int M = 20;
   int tab4 [M];
  sizeof(tab1) / sizeof(bool);   // the number of objects in tab1 (12)
  sizeof(tab2) / sizeof( float ); // the number of objects in tab2 (4)
  sizeof(tab3) / sizeof( int );   // the number of objects in tab3 (13)
}
```

Array in memory

T *tab* **[n]**

in an **expression `tab`** is **converted** to a **pointer** to the **first element** of the **array**

in an **expression `*tab`** is the **first element** of the **array**

in an **expression `tab+i`** is a **pointer** to the $i+1$ **element** of the **array** (if any !)

```
int main () {
  int tab [3] { 42, 17, 81 };
  sizeof(tab)/sizeof( int );  // number of ints
  tab [0];      // the first element
  *tab;         // the first element
  tab [1];      // the second element
  *(tab+1);     // the second element
  tab[ i ];     // the i+1 element
  *(tab+i );    // the i+1 element
}
```

Arrays in memory

`tab[i]` is equivalent to **`*(tab+i)`** :

- we consider the address **`tab`**
- we add to this address *i* piece of memory (the size of the type here **`int`**)
- we have the **`(tab+i)`** the address of the $i+1$ element

```
tab             tab+1           tab+2
|               |               |
v       0       v       1       v       2
|---------------|---------------|---------------|
|      42       |      17       |      81       |
|---------------|---------------|---------------|
  <-sizeof(int)->  <-sizeof(int ->  <-sizeof(int)->
```

Problem with **local arrays**

```
int main () {
  int t[-50]; // COMPILE-TIME ERROR (size of array 't' is negative)
}
```

```
int main () {
  int n {-50};
  int tab [n];   // RUNTIME ERROR (invalid memory manipulation)
}
```

the execution of your program can be aborted with a **segmentation fault (core dumped)**

the execution of your program can continue a little while in a corrupted memory

Multidimensional array

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

int main() { // definition of the main function
    int mat[3][4] { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            cout << mat[i][j] << " "; // NOT mat[i,j] !
        }
        cout << endl;
    }
    return 0;
}
0 1 2 3
4 5 6 7
8 9 10 11
```

constexpr *in c++11 only*

remember that global array must have a constant size (a compile-time constant)

```
const int N = 13;
int t3 [N];          // OK the size is a COMPILE-TIME CONSTANT
```

```
int M = 20;
int t4 [M];          // ERROR M is not a compile-time CONSTANT
```

```
const int size () {
  return 20;
}
const int t4 [size ()];   // ERROR the size is not a COMPILE-TIME constant
```

```
const double gEarth = 9.8;
double gMoon = gEarth / 6.0;  // gMoon cannot be a constant !!
```

in **previous c++ versions**, a **constant expression** was **not allowed** to **contain** a **function call**

in c++11 only you have a way to **guarantee** that an **initialization** is done at **compile time**

objects declared **constexpr** have their initializer evaluated at compile time

`constexpr` for array initialization

because c++ **requires** the **use** of **constant expressions** when **defining** a global **array**

C++11 introduced the **keyword `constexpr`**

`constexpr` allows the **user** to **guarantee** that a **function** is a **compile-time** constant

```
constexpr int size () { return 20;}
int tab[size ()];
```

the **compiler understands** that **`size()`** is a **compile**-**time constant**

```
constexpr double gEarth = 9.8;
constexpr double gMoon = gEarth / 6.0;
```

`constexpr` can be **used** for **non integral types**

reference

Reference

a **reference** is like a **constant pointer** that is **automatically dereferenced**

it is **usually used** for **function arguments** list and **return value**

When a **reference** is **created** it must be **initialized** to an **existing object**

```
int main () {
  int i = 12;
  int& ri = i;
  ri++;     // i is now 13
  ri=156; // i is now 156
}
```

Incrementing `ri` here is **incrementing i**

Reference Rules

A **reference** must be **initialized** when it is **created** to *an existing piece of storage*

when **created** a **reference cannot change** to **refer** to **another** object

you **cannot access** the **reference** : you **directly** access the **object refered** to

You **cannot have null reference** : a **reference** is **connected** to an **existing piece of storage**

References and Functions Arguments

arguments to **functions** can be passed by **reference**

as the **reference refers** to an **existing** object :

- any **modification** throught the **reference inside** the **function**
- cause **change** to an object that is **outside** the function

```
void swap_ptr (int* pi, int* pj) {
  int aux = *pi;
  *pi = *pj;
  *pj = aux;
}
void swap_ref (int& i, int& j) {
  int aux = i;
  i = j;
  j = aux;
}
```

```
int main () {
  int a = 12;
  int b = 89;
  // a == 12, b == 89
  swap_ptr(&a, &b);
  // a == 89, b == 12
  swap_ref(a, b);
  // a == 12, b == 89
}
```

Reference and Lvalue (assignable value)

the **initializer** for a `T&` **must be** a **lvalue** of type `T`

```
int& ri = 1;
ex20.cxx: In function 'int main()':
ex20.cxx:2: error: could not convert '1' to 'int&'
```

Reference on Const Objects

the **initializer** for a `const T&` **need not** be a **lvalue** of type `T`

```
const int& ri = 1;
```

a **temporary object** is **created** with the **value**

```
const int temp = 1;
const int& ri = temp;
```

the **temporary** is **used** as the **initializer** for the **reference**

the **temporary persists** until the **end** of its **reference scope**

Reference on Return Values

with **reference**, **functions** can be **used** on both **left-hand** and **right-hand** side of an **assignment**

```
class Foo {
  int value;
public:
  Foo (int i) { value = i; }
  int& getValue () {
    return value;
  }
};
```

```
int main () {
  Foo foo(10);
  foo.getValue()++;
  int i1 = foo.getValue();
  // i1 == 11
}
```

Statements : labeled, expression, compound, selection, iteration, jump, declaration, exceptions, list

Labelled Statements

a **labelled** statement is **followed** by a **:**

labelled statements are :

- the label **`case`** and the label **`default`** in **`switch`** statement
- identifier labels for **`goto`**

```cpp
#include <iostream>
int main () {
  int x = 10;
  ici :
  {
    std::cout << x << std::endl;
    x--;
    if (x) goto ici ;
  }
}
```

here **`ici`** is a labelled statement

a label can be used in a **`goto`** before its definition

labels have their own name space (they do not interfere wih other identifiers)

avoid labelled statements !

Expression Statements

an expression is a sequence of operators and operands that specifies a computation

a statement can be a single expression

```
int main () {
   10 + 2;
   return 0;
}
```

the **expression** is **evaluated**

its **value** is **discarded** after the `";"`

all **side-effects** are completed **before** the **next** statement is executed

statements can be function calls, assignments, definitions, ...

```
void f (int j) {}
int main () {
   f(i);
   int i = 10 + 2;
   return 0;
}
```

Problem with expression evaluation

if during the **evaluation** of an expression

the **result** is not **mathematically defined** or outside the **range** of representable values for its type

then the behavior is **undefined**

Compound Statements

a compound statement is enclosed by brackets **{** and **}**

```
int main () {
  int i;
  {
    int j = i;
    {
      int k = j;
    }
  }
}
```

a compound statement can be used whenever one statement is expected

```
#include <iostream>
void foo (int v) {
  if (v != 0)
    v = v + 12;
  std::cout << v << std::endl;
}
```

```
#include <iostream>
void foo (int v) {
  if (v != 0) {
    v = v + 12;
    std::cout << v << std::endl;
  }
}
```

with compound statements, you define nested blocks with local scopes

the `if` selection statements

`if` **(** *condition* **)** *statement*

```
if (x) {
  int i;
}
```

```
if (x)
  int i;
```

`if` **(** *condition* **)** *statement* **`else`** *statement*

```
if (int x = f()) {
  int x;
// error: redeclaration of x
} else {
  int x;
// error: redeclaration of x
}
```

where condition is implicitly converted to **`bool`**

`if` Statement

to which **`if`** is the **`else`** associated ?

```
if cond0
   if cond1
      f1 ();
   else
      f ();
```

```
if cond0
   if cond1
      f1 ();
else
   f ();
```

the **`else`** is associated with the **nearest un-elsed `if`**

Use blocks !

```
if cond0 {
   if cond1
      f1 ();
   else  f ();
}
```

the **switch** Selection Statement

switch (*condition*) *statement*

used to compare an **integral** variable to a list of integral values

the variable is compared, in **sequence**, to the values following the **case** label

when one matches : the computer executes the **case** part, then it continues ...

```
switch (value) {
case v0:
    ...
case v1:
    ...
default :
    ...
}
```

use **break** in **switch** selection statement when needed

```
#include <iostream>
int main () {
  using namespace std;
  int value;
  cout << "enter␣a␣value:␣";
  cin >> value;
  switch (value) {
  case 0:
    cout << "the␣value␣is␣0" << endl;
  case 1:
    cout << "the␣value␣is␣1" << endl;
  case 2:
    cout << "the␣value␣is␣2" << endl;
  default :
    std::cout << "the␣value␣is␣a␣value␣by␣default";
  }
}
```

```
$$ ./a.out
enter a value: 1
the value is 1
the value is 2
the value is a value by default
```

```
#include <iostream>
int main () {
  using namespace std;
  int value;
  cout << "enter␣a␣value:␣";
  cin >> value;
  switch (value) {
  case 0:
    cout << "the␣value␣is␣0" << endl;
    break;
  case 1:
    cout << "the␣value␣is␣1" << endl;
    break;
  case 2:
    cout << "the␣value␣is␣2" << endl;
    break;
  default :
    cout << "the␣value␣is␣a␣value␣by␣default";
  }
}
```

```
$$ ./a.out
enter a value: 1
the value is 1
```

while and do while (until) Iteration Statements

while (condition) statement

```
#include <iostream>
int main () {
  int x = 10;
  while (x) {
    std::cout << x << std::endl;
    x--;
  }
}
```

do statement **while (*expression*) ;**

```
#include <iostream>
int main () {
  int x = 10;
  do {
    std::cout << x << std::endl;
    x--;
  } while (x);
}
```

for iteration statements

for (*initialization* ; *continuation condition* ; *expression*)

```
#include <iostream>
int main () {
  for (int x = 10; x != 0; x--)
    std::cout << x << std::endl;
}
```

```
for (int i = 0; i < 6; i++) { ...}
for (int j = 0; j < 3; j++) { ...}
int i = 0;
for (; i < 12; ++i) {...}
for (;;++i) {...}
for (;;) {...}
```

Range-for statement *in c++11 only*Statement

for (*range* : *range initializer*) *statement*

to iterate through a range (à-la-python) :*for all x in v starting with v.begin() and iterating to v.end()*

a range is anything you can iterate through :

- like an STL-sequence defined by a begin() and **end()**
- so anything for which you define **begin()** and **end()**
- all standard containers
- std : :string, initializer-list, array, ...

```
#include <iostream>
int main () {
  int tab [10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  for (auto& x : tab) // notice the auto and the reference !
    x*=100;
  for (auto&& x : tab) // the move operator
    x*=100;
  for (const auto x: tab)
    cout << x << endl;
  for (const auto x : {10, 9, 8, 7, 6, 5, 4, 3, 2, 1})
    cout << x << '\n';
```

break Jump Statement

break causes termination of the smallest enclosing iteration-statement :

```
#include <iostream>
int main () {
  int i = 0;
  while (i < 10) {
    std::cout << i << ' ';
    if (i == 3)
      break;
    i+=1;
  }
}
```

```
$$ ./a.out
0 1 2 3
```

break is very useful for **switch** statements

continue Jump Statements

continue causes control to pass to the end of the smallest enclosing iteration-statement

```
#include <iostream>
int main () {
  for ( int i =10; i < 10; i+=1) {
    // if i is even, skip the print
    if ((i % 2) == 0)
      continue;
    std::cout << i << ' ';
  }
}
$$ ./a.out
1 3 5 7 9
```

```
#include <iostream>
int main () {
  int i = 1;
  while (i < 10) {
    // if i is even, skip the print
    if ((i % 2) == 0)
      continue;
    std::cout << i << std::endl;
    i+=1;
  }
}
$$ ./a.out
1
 infinite loop !
```

Functions

Function arguments are passed by copy

```
   stack 1         stack 2         stack 3         stack 4

+-----------+   +-----------+   +-----------+   +-----------+
|           |   |           |   |           |   |           |
|           |   +-----------+<  +-----------+<  |           |
|           |   | i      17 |   | i      29 |   |           |
+-----------+<  +-----------+   +-----------+   +-----------+< top
| j      17 |   | j      17 |   | j      17 |   | j      17 |
+-----------+   +-----------+   +-----------+   +-----------+
```

```cpp
void foo (int i) { // stack 2
   i = i + 12;      // stack 3
}
int main () {
  int j {17}; // stack 1
  foo(j);     // stack 4
  // j stays unchanged
}
```

stack 1 **`j`** is **automatically allocated** in the **stack**

stack 2 at the call of **`foo(j)`** **`i`** is automatically allocated in the stack with the value of **`j`**

stack 3 **`i`** is incremented by **`12`** but not **`j`**

stack 4 after exiting the function **foo**, **`i`** is automatically removed from the stack

- the **modification** of **`i`** is **local** to the function **`foo`**

Pointer arguments are also passed by copy

```
   stack 1         stack 2         stack 3         stack 4
+------------+  +------------+  +------------+  +------------+
|            |  |            |  |            |  |            |
|            |  +------------+< +------------+< |            |
|            |  | pi   --> j |  | pi -->   j |  |            |
+------------+< +------------+  +------------+  +------------+< top
| j       17 |  | j       17 |  | j       29 |  | j       29 |
+------------+  +------------+  +------------+  +------------+
```

```cpp
void foo (int* pi) {  // stack 2
   *pi = *pi + 12; // stack 3
}
```

```cpp
int main () {
   int j {17};  // stack 1
   foo(&j);   // stack 4
   // j has changed
}
```

stack 1 **`j`** is **automatically allocated** in the **stack**

stack 2 at the call of **`foo(j)`** **`pi`** is automatically allocated in the stack with the address of **`j`**

stack 3 the object pointed by **`pi`** (**`j`**) is incremented by **`12`**

stack 4 after exiting the function **foo**, **`pi`** is automatically removed from the stack

- in the function **`foo`** the **modification** of **`pi`** has done a **side effect** outside of **`foo`**

functions can be overloaded

overloading is giving the same name to several functions

their use is discriminated by the number and the types of the functions' arguments

```
void print (char) {}
void print (char, char) {}
void print (const char*) {}
int main () {
  print('c');
  print('c', 'a');
  print("hello");
}
```

functions Default Arguments

functions can have default arguments

```
void foo ( int  i = 1,  int  j = 2,  int  k = 3)  {}
int  main () {
                          //   i    j    k
  foo ();                 //   1    2    3
  foo (10);               //  10    2    3
  foo(10,  20);           //  10   20    3
  foo(10,  20,  30);      //  10   20   30
}
```

Order of Evaluation of expressions

the order of evaluation of function arguments in a function-call expression is unspecified

the order of evaluation of subexpressions (function calls) within expression is unspecified

the order of evaluation may changer when the same expression is evaluated again

operator **+** has a left-to-right associativity $a+b+c$ is $(a+b)+c$

```
int i;
int foo () { return i+=10; }
int bar () { return i-=30; }
int gee () { return i*=20; };
int main () {
  int r = foo() + bar() + gee();
}
```

function calls of **`foo`**, **`bar`** and **`gee`** can be done in any order

The notion of namespaces

namespaces : a technique for logical grouping

in a program, a **`namespace`** is a **declarative region**

`namespaces` are provided to group facilities that belong **together**

it is a way to **logically** group facilities

for example, the **standard-library** is defined in the namespace **`std`**

```
#include <iostream>
int main () {
  std::cout << "Hello World !" << std::endl;
}
```

namespaces : a technique to avoid name clashes

but above all it is a way to **avoid** name **clashes**

because you won't be the only one to call a function **`print`** or a variable **`n`** !

```
int n;
void print () {}
```

```
namespace my_library {
  int n;
  void print () {}
}
```

splitting namespaces

the definition of a **`namespace`** can be split (in the same translation unit or in several ones)

```
namespace my_library {
  int n;
}
```

```
namespace my_library {
  void print () {}
}
```

the members of a namespace are in the **same** scope

they refer to each other **without** special notation

```
namespace my_library {
  int n;
  void print () {
    n = n + 1;
  }
}
```

accessing namespaces facilities

but access from **outside** the namespace requires **explicit** notation (`std::cout`)

the global scope is a namespace referred to using `::`

a namespace can have an alternative names

```
namespace my_lib = my_library
```

`using` declarations

a **`using namespace` declaration** introduces all the names of a namespace (into the **declarative region** in which it appears)

```
#include <iostream>
using namespace std; // not so good !!
int main () {
  cout << "Hello␣World␣!" << endl;
}
```

but keep in mind that a global **`using`** namespace declaration can reintroduce names clashes

prefer local **`using`** declaration

```
#include <iostream>
int main () {
  using namespace std; //  really  better
  cout << "Hello␣World␣!" << endl;
}
```

using declarations

a **using** declaration can introduce a single name into the declarative region in which it appears :

```
#include <iostream>
int main () {
  using std::cout;
  cout << "Hello World !" << std::endl;
}
```

```
void f () {}

namespace A {
  void g() {}
  void f () {}
}
namespace X {
  using ::f;
  using A::g;
  void foo () {
    f ();   // global ::f ()
    g ();   // A::g()
    A::f ();
  }
}
```

inline specifier 1/2 *in c++11 only*

all **members** of an **`inline` namespace** are **automatically members** of the **enclosing namespace** (ex : *the current version of a function from a library*)

`foobar.h`

```
namespace foobar {
#include "foov1.h"
#include "foov2.h"
}
```

`foov1.h`

```
inline namespace foov1 {
  string version = "v1";
  void bar () {
    cout << "foov1::bar\n";
  }
}
```

`foov2.h`

```
namespace foov2 {
  std::string version = "v2";
  void bar () {
    cout << "foov2::bar\n";
  }
}
```

```
#include "foobar.h"
using namespace foobar;
int main () {
  cout << version << "\n";
  bar ();
}
```

```
v1
foov1::bar
```

inline specifier 2/2 *in c++11 only*

the **`inline`** specifier makes a declaration appear **as if** it had been declared in the enclosing namespace

`foobar.h`

```
namespace foobar {
#include "foov1.h"
#include "foov2.h"
}
```

`foov1.h`

```
namespace foov1 {
  string version = "v1";
  void bar () {
    cout << "foov1::bar\n";
  }
}
```

`foov2.h`

```
inline namespace foov2 {
  std::string version = "v2";
  void bar () {
    cout << "foov2::bar\n";
  }
}
```

```
#include "foobar.h"
using namespace foobar;
int main () {
  cout << version << "\n";
  bar();
}
```

```
v2
foov2::bar
```

Nested Namespaces

avoid writing confusing code !

```
namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; } // Outer::i
    int i;
    void g() { i++; } // Inner::i
  }
}
```

Unnamed Namespace

a namespace can be **anonymous**

it won't be known outside its local context

```
namespace {
  int n;
  void print () {}
}
```

there is no way of **naming** a member of an unnamed namespace from another translation unit

its members are only seen inside the whole file defining it

the outermost declarative region of a translation unit is also a namespace (the **global** `::` namespace)

Stroustrup's advice on **namespaces**

place every nonlocal name, except **`main()`**, in some namespace

avoid very short names for namespaces

use separate namespaces for interfaces and implementations

don't put a using-directive in a header file

introduction to user’s defined types

adding new structured types to your programs

create a new data type by grouping together inside a **`class`** declaration :

- declarations of **data** (called **data member**)
- **functions** working on those data (called **member functions**)

for example, to implement a type **`Integer`** with :

- an integer value
- a function to increment the value
- a function to decrement the value

```
class Integer {
  int value;
  void incr () { value+=1; }
  void decr () { value-=1; }
};  // the ';' is mandatory !
```

you can define member functions **inside** the class declaration

in a class definition [a], no member can be **declared elsewhere** must declare the **full set** of class's members (data and functions)

a. as opposed to **`namespace`**

a fast introduction to **`inline`** functions

you can **avoid** the **cost** of function calls : pushing the arguments on the stack and popping the arguments from the stack

by specifying a function as been **`inline`**

an **`inline`** function behaves like an ordinary function

an **`inline`** function is *expanded* in place during compilation

i.e. at compile-time, the call of an **`inline`** function is **replaced** by the function's body

the overhead of the function call is **eliminated**

for **`inline`** function no code is generated

type checking is performed on **`inline`** functions

defining member functions outside the class declaration

you can define member functions **outside** the class declaration by using the `::` scope operator

```
class Integer {
  int value;
  void incr ();
  void decr ();
};
inline void Integer::incr () {
  value+=1;
}
inline void Integer::decr () {
  value-=1;
}
```

defining a member function inside or outside the class is **not** equivalent

functions **defined inside** the class are **`inline`** by default

functions **defined outside** are **not `inline`** by default, you must specify them **`inline`**

scope of data members and member functions

the class declaration defines a type, its name (**`Integer`**) becomes the name of a new type in your program

a class declaration introduces the class name into the scope where it is declared

```
class Integer {
  int value;
  void incr () { value+=1; }
  void decr () { value-=1; }
};

int main () {
  Integer i;
}
```

the **`main`** function declares and defines an **`Integer`** object named **`i`**

what is happening in this code ?

either you chose this class definition

```
class Integer {
  int value;
  void incr () { value+=1; }
  void decr () { value-=1; }
};
```

or this one

```
class Integer {
  int value;
  void incr ();
  void decr ();
};
inline void Integer::incr () {
  value+=1;
}
inline void Integer::decr () {
  value-=1;
}
```

```
int main () {
  Integer i;
  i.value;      // error: int Integer::value is private within this context
  i.incr ();    // error: void Integer::incr () is private within this context
  i.decr ();    // error: void Integer::decr() is private within this context
}
```

Why does the compiler raise an error when you try to access a member of your class from the **`main`** ?

accessibility (class members are by default private)

by default, members of a class are **private** to that class

i.e. members are accessible only in the member functions of the class

you can see that **`Integer::incr`** can access the data member **`value`**

```
inline void Integer::incr () {
  value+=1;
}
```

but the **`main`** can't

```
int main () {
  Integer i;
  i.value;    // error: int Integer::value is private within this context
}
```

c++ allows you to **keep** some information **private** to your class

when you do not want users to access **implementation** details of your code

or when you do not want users to access some important **algorithm** in your code

accessibility can be changed (public)

to be seen outside, a **`class`** member must be made **`public`**

```
class Integer {
private:
   int value;
public:
  void incr () { value+=1; }
  void decr () { value-=1; }
};
int main() {
  Integer i;
  i.incr ();
  i.decr();
}
```

keep your implementation details **private** to you class (**`value`**)

give access to the outside to a **public interface** of your class (**`incr`**, **`decr`**)

if you replace the keyword **`class`** by **`struct`**, then members will be **`public`** by default

when implementing a class, always separ **implementation** details from **public interface**

naming the hidden object

a member function **knows** the object on which the function was **invoked**

the body of a member function is allowed to **access** the address of the object on which the function was invoked

this address is called **`this`**

because changing the address of **`this`** in a member functions has **no meaning**, it is forbiden by **`const`** :

inside member functions, **`this`** is a constant address

```
class Integer {
  int value;
  void incr () { this->value+=1; }
  void decr () { this->value-=1; }
};

int main () {
  Integer i;
// this is the constant adress of the object i
}
```

```
Integer const * this;
```

`Integer const * this;`

what is happening in this code ?

```
#include <iostream>
class Integer {
private:
  int value;
public:
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};
int main() {
  Integer i;
  i.incr();
  i.decr();
  i.print();
}
```

```
$$ g++ -std=c++11 integer.cxx -o integer
$$ integer
-1866067120
```

you **forgot** to **initialize** the data member **`value`** of the object **`i`** !

nevertheless your object has been **allocated** and **initialized** !

c++ has **implicitly** defined a **default** way to construct an object

initializing data members at declaration point *in c++11 only*

inside a class *in c++11 only* a data member can be **default-initialized** where it is declared

```
#include <iostream>
class Integer {
private:
  int value {};
public:
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};
int main() {
  Integer i;
  i.print();
}
```

```
$$ g++ -std=c++11 integer.cxx -o integer
$$ integer
0
```

the object **`i`** is default-initialized to the zero of the **`int`**

in older c++ versions, only constant static data members can be initialized at their declaration point

implementing constructors

to initialize differently, you can implement **constructors**

the constructors are **member** functions

the names of the constructors are the **name** of the class

in *in c++11 only*

```
#include <iostream>
class Integer {
private:
  int value;
public:
  Integer (int v) : value{v} {}
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};

int main() {
  Integer i {11};
  i.print ();
}
```

in older c++ versions

```
#include <iostream>
class Integer {
private:
  int value;
public:
  Integer (int v) : value(v) {}
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};
int main() {
  Integer i (12);
  i.print ();
}
```

Where are data members initialized ?

initializing data members inside constructor

in a constructor, data members can be initialized in a special **place** (it is called the constructor initialization-list)

inside the constructor **body**, data members can only be **assigned**

at the entry of the constructor body, the data members have already been **initialized** (either by you or by default)

in *in c++11 only*

```
#include <iostream>
class Integer {
private:
  int value {12};
public:
   Integer (int v) {
      print ();
      value = v;
      print ();
   }
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};
int main() {
  Integer i {11};
  i.print ();
}
12 11
```

Notice the use of the **initialization** at declaration point

what is happening in this code ?

```
#include <iostream>
class Integer {
private:
  int value;
public:
  Integer (int v) : value{v} {}
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};

int main() {
  Integer j;     // error: no matching function for call to Integer::Integer()
}
```

because **you** have specified a constructor (here the one taking **one** argument)

construction is now **under your** responsability

c++ will **no** longer implicitly generate a **default-constructor** (constructor without argument)

implementing a default-constructor for a class

you can use function's default arguments

in *in c++11 only*

```
#include <iostream>
class Integer {
private:
  int value;
public:
  Integer (int v = {}) : value{v} {}
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};

int main () {
  Integer i {};
  Integer j;
  Integer k {1};
}
```

you have now two constructors

when the argument is omitted, value is zero-initialized

implementing several constructors

when several constructors are needed, you can also use overloading

in *in c++11 only*

```
#include <iostream>
class Integer {
private:
  int value {};
public:
  Integer (int v) : value{v} {}
  Integer () : value{} {}
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};
int main () {
  Integer i {};
  Integer j;
  Integer k {1};
}
```

nevertheless it is better to avoid **redundant** code !

initializing at declaration point and in constructors

the constructor's initialization **overrides** the initialization at declaration point

```
class foo {
  int i {42};
};
int main () {
  foo f;    // f.i == 42
}
```

```
class foo {
  int i {42};
public:
  foo () : i{43} {}
};
int main () {
  foo f;    // f.i == 43
}
```

the two implementations are **not** completely equivalent

in the first case, the **default-constructor** is implicitly defined by the compiler and your code will **run faster !**

more generally, when you let the compiler define a member function : your code will **run much faster**

mixing initializing at declaration point and constructors

you can have initialization at declaration point **and** initialization in the constructor initialization-list

```
#include <iostream>
class foo {
  int number;
  int i {42};
public:
  foo (int n = {}) : number{n} {}
};
int main () {
  foo f1;
// f.i == 42 and f.number == 0
  foo f2 {11};
// f.i == 42 and f.number == 11
}
```

objects of user-defined type can be created in the three memory zones

```
class Integer {
private:
  int value;
public:
  Integer (int v = {}) : value{v} {}
  void incr () { value+=1; }
  void decr () { value-=1; }
};
Integer I;                      // static store
int main () {
  I.incr ();
  Integer i {11};               // stack
  i.incr ();
  Integer* pi {new Integer{42}};  // heap
  pi->incr();
}
```

notice : that the object **`I`** **exists** *before* the **`main`** function is called

notice that : members are accessed using **`.`** for **objects** and **->** for pointers

what is the value of **`I.value`** ?

Do you see a problem in this code ?

correcting the memory leak

the memory allocated in the heap with **`new`** has never been deleted by **`delete`**

at exit, you have a **memory leak** (the size of an **`Integer`**)

```
class Integer {
private:
  int value;
public:
  Integer (int v = {}) : value{v} {}
  void incr () { value+=1; }
  void decr () { value-=1; }
};
Integer I;
// static store
int main () {
  Integer i {11};                    // stack
  Integer* pi {new Integer{42}};     // heap
  delete pi;
}
```

this code is correct : there is no more a memory leak

what is happening in this code ?

```
class foo {
  int* pi {new int};
};
foo t[1000];
int main () {
}
```

each time an object of type **foo** is created, an **int** is allocated in the heap and never deleted !

you allocate an array of 1000 objects (reachable) of type **foo**

at exit your code has generated memory leaks

for example, on my computer, i forgot to delete 4000 bytes (4 per object of type **foo**)

Where can you call **delete** ? (we will see that latter)

class initialization at declaration point and unnamed class *in c++11 only*

```
#include<iostream> struct { short day, month, year; } defaultDate =
        {25, 12, 2015}; // unnamed class

class MyDate {
private:
  short day {defaultDate.day};
  short month {defaultDate.month};
  short year {defaultDate.year};
public:
  void print () {
    std::cout << day << '/' << month << '/' << year << std::endl;
  }
};
int main () {
  MyDate d;
  d.print ();
}
```

```
25/12/2015
```

each objects of type **`MyDate`** will have its **`day`**, **`month`**, and **`year`** data members, default-initialized

*in c++11 only*calling a constructor from a constructor

constructors can also be called in the initialization-list

```
class A {
private:
   int a;
public:
   A (int i) : a{i} {}
   A () : A(0) {}
};
```

using const inside classes

what is happening in this code ?

```cpp
#include <iostream>
class Integer {
private:
  int value;
public:
  Integer (int v = {}) : value{v}{}
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};

int main() {
  Integer i {11};
  i.incr ();
  i.print ();
  const Integer max_limit {1023};
  max_limit.print ();            // ERROR
}
```

what is happening in this code ?

```
#include <iostream>
class Integer {
private:
  int value;
public:
  Integer (int v = {}) : value{v}{}
  void incr () { value+=1; }
  void decr () { value-=1; }
  void print () { std::cout << value; }
};

int main() {
  Integer i {11};
  i.incr ();
  i.print ();
  const Integer max_limit {1023};
  max_limit.print ();                // ERROR
}
```

you don't want **`max_limit`** to be modified

you tell c++ that **`max_limit`** it is a constant

you try to **`print max_limit`**

so you try to call on a constant object, a member function that is allowed to modify the object pointed by **`this`** !

a compile-time error occurs ! and your code won't compile

using member functions for const objects

tell the compiler that a member function does **not** modify the object pointed to by **`this`** by declaring the function **`const`**

```
#include <iostream>
class Integer {
private:
  int value;
public:
  Integer (int v = {}) : value{v}{}
  void incr () { value+=1; }
  void decr () { value−=1; }
  void print () const { std::cout << value; }
  int to_integer () const { return value; }
};

int main() {
  Integer i {11};
  i.incr ();
  i.print ();
  const Integer max_limit {1023};
  max_limit.to_integer ();      // ok
  max_limit.print ();           // ok
}
```

every member functions that do not modify the object pointed by **`this`** must be declared **`const`** : it is a good programming style

it is a **good** programming style

`const` member functions

a **`const`** member function can be called for **`const`** objects and for variable objects

an ordinary member function can only be called for variable objects

a **`const`** member function can read but not write the object pointed to by **this**

the type of **`this`** in a **`const`** member function of class **`X`** is : **`const X * const this`**

[TP] a very simplified version of rational numbers

you **`rational`** number is represented by two integer values : a numerator **`num`** and a denominator **`denom`** (for example $-1/2$, $0/-10$, $-7/-14$, ...)

do **not** access data members directely but implement member functions to access them (to read their value)

do **not** implement a function to **normalize** (i.e. $7/-14$ is $-1/2$!)

implement a default-constructor that initializes the object $0/1$

implement a constructor with one argument that initializes the object $n/1$

implement a constructor with two arguments

implement a global function for equality test ($a/b == c/d$ if $a*d == b*c$)

implement a global function for addition ($a/b+c/d == ad+bc/db$) that returns an object of type rational

implement a function to compute a floating point approximation of your rational **`to_float`**

implement your class inside a personnal **`namespace`**

Can you code have a corrupted execution ?

Can you code have a corrupted execution ?

be careful of potential run-time errors

```
class rational {
private:
  int num;
  int denom;
public:
  rational (int n = {0}, int d = {1}): num{n}, denom{d} {}
  int numerator () const { return num; }
  int denominator () const { return denom; }
  float to_float () const {
    return numerator()/denominator();
  }
};
```

```
int main () {
  rational r1{1, 2};
  rational r2{1, 0};
  r2.to_float ();
}
```

```
Floating point exception (core dumped)
```

the execution is **aborted** !

integer division versus floating point division

```
#include <iostream>
int main () {
  int p = 1;
  int q = 0;
  std::cout << p/q << std::endl;
}
$$ ./a.out
Floating point exception (core dumped)
```

```
#include <iostream>
int main () {
  int p = 1;
  int q = 0;
  std::cout << (float)p/q << std::endl;
}
$$ ./a.out
inf
```

(due to the c++ standard sparseness on that subject) on **most** c++ compiler **implementations** :

an integer division by **zero** is a fatal error and the program is aborted

a floating point division by **zero** follows the *IEEE Standard for Floating-Point Arithmetic* that says (in section 7.3.0 about Division by zero) :

- ***The default result of divideByZero shall be an*** inf ***...***

what can you do ?

while implementating a function, if you find a problem that you cannot cope with : you **throw** an **exception** [a]

a. we will see the exception mechanism more in details

a function that wants to handle that kind of problem can indicate what to do by **catching** that **exception**

for example, when a user of your class `rational` try to create a rational with a null denominator, you can refuse by throwing an exception

throwing an exception in the rational constructor

for example, when a user of your class **`rational`** try to create a rational with a null denominator, you can refuse by throwing an exception

```cpp
#include <exception>
class rational {
private:
  int num;
  int denom;
public:
  rational (int n = {0}, int d = {1}): num{n}, denom{d} {
    if (denominator() == 0) throw std::exception();
  }
  int numerator () const { return num; }
  int denominator () const { return denom; }
  float to_float () const {
    return numerator()/denominator();
  }
};
```

```cpp
int main () {
  rational r2{1, 0};  // an exception is thrown !
  r2.to_float ();
}
```

```
$$ g++ -std=c++11 rational.cxx
$$ ./a.out
terminate called after throwing an instance of 'std::exception'
Aborted (core dumped)
```

catching the throwed exception

some caller function (here the **main**) must catch the exception to avoid the program to be aborted

```
#include <exception>
#include <iostream>
class rational {
private:
  int num;
  int denom;
public:
  rational (int n = {0}, int d = {1}): num{n}, denom{d} {
    if (denominator() == 0) throw std::exception();
  }
  int numerator () const { return num; }
  int denominator () const { return denom; }
  float to_float () const {
    return numerator()/denominator();
  }
};
```

```
int main () {
  try {
  rational r2{1, 0};   // an exception is thrown !
  r2.to_float ();
  } catch (std::exception e) {
    std::cout << " the exception is catched";
  }
}
```

```
$$ g++ -std=c++11 rational.cxx
$$ ./a.out
terminate called after throwing an instance of 'std::exception'
Aborted (core dumped)
```

[CORRECTION]of the [TP]on rational numbers 1/2

```
#ifndef RATIONAL_H
#define RATIONAL_H
#include <iostream>
#include <exception>
namespace vr {
  class rational {
  private:
    int num;
    int denom;
  public:
    rational (int n = {0}, int d = {1}): num{n}, denom{d} {
      if (denominator() == 0) throw std::exception();
    }
    int numerator () const {
      return num;
    }
    int denominator () const {
      return denom;
    }
    float to_float () const {
      // otherwise you get an integer division
      return (static_cast<float>(numerator()))/denominator();
    }
    void print () const {
      std::cout << numerator() <<'/'<< denominator();
    }
  };
  inline bool equal (const rational r1, const rational r2) {
    return r1.numerator()*r2.denominator() == r1.denominator()*r2.numerator();
  }
  inline rational addition (const rational r1, const rational r2) {
    return rational {r1.numerator()*r2.denominator() + r1.denominator()*r2.numerator(),
r1.denominator() * r2.denominator()};
  }
}
#endif
```

[CORRECTION]of the [TP]on rational numbers 2/2

```
#include "rational.h"
int main() {
  using namespace vr;
  rational n1;
  rational n2{12};
  rational n3{1, 2};
  rational n4{2, 4};
  n1.print(); n1.to_float();
  n2.print(); n2.to_float();
  n3.print(); n3.to_float();
  n4.print(); n4.to_float();
  rational a = addition(n3, n4);
  a.print(); a.to_float();
}
```

```
0/1 0
12/1 12
1/2 0.5
2/4 0.5
8/8 1
```

constructors

Constructors

remember that the **name** of the **constructor** is the **name** of the **class**

constructors cannot **return** a **value**

constructors are **automatically** called when an **object** is **defined**

a **constructor** is called a ***special*** member functions

initialization list of a constructor

it is a sequence of initialization appearing after the constructor arguments list and before the constructor body

```
class foo {
  int i;
  int j;
  int k {};
public:
  foo (int a) : i{a} {
    // i == a
    // k == 0
    // j == ???
    j = 12;      // assignment
  }
};
```

it is the **only** place where to perform **initialization** of data members

notice that inside the constructor body, you cannot perform **initialization** of data member only **assignment** !

when you enter the constructor body, initializations have **already** been **performed** (either implicitely or explicitely)

the initialization list of a constructor

it is the **only** place where to perform **mandatory** initializations

```
class bar { public: bar (int) {} };

class foo {
  const int i;
  float& f;
  bar b;
public:
  foo () {
  // ERROR uninitialized member 'foo::i' with 'const' type
  // ERROR uninitialized reference member 'foo::f'
  // ERROR no matching function for call to 'bar::bar()'
          candidates are bar::bar(int)
  }
};
```

what are the three initializations that must be done in the initialization-list ?

mandatory initializations of a constructor data members

```
class bar { public: bar (int) {} };

class foo {
  const int i;
  float& f;
  bar b;
public:
  foo (float& r) : i{12}, f{r}, b{0} {}
};
```

it is the **only** place where to initialize `const` data member

it is the **only** place where to initialize a data member passed by **reference**

it is the **only** place where to initialize a data member that do not have default constructor

it is the **only** place where to initialize a base class during derivation

default constructor

remember that the default constructor is **implicitely** defined by c++ in a class **without** constructor

```
class A {};
A a;          // ok call one time the implicitely -defined A::A()
A tab[100];   // ok call 100 times the implicitely -defined A::A()
```

C++ **stops** defining default constructor if you provide any constructor in a class

```
class A {
  A ( int )  {}
};
A a;            // nok no more A::A() to call (once)
A tab [100];    // nok no more A::A() to call (or a hundred times)
```

default constructors are mandatory for arrays of objects

default constructor is called when **arrays** of objects are initialized

```
#include <exception>
#include <iostream>
class rational {
private:
  int num, denom;
public:
   rational (int n = {0}, d = {1}) : num{n}, denom{d} {
     if (denominator() == 0) throw std::exception();
  }
   int numerator () const { return num; }
   int denominator () const { return denom;}
};
```

```
int main () {
  rational r [3];
  for (auto e : r)
     std::cout << c << "␣";
}
```

```
struct X {
  int a;
};

void foo (int X) {
  struct X* px = new struct X;
  px->a = X;
  delete px;
}

int main () {
  foo (12);
}
```

- inside **foo**, the integer argument **X** hides the structure **X**
- inside **foo**, **X** is thus the integer argument
- inside **foo**, to access the struct **X** : use the global scope-access
- inside **foo**, to access the struct **X** : use the **struct** qualifier

[TP] implement a simple single linked list

[TP]Implement a simple container : the **node** class

a **simple container** is **composed** of chained **cells** (called **node**)

→ 231 → 782 → 542 → 20 → -2

a **node** is a **cell** that **stores** :

- an integer **value**
- the address of the **following cell** (782 follows 231) (can be **nullptr**)
- and **contains** a function **print** : applied to a given **node**, it prints in order the value of the node and the values that follow, on the standard output stream

```
class node {
public:
   /* your constructor here */
   void print () const { /* your code here */}
private:
  /* your data members here */
};
```

```
int main () {
  node n1 {-2};
  node n2 {20, &n1};
  node n3 {542, &n2};
  node n4 {782, &n3};
  node n5 {231, &n4};
  n4.print ();
  n5.print ();
}
$$ g++ node.cpp -o node
$$ ./node
782 -> 542 -> 20 -> -2
231 -> 782 -> 542 -> 20 -> -2
```

[TP]Correction of the simple chained cell container

```
#include <iostream>
class node {
private:
  int value;
  node* next;
public:
  node (int v, node* n = {}) : value{v}, p_next{n} {}
  void print const () {
    std::cout << value << '␣';
    if (next != nullptr )
      next->print();
  }
};
```

```
int main () {
  node n1 {-2};
  node n2 {20, &n1};
  node n3 {542, &n2};
  node n4 {782, &n3};
  node n5 {231, &n4};
  n5.print ();
  std::cout << std::endl;
}
```

notice that :

- the data members are initialized in the constructor's initialization-list
- in the constructor, the default value of the **`p_next`** data member is **`nullptr`**
- the **`print`** function calls the **`print`** function of the next cell

[TP] the **node** class

[TP]Implement an integer stack : a **last in first out** container

```
class IntStack {
public:
  IntStack (int s);
  void push (int );
  int pop ();
  bool is_empty () const;
  bool is_full  () const;
  void print  () const;
private:
  /* your data members here */
};
int main () {
  IntStack s1 {3}; s1.print ();
  s1.push(1);      s1.print ();
  s1.push(2);      s1.print ();
  int e {s1.pop()};
  s1.push(3);      s1.print ();
}
$$ ./a.out
[ [
[ 1 [
[ 1 2 [
[ 1 3 [
```

`IntStack(int s)` :

- creates an array of **`s`** integer in dynamic memory
- keeps the size of the stack (needed for other functions)
- reminds that there is no element on this stack

`push(int e)` stores **`e`** at the top of the stack **`pop()`**

returns the top of the stack and consider it as removed from the stack **`is_full()`** (resp. **`is_empty`**)

returns **`true`** if stack is full (resp. empty) Refuse :

- to create a stack with a negative or null size
- to **`push`** on a full stack
- to **`pop`** an empty stack

a very small introduction to operator overloading

a small introduction on operator overloading

in c++ the **`int`** type together with operators such as **`+, -, *, /`** provides a (very restrictive) **implementation** of the mathematical **`integer`** type

c++ provides such facilities for **user-defined types**

you can overload a **set** of **operators**

it is only a **syntactic sugar**

it is just another syntax to make function calls

```
#include "my_integer.h"
int main () {
  int i, j;
  i = i + j;
  Integer I, J;
  I = I + J;
}
```

for example, overloading `operators+` on `Integer`

```
class Integer {
private:
  int val;
public:
  Integer (int v = {}) : val{v}{}
  int value () const { return val; }
};

inline Integer operator+ (const Integer i, const Integer j) {
  return Integer{i.value() + j.value()};
}
```

```
int main () {
  int i, j;
  i = i + j;
  Integer I, J;
  I = I + J;
}
```

notice the use of **`const`** function parameters

[TP] replace the **`equal`** and **`addition`** member functions of your class **`rational`** with **`operator ==`** and **`operator <`**

operator you can overload

you can **redefine** a lot of operators on your user-defined types :

new delete new[] delete[]							
+	−	∗	/	%	^	&	
|	~	!					
=	<	>	+=	−=	∗=	/=	
%=	^=	&=					
|=	<<	>>	>>=	<<=	==	!=	
<=	>=	&&					
||	++	−−	−>∗ ,		−>	[]	()

you **cannot** redefined several operators : `::`, `.`, `.*`, ...)

operator you can overload

you can also overload **types** (to create conversion operators)

```
class Integer {
private:
  int val;
public:
  Integer (int v = {}) : val{v}{}
  operator int () const { return val; }
};
```

```
int main () {
  Integer I{12};
  int i {I};
}
```

printing objects of user-defined types

to **print** an object of a user-defined type (like you print an object of a **built-in** type) : overload the **operator**<<

```
#include "integer"
int main () {
  Integer i {90};
  std::cout << i;
}
```

```
#include <iostream>
class Integer {
  friend std::ostream& operator<< (std::ostream& os, const Integer& n);
private:
  int value;
public:
  Integer (int v = {}) : value{v} {}
};
```

```
inline std::ostream& operator<< (std::ostream& os, const Integer& n) {
  os << '[' << n.value << ']';
  return os;
}
```

do not forget to return the output stream to **chain** the prints !

accessing private data members

declare the **`operator<<`** **`friend`** if it needs access private members of

it is better than implementing accessors

never put a data member **`public`**

you can **read** an object of some user-defined type like you read an object of a built-in type

```
#include <iostream>
int main () {
  int i;
  std::cout << "enter␣a␣number:␣";
  std::cin >> i;
  std::cout << i;
}
```

```
enter a number: 12
12
```

```
#include <iostream>
class Integer {
  friend std::istream& operator>> (std::istream&, Integer&);
private:
  int value;
public:
  Integer (int i = {}) : value{i} {}
};
std::istream& operator>> (std::istream& is, Integer rx) {
  is >> rx.value;
  return is;
}
int main () {
  Integer I;
  std:: cout << "enter␣an␣Integer␣value:␣";
  std::cin >> I;
}
```

[TP] overload the **`operator«`** on your classes

`nodelist`, **`intstack`**, **`rational`**,...

destructor

Reminder on `new` and `delete`

```
struct X {};
int main ()
  X* px = new X;
  delete px;
}
```

- **`new X`** allocates **storage** to hold one object of type **`X`**
- if the storage **allocation successes**, the **constructor** is invoked
- the constructor initializes the piece of storage
- **`delete px`** frees the **allocated** memory

```
struct X {};
int main ()
  X* tab = new X [10];
  delete [] tab;
}
```

- **`new X [10]`** allocates **storage** to hold an **array** of 10 objects of type **`X`**
- if the storage **allocation successes**, it calls 10 times the default constructor
- **`delete[]` frees** the **allocated array** of objects

[CORRECTION] of the integer stack 1/2

```
#include<iostream>
#include<exception>
namespace vr {
  class IntStack {
    friend std::ostream& operator<< (std::ostream& os, const IntStack&);
    public:
    IntStack (int s) : size{s}, top{0} {
      if (size <= 0) throw std::exception();
      tab = new int [size];
    }
    void push (int e) {
      if ( is_full ()) throw std::exception();
      tab[top] = e;
      top++;;
    }
    int pop () {
      if (is_empty()) throw std::exception();
      top--;
      return tab[top];
    }
    bool is_empty () const { return top == size; }
    bool is_full () const { return top == size; }
    private:
    int size;
    int top;
    int* tab;
  };
}
```

[CORRECTION] of the integer stack 2/2

```
namespace vr {
  inline std::ostream& operator<< (std::ostream& os, const IntStack& rs) {
    os << "[␣";
      for (int i = 0; i < rs.top; i++)
os << rs.tab[i] << "␣";
      os << "[\n";
    return os;
  }
}
```

```
int main () {
  using namespace vr;
  IntStack i{12};
  for (auto e :{1,2,3,4,5,6,7})
    i.push(e);
  std::cout << i;
  i.pop();
  std::cout << i;
}
```

```
[1 2 3 4 5 6 7 [
[1 2 3 4 5 6 [
```

what is Happening in this code ?

```
#include<iostream>
#include<exception>
class IntStack {
 private:
   int size;
   int top;
   int* tab;
 public:
  IntStack (int s);
  void push (int e);
   int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
void foo () {
  IntStack st {100};
}
int main () {
  foo ();
}
```

what is Happening in this code ?

```
#include<iostream>
#include<exception>
class IntStack {
 private:
  int size;
  int top;
  int* tab;
 public:
  IntStack (int s);
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
void foo () {
  IntStack st {100};
}
int main () {
  foo ();
}
```

in the function **`foo`**

- an **automatic** object **`st`** is allocated on the stack
- the **constructor** of **`IntStack`** is called
- 100 integers are **allocated** (contiguously) in the heap
- this memory is pointed to by **`st.tab`**

what is Happening in this code ?

```
#include<iostream>
#include<exception>
class IntStack {
 private:
  int size;
  int top;
  int* tab;
 public:
  IntStack (int s);
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
void foo () {
  IntStack st {100};
}
int main () {
  foo ();
}
```

in the function **`foo`**

- an **automatic** object **`st`** is allocated on the stack
- the **constructor** of **`IntStack`** is called
- 100 integers are **allocated** (contiguously) in the heap
- this memory is pointed to by **`st.tab`**

after exiting the function **`foo`** :

- **`st`** is automatically removed from the stack
- the memory pointed to by **`st.tab`** became inaccessible
- the 100 integers **remains** in the heap

your program has a serious memory leak problem !

Where does your memory leak come from ?

you have **allocated** a chunk of dynamic memory in the constructor

you have never freed it

not deleting a dynamically-allocated memory is considered as a programming error

it is not only a waste of space

for program meant to run for a **long time** : it can become a very serious **problem**

you need a **way** to **delete** the **dynamic memory** allocated in **constructor**

c++ introduces a **destructor** for this purpose

Destructor

in c++ proper **initializations** are **guaranted** by **constructors**

because **cleanup** is as **important** as **initialization**

c++ provides a **destructor** to **force cleanup**

in a `class`, the name of the **destructor** is the **name** of the class **prefixed** by a $\sim$

The **destructor** is **called automatically** when an automatic object **goes out of scope**

```
struct X {
  X () {}
  ~X() {}
};
void foo () {
  X x;          // here X::X() is called
}
int main () {
  foo ();
// here ~X::X() is called after the exit of foo
}
```

[TP] Add a destructor to your integer stack

```
class IntStack {
private:
   int size;
   int top;
   int* tab;
public:
  IntStack (int s);
  ~IntStack () {
    /* your code here ! */
  }
   void push (int e);
   int pop ();
   bool is_empty () const;
   bool is_full () const;
};
```

```
void foo () {
  IntStack st (100);
}
int main () {
  foo ();
}
```

after **exiting** the function **`foo`**

- the **automatic object `st`** becomes out-of scope
- it is automatically removed from the stack
- but just before, the destructor **`IntStack::~IntStack`** is **called**

the destructor of `class IntStack`

```
class IntStack {
private:
  int size;
  int top;
  int* tab;
public:
 IntStack (int s);
 ~IntStack () {
   delete [] tab;
 }
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full  () const;
};
```

```
int main () {
  IntStack* pst = new IntStack(20);
  delete pst;
  IntStack* tabst = new IntStack[30]
  delete [] tabst;
}
```

IntStack::~IntStack calls **delete[]** to free the dynamically-allocated array **tab**

pst is the address of a dynamically-allocated **IntStack** :

- **delete pst frees** this object
- but first it calls **IntStack::~IntStack** to free **pst.tab**
- then it frees the memory pointed to by **pst**

tabst is the address to a dynamically-allocated array of 30 objects of type **IntStack** :

- **delete [] tabst** frees this object
- but first it will call 30 times **IntStack::~IntStack** to free the 30 **tab**
- then it will free the array

Do we need a destructor for the `class Rational` ?

```
#include <exception>
#include <iostream>
class rational {
  int num;
  int denom;
public:
  rational (int n = {0}, int d = {1}) : num{n}, denom{d} {
    if (denominateur() == 0) throw std::exception();
  }
  int numerator () const { return num; }
  int denominateur () const { return denom;}
};
int main () {
  rational r;
  rational* r1 = new rational {11, 17};
  delete r1;
  rational* r2 = new rational[100];
  delete [] r2;
}
```

Do we need a destructor for the `class Rational` ?

```
#include <exception>
#include <iostream>
class rational {
  int num;
  int denom;
public:
  rational (int n = {0}, int d = {1}) : num{n}, denom{d} {
    if (denominateur() == 0) throw std::exception();
  }
  int numerator () const { return num; }
  int denominateur () const { return denom;}
};
int main () {
  rational r;
  rational* r1 = new rational {11, 17};
  delete r1;
  rational* r2 = new rational[100];
  delete [] r2;
}
```

no, we don't **need** a **destructor**

nothing special is done in the **`rational`** class only the initialization of two integers

so you don't need a special cleanup for objects of the class **`rational`**

beware ! adding an empty destructor will slow your code execution !

when do you need destructor ?

you only **need** a **destructor** when you do **something** in the class that needs to be ended properly in the **destructor** :

- **dynamic memory allocation**
- **file manipulation** : you open a file inside the constructor
- ...

destructor

a destructor is called whenever an object became out of scope (i.e. the end of the object's lifetime)

when a program terminates, the destructor is called for objects with static storage duration

```
#include "intstack.h"
IntStack S{100};
int main () {
  S.push(10);
}  // after the main exits
   // IntStack::~IntStack is called for the static object S
```

when the life of an object with automatic storage duration ends (stack unwinding), the destructor is called

```
#include "intstack.h"
int main () {
  if (true) {
    IntStack s{10};  // definition of the automatic object s
    s.push(10);
  }
   // after the block exits, the life of s ends
   // IntStack::~IntStack is called on s
}
```

when you call **`delete`** on object with dynamic storage duration, ... and some other situations.

in c++11 only deleted destructor

the destructor is a special member function

if **no** user-defined destructor is provided for a class type, the compiler will always **declare** a destructor as an `inline public` member of this class

if, for example, you don't want a destructor to be generated for some class (the class forbids destruction) *in c++11 only* you can `delete` it !

```
class foo {
public:
  ~foo() = delete;
};
```

```
int main () {
  foo f; // ERROR at compile-time
         // cannot delete => cannot create !
  foo* pf = new foo; // OK as long as you don't call delete on pf !
}
```

(still) the destructor is a special member function

(still) if **no** user-defined destructor is provided for a class type, the compiler will always **declare** a destructor as an **`inline public`** member of this class

if, for example, you want the destructor of some class to be **`private`** but you also want the compiler to generate the destructor

your code can run faster !

in c++11 only you can declare the destructor defaulted

```
class foo {
private:
  ~foo() = default;
};
```

```
int main () {
  foo f; // ERROR at compile-time
         // cannot delete => cannot create !
  foo* pf = new foo; // OK as long as you don't call delete
}
```

copy constructor

Remember the implementation of the integer stack

```
#include<iostream>
#include<cstddef>
#include<exception>
class IntStack {
 private:
  int size;
  int top;
  int* tab;
 public:
 IntStack (int s) : size{s}, top{0} {
    if (size <= 0)
       throw std::exception();
   tab = new int [size];     // HERE new[]
  }
  ~IntStack () {
   delete [] tab;            // HERE delete[]
  }
```

```
  void push (int e) {
     if ( is_full ())
        throw std::exception();
    tab[top] = e;
    top++;;
  }
  int pop () {
     if (is_empty())
        throw std::exception();
    top--;
    return tab[top];
  }
  bool is_empty () const {
    return top == size; }
  bool is_full () const {
    return top == size; }
};
}
```

because the constructor allocates a chunk of memory in the heap (dynamic memory)

the destructor must desallocated it !

what is happening in this code ?

```
class IntStack {
private:
   int size;
   int top;
   int* tab;
public:
  IntStack (int s);
  ~IntStack ();
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
int main () {
  IntStack s1 {10}
  IntStack s2 {s1};
  s1.push(10);
  s1.print ();
  s2.push(20);
  s1.print ();
  s2.print ();
}
```

```
the execution:
s1 [ 10 [
s1 [ 20 [    OUPS !!
s2 [ 20 [
and a run-time error with a double free or corruption !!
the execution is aborted
```

in the **`main`** function :

- **`s1`** is allocated on the stack
- **`s1`** is constructed by calling **`IntStack::IntStack(int)`**
- **`s2`** is allocated on the stack
- but how is **`s2`** constructed ?

How is s2 constructed ?

```
class IntStack {
private:
  int size;
  int top;
  int* tab;
public:
  IntStack (int s);
  ~IntStack ();
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
int main () {
  IntStack s1 {10}
  IntStack s2 {s1};
  s1.push(10);
  s1.print ();
  s2.push(20);
  s1.print ();
  s2.print ();
}
```

```
the execution:
s1 [ 10 [
s1 [ 20 [    OUPS !!
s2 [ 20 [
and a run-time error with a double free or corruption !!
the execution is aborted
```

in the **main** function :

- **s1** is constructed by calling **IntStack::IntStack(int)**
- **s2** is constructed by calling a constructor with an object of type **IntStack** as its argument
- so c++ must have implicitly defined such a function !
- but clearly here you don't like the way c++ has constructed your object **s2**

s2 has been constructed by copying **s1**

the problems occured because you let the compiler decide how to copy objects of **your** user-defined type

by default : it chooses a member-to-member copy

`s1.top` is copied in **`s2.top`** (ok same top)

`s1.size` is copied in **`s2.size`** (ok same size)

but **`s1.tab`** is copied in **`s2.tab`** ⇒ **`s1.tab`** and **`s2.tab`** are the same memory addresses

the two stacks are sharing the same integer array

the first error is that you **overwrite** the first element of **`tab`** (because **`s2.top == 0`**)

the second error is that the destructor is called **two** times (you destroy two times on the same integer array)

- first time when the destructor is called for **`s2`**
- second time when the destructor is called for **`s1`** (and in this order !)

adding a copy-constructor to your class

when it is **not** the behavior you want for a **copy**, you must define your **own** function

it is a c++ **special** function called a **copy-constructor**

a copy-constructor is a constructor that *takes* one argument : the **object to copy**

```
class IntStack {
private:
  int size;
  int top;
  int* tab;
public:
  IntStack (int s);
  IntStack (const IntStack /* 1 */);
  ~IntStack ();
```

```
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
int main () {
  IntStack s1 {10};   // IntStack::IntStack{10}
  IntStack s2 {s1};   // IntStack::IntStack{s1}
}
```

you really don't need to modify the argument in the copy-constructor : you must pass it **`const`**

1 How do you pass the argument ?

the prototype of the copy-constructor

```
class IntStack {
private:
  int size;
  int top;
  int* tab;
public:
  IntStack (int s);
  IntStack (const IntStack &);   // HERE !!
  ~IntStack ();
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
class foo {
  public:
    foo();                 // default-constructor
    ~foo();                // destructor
    foo(const foo&);  // copy-constructor
};
```

in a copy-constructor, you pass the argument :

- by **reference** of course ! because you are defining the copy ! and you must avoid copying the argument of such a function ! [a]
- **`const`** to avoid modifying it by mistake

a. Notice that you need a way to pass the argument, that does not copy it ! and it is the reason why the reference has been introduced in c++

what is happening in this code ?

```cpp
class IntStack {
private:
   int size;
   int top;
   int* tab;
public:
  IntStack (int s);   // NO COPY-CONSTRUCTOR
  ~IntStack ();
  void push (int e);
   int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```cpp
int main () {
  IntStack s1 {10};
  {                   // entering a block
    IntStack s2 {s1};
  }                   // exiting the block
  s1.push(10);
}
```

what is happening in this code ?

```
class IntStack {
private:
  int size;
  int top;
  int* tab;
public:
  IntStack (int s);   // NO COPY-CONSTRUCTOR
  ~IntStack ();
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full  () const;
};
```

```
int main () {
  IntStack s1 {10};
  {                  // entering a block
    IntStack s2 {s1};
  }                  // exiting the block
  s1.push(10);
}
```

the object **`s1`** is local to the **`main`** function

the object **`s2`** is local to an inner scope

`s2` is constructed by a member-to-member of **`s1`**

⇒ **`s1.tab`** and **`s2.tab`** refer to the same integer array in the heap

at the exit of the inner block :

- **`s2`** is destructed
- ⇒ so is **`s2.tab`**
- ⇒ so is **`s1.tab`**

`s1.push(10)` accesses an already deleted memory ! it is a serious memory problem

Same problem occurs

when an object of type **IntStack** is passed by copy in a function

```
#include "iostream"
// WITHOUT COPY-CONSTRUCTOR
void foo (IntStack s) {}
int main() {
  IntStack s1 {1000};
  foo (s1);
  si.push(42);
}
```

when an object of type **IntStack** is returned by copy from a function

```
IntStack bar () {
  return 12;
}
int main() {
  IntStack s2 {bar ()};
  s2.push(20);
}
```

you end up in an **Aborted (core dumped)** problem

[TP] Add a copy-constructor to your integer stack

```
#include<iostream>
#include<exception>
class IntStack {

private:
  int size;
  int top;
  int* tab;

public:
 IntStack (int s);
 ~IntStack ();
 IntStack (const IntStack& ri) {
   /* your code here ! */
 }
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
int main () {
  IntStack s1 (1000);
  { // entering a block
    IntStack s2 = s1;
  } // exiting a block
  s1.push(10);
}
```

- **`IntStack::IntStack(const IntStack&)`** is **called** to construct **`s2`** by copying **`s1`**
- when the block is exited, **`s2`** has became **out-of** scope and has been destroyed by a call to **`IntStack::~IntStack()`**

Adding a copy constructor for the `intstack class`

```
#include<exception>
class IntStack {
 private:
  int size;
  int top;
  int* tab;
 public:
 IntStack (int s) : size{s}, top{0} {
     if (size <= 0)
        throw std::exception();
    tab = new int [size];
  }
  ~IntStack () {
    delete [] tab;
  }
  IntStack (const IntStack& ri );
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
inline IntStack::IntStack (const IntStack& ri) :
  size{ri.size},
  top{ri.top},
  tab{new int[size]} {
  for (int i = 0; i < top; ++i)
    tab[i] = ri.tab[i];
}
inline bool IntStack::is_empty () const {
  return top == 0;
};
int main () {
  IntStack s1{10};
  IntStack s2{s1};
  s1.push(10);
  s1.print ();
  s2.push(20);
  s1.print ();
  s2.print ();
}
```

if you don't want a type to be copied, declare its copy-constructor **= delete**

you will forbit any copy of an object of this type

```
class IntStack {
 private:
  int size;
  int top;
  int* tab;
 public:
 IntStack (int s);
  ~IntStack ();
  IntStack (const IntStack& ri) = delete;
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
IntStack foo () { return 10; }
void bar (IntStack) {}
int main () {
  IntStack s1{10};
  IntStack s2{s1}; // ERROR
  IntStack s3 { foo() }; //ERROR
  bar(s2); // ERROR
}
```

you have three errors : *use of deleted function 'IntStack : :IntStack(const IntStack&)'*

in c++11 only preventing pass-by-value

```
class X {
public:
  X ()  {}
  X (const X&) = delete;
};
void foo (X) {}
X bar () {return X{};}
int main () {
  X x1;
  X x2 {x1};      // ERROR use of deleted function X::X(const X&)
  foo (x1);       // ERROR use of deleted function X::X(const X&)
  X x3 {bar ()};  // ERROR use of deleted function X::X(const X&)
}
```

Do we need a copy-constructor for the `class rational` ?

```cpp
#include <exception>
#include <iostream>
class rational {
private:
  int num, denom;
public:
    rational (int n = {0}, d = {1}) : num{n}, denom{d} {
      if (denominator() == 0) throw std::exception();
   }
    int numerator () const { return num; }
    int denominator () const { return denom;}
};
```

we don't **need** a **copy constructor** for the **`class rational`**

never define an empty copy-constructor : let the compiler generate it ! the default copy-constructor will be more efficient

for execution time, it is really better to let c++ **copy** two **integers**

We don't need a copy constructor for all classes

```
class IntStack {
 private:
  int size;
  int top;
  int* tab;
 public:
 IntStack (int s);
  ~IntStack ();
  IntStack (const IntStack& ri );
  void push (int e);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
};
```

```
class X {
public:
  IntStack st {120};
};
```

```
int main () {
  X x1;
  X x2 {x1};
}
```

`x2` is allocated on the stack and constructed by copy of **`x1`**

because no copy-constructor is defined in class **`X`**, a member-to-member copy is done to copy **`x1`**

the copy-constructor of **`IntStack`** is called for the copy of **`x1.st`**

we don't **need** a **copy constructor** for the **`class X`**

what *, do you think, will happen if you add a copy constructor to the class X ?*

implement a copy-constructor for the class X

but remember that it is really better to let the compiler generate the default copy-constructor for a class when it is trivial

```
class IntStack {
public:
   IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
class X {
  IntStack my_stack {120};
  X (const X&);
};
inline X::X (const X& rx)
 /* your code here */
```

```
int main () {
  X x1;
  x1.push(5);
  X x2 {x1};
  x2.push(10);
}
```

implement a copy-constructor for the class X

but remember that it is really better to let the compiler generate the default copy-constructor for a class when it is trivial

```
class IntStack {
public:
   IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
class X {
  IntStack st {120};
  X (const X& rx);
};
inline X::X (const X& rx) : st{rx.st}{}
```

```
int main () {
  X x1;
  x1.push(5);
  X x2 {x1};
  x2.push(10);
}
```

you get the compile-time error *no matching function for call to 'X : :X()' at X x1 ;*

what is happening here ?

implement a copy-constructor for the class **X**

because you add a copy-constructor, which is a constructor, the compiler will not generate a default-constructor for the class **X**

you must add one, because the default version is the good one, you use **default**

```
class IntStack {
public:
   IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full  () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
class X {
public:
  IntStack st {120};
  X () = default;
  X (const X& rx);
};
inline X::X (const X& rx) : st{rx.st}{}
```

```
int main () {
  X x1;
  x1.push(5);
  X x2 {x1};
  x2.push(10);
}
```

assignment operator

what is happening in this program ?

```
class IntStack {
public:
  IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
int main () {
  IntStack s1{100};
  { // an inner block
    IntStack s2 {20};
    s2.push(17);
    s1 = s2;
  }
  s1.push(12);
}
```

what is happening in this program ?

```
class IntStack {
public:
  IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
int main () {
  IntStack s1{100};
  { // an inner block
    IntStack s2 {20};
    s2.push(17);
    s1 = s2;
  }
  s1.push(12);
}
```

s1 is constructed :**s1.tab** is dynamically-allocated

what is happening in this program ?

```
class IntStack {
public:
  IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
int main () {
  IntStack s1{100};
  { // an inner block
    IntStack s2 {20};
    s2.push(17);
    s1 = s2;
  }
  s1.push(12);
}
```

`s1` is constructed :**`s1.tab`** is dynamically-allocated

inside the inner block :

- **`s2`** is constructed : **`s2.tab`** is dynamically-allocated
- **`s2`** is assigned to **`s1`**
- because you **let** the compiler **decide how to do** the assignment : it performs a member-to-member assignment :
  - **`s1.size`** became an integer copy of **`s2.size`** (20)
  - **`s1.top`** became an integer copy of **`s2.top`** (1)
  - the address **`s1.tab`** became a (pointer) copy of **`s2.tab`**

what is happening in this program ?

```
class IntStack {
public:
  IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  void push (int);
   int pop ();
  bool is_empty () const;
  bool is_full  () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
int main () {
  IntStack s1{100};
  { // an inner block
    IntStack s2 {20};
    s2.push(17);
    s1 = s2;
  }
  s1.push(12);
}
```

`s1` is constructed :**`s1.tab`** is dynamically-allocated

inside the inner block :

- **`s2`** is constructed : **`s2.tab`** is dynamically-allocated
- **`s2`** is assigned to **`s1`**
- because you **let** the compiler **decide how to do** the assignment : it performs a member-to-member assignment :
  - **`s1.size`** became an integer copy of **`s2.size`** (20)
  - **`s1.top`** became an integer copy of **`s2.top`** (1)
  - the address **`s1.tab`** became a (pointer) copy of **`s2.tab`**

at the exit of the inner block :

- **`s2`** (an automatic variable) became out-of-scope
- **`s2`** is removed from the stack
- the destructor of **`IntStack`** is called on **`s2`**
- **`s2.tab`** is deleted : **so** is **`s1.tab`** ! ! !

two problems occur :

- **a memory leak** : the initialy allocated **`s1.tab`** is lost
- **a memory problem** : you access **`s1.tab`** that has been deleted

Assignment Operator

tell c++ how to perform the **assignment** on user-defined types by defining the assignment operator

the prototype of a assignment operator is :

```
class X {
public:
  T& operator= (const T& rt) {
   /* the assignment operator code here */
  }
};
```

```
class X {
public:
  T& operator= (const T&);
};
inline T& T::operator= (const T& rt) {
   /* the assignment operator code here */
}
```

pass the argument :

- by reference, you are defining the assignment, avoid copying the argument !
- `const` to avoid modifying the argument by mistake

[TP]Add an assignment operator to IntStack

```
class IntStack {
public:
  IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  IntStack& operator= (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
  int get_size () const;
private:
  int size;
  int top;
  int* tab;
};
inline IntStack& IntStack::operator= (const IntStack& ri) {
  /* your code here */
}
```

the assignment operator for the `IntStack class`

```
class IntStack {
public:
  IntStack (int);
  ~IntStack ();
  IntStack (const IntStack&);
  IntStack& operator= (const IntStack&);
  void push (int);
  int pop ();
  bool is_empty () const;
  bool is_full () const;
  int get_size () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
inline IntStack& IntStack::operator= (const IntStack& ri) {
  if (this != &ri) { // test for self assignment !
    size = ri.size;
    top = ri.top;
    delete [] tab;            // delete the old tab
    tab = new int[size];
    for (int i = 0; i < top; ++i)
      tab[i] = ri.tab[i];
  }
  return *this;               // to chain assignment
}
```

notice in the assignment operator :

- the test for **self** assignment
- the return value to **chain** assignment

Chaining assignments

if you want to chain assignments (like for built-in types) :

```
int main () {
  int i{10}, j{12}, k{34}, l{21};
  i = j = k = l = 12;
}
```

```
int main () {
  IntStack i{10}, j{12}, k{34}, l{21};
  i = j = k = l = 12;
      // is equivalent to:
  i.operator=(j.operator=(k.operator=(l.operator=(IntStack {12}))));
}
```

the `operator=` function must :

- be declared to return a reference to the same type
- the assignment operator must return `*this`

always check for **self-assignment** in assignment operators !

what *will happen if you do not test for self-assignment in the assignment operator of IntStack ?*

```
IntStack& IntStack::operator= (const IntStack& ri) {
  size = ri.size;
  top = ri.top;
  delete [] tab;
  tab = new int[size];
  for (int i = 0; i < top; ++i)
    tab[i] = ri.tab[i];
  return *this;
}
```

```
int main () {
  IntStack s {100};
  s = s;
    // is equivalent to:
  s.operator={s};
}
```

in the `operator=` function :

- `*this` and `ri` refer to the same object `s`
- when you delete `tab`, you also have deleted `ri.tab`
- when you access `ri.tab` inside the `for`, you access an already deleted memory zone

Preventing assignment - *in c++11 only*

in c++11 the **= delete** specifier can be used to prohibit calling the function

it is used to explicitly disable certain features, for example to make a type non-assignable :

```
class X {
public:
  X () {}
  X& operator= (const X&) = delete;
};
int main () {
  X x1;
  X x2;
  x2 = x1;        // nok:  error: use of deleted function X& X::operator=(const X&)
}
```

and your code won't compile !

Do we need an assignment operator for the `class Rational` ?

```
#include <exception>
class rational {
public:
  rational (int n = 0, int d = 1) : num{n}, denom{d} {
    if (denominator() == 0) throw std::exception();
  }
  int numerator () const   { return num; }
  int denominator () const { return denom;}
private:
  int num, denom;
};
```

we don't **need** an assignment operator for the **`class Rational`**

for execution time, it is really better to let c++ **assign** two **integers** : the c++ default assignment operator will be much more efficient than yours on trivial objects !

Assignment Operators for `X` and `Y` ? be careful !

```
class A {
public :
  A () {}
  A& operator= (const A& ry) {}
};
```

```
class B {
  A y;
public :
  B () {}
};
```

```
class C {
  A y;
public :
  C () {}
  C& operator= (const C& r) {}
};
```

```
class D {
  A y;
public :
  D () {}
  D& operator= (const D& r) {
    y = r.y;        // notice !!
  }
};
```

```
int main () {
  B b1, b2;     // call A::A then B::B for b1 and for b2
  b1 = b2;      // calls A::operator=

  C c1, c2;
  c1 = c2;      // calls C::operator= only

  D d1, d2;
  d1 = d2;      // calls A::operator= then D::operator=
}
```

assignment operator

most of the time, the code of copy-constructor and the code of the assignment operator are quite similar, the solution is :

- to implement a private function that **clone** the object
- to implement a private function that **reset** the object
- to call these functions in the copy constructor, assignment operator and the destructor

in user-defined type :

- if you have implemented a destructor, you surely need a copy constructor
- if you have implemented a copy constructor, you surely need an assignment operator

Move Constructor *in c++11 only*

transfering a value from one place to another 1/4

```cpp
#include "intstack.h"
int main () {

  IntStack s2 {200};

  IntStack s1 = s2;

  IntStack* ps3 = &s2;
}
```

```cpp
#include "intstack.h"
int main () {

  IntStack s1 {100};

  IntStack s2 {200};

  s1 = s2;

  IntStack* ps3 = &s2;
}
```

you are transfering an object from one place **`s2`** to another **`s1`**

at the end of the copy or assignment, **`s1`** and **`s2`** will be equal to the value of **`s2`** before the assignment

because **`s2`** can still be used at the end of the assignment (refered to), it must be **well-formed**

notice that **`s2`** is a **left-value**

- it can be the left-hand side of an assignment **`s2 = ...;`**
- it refers to a memory location
- you can reach the address of that location using the & (address-of) operator

transfering a value from one place to another 2/4

what special functions (constructor, copy-constructor, assignment, destructor) have been called and in which order ?

```
#include "intstack.h"
int main () {
  IntStack s2 {200};
  IntStack s1 = s2;
  IntStack* ps3 = &s2;
}
```

⇕

```
IntStack::IntStack {200}   (s2)
IntStack::IntStack {s2}    (s1)
IntStack::~IntStack        (s1)
IntStack::IntStack {s2}    (s2)
```

```
#include "intstack.h"
int main () {
  IntStack s1 {100};
  IntStack s2 {200};
  s1 = s2;
  IntStack* ps3 = &s2;
}
```

⇕

```
IntStack::IntStack {100}   (s1)
IntStack::IntStack {200}   (s2)
s1.IntStack::operator={s2}
IntStack::~IntStack        (s2)
IntStack::~IntStack        (s1)
```

transfering a value from one place to another 3/4

```cpp
#include "intstack.h"
int main () {

  IntStack s1 = IntStack{300};
}
```

```cpp
#include "intstack.h"
int main () {

  IntStack s1 {100};

  s1 = IntStack{300};
}
```

in this code, the value you affect to **`s1`** is not **accessible**

it is not a left-value, it is a **temporary** value that does not exist any more after the assignment !

you do not need the temporary (resulting of **`IntStack`**{300}) to have a special value after the assignment

the temporary resulting of **`IntStack`**{300} only need to be **well-formed** because it will be destroyed

what special functions (constructor, copy-constructor, assignment, destructor) have been called and in which order ?

```
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

⇕

```
IntStack::IntStack {300}       (Temp.)
IntStack::IntStack {Temp.}     (s1)
IntStack::~IntStack            (Temp.)
IntStack::~IntStack            (s1)
```

```
#include "intstack.h"
int main () {
  IntStack s1 {100};
  s1 = IntStack{300};
}
```

⇕

```
IntStack:IntStack {100}        (s1)
IntStack::IntStack {300}       (Temp.)
s1.IntStack::operator= {Temp}
IntStack::~IntStack            (Temp.)
IntStack::~IntStack            (s1)
}
```

Note that to test this kind of code, you might need to pass an option to your compiler to avoid *copy ellision optimisation*[a]

```
$$ g++ -std=c++11 -fno-elide-constructors my_file.cxx
```

a. a **copy-elision** is a compiler optimization technique that eliminates unnecessary copying of objects

So what is happening here ?

```cpp
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

```cpp
#include "intstack.h"
int main () {
  IntStack s1 {100};
  s1 = IntStack{300};
}
```

a temporary object is created (**`IntStack`**{300})

it is copied inside an other object (**`s1`**)

then it is destroyed

in such a situation, a better behavior will be to :

- to simply **move** (slide) the content of the temporary object inside the other object **`s1`**
- to left the temporary object empty but well-formed enought to be destroyed correctly
- so as to avoid the overhead of the copy and the destruction of a non-empty object

such temporary object is called a right-value i.e. a value that cannot be assigned

it is the **purpose** of the **move-operator** (&&) *in c++11 only*and the **move-constructor**

examples of left-values and right-values

```
int& foo();
int bar();

int main() {

  int i = 42;           // i is a left-value
                        // 42 is a right-value

  int& ri = i;          // i is a left-value
  int* pi = &ri;        // ri is a left-value

  foo() = 42;           // foo() is a left-value
  int* pf = &foo();     // foo() returns a left-value

  int j = bar();        // j is a left-value
                        // bar() is a right-value

  int* p2 = &bar();   // ERROR left-value required as unary '&' operand
}
```

adding a move-construction to your `IntStack` 1/2

the **move-constructor** constructor will only be used by the compiler if the *moved* object cannot be used again !

```
class IntStack {
public:
   IntStack (int s);
   ~IntStack ();
   IntStack (const IntStack& ri);

   IntStack (IntStack&& ri);   // HERE !!

   IntStack& operator= (const IntStack&);
   void push (int e);
    int pop ();
   bool is_empty () const;
   bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
inline IntStack::IntStack (IntStack&& ri) :

     // the initialisation -list

   {

     // the move-constructor's body

    }
```

```
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

How will you program the move-constructor ?

adding a move-construction to your `IntStack` 1/2

the **move-constructor** constructor will only be used by the compiler if the *moved* object cannot be used again !

```
class IntStack {
public:
   IntStack (int s);
   ~IntStack ();
   IntStack (const IntStack& ri);

   IntStack (IntStack&& ri);   // HERE !!

   IntStack& operator= (const IntStack&);
   void push (int e);
   int pop ();
   bool is_empty () const;
   bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
inline IntStack::IntStack (IntStack&& ri) :

     // the initialisation -list

   {

     // the move-constructor's body

   }
```

```
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

How will you program the move-constructor ?

- in the initialisation-list : you **move** the containt of the argument (**`ri`**) in **`this`**
- in the move-constructor : you *empty* the argument
- Note that the argument cannot be **`const`**

adding a move-copy constructor to your `IntStack` 2/2

the **move-constructor** constructor will only be used by the compiler if the *moved* object cannot be used again !

```
class IntStack {
public:
   IntStack (int s);
   ~IntStack ();
   IntStack (const IntStack& ri);

   IntStack (IntStack&& ri);   // HERE !!

   IntStack& operator= (const IntStack&);
   void push (int e);
    int pop ();
   bool is_empty () const;
   bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
inline IntStack::IntStack (IntStack&& ri) :
      size(ri.size),
      top(ri.top),
      tab(ri.tab)
   {
      ri.size = 0;
      ri.top = 0;
      ri.tab = std::nullptr;
   }
```

```
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

How will you program the move-constructor ?

- in the initialisation-list : you **move** the containt of the argument (**`ri`**) in **`this`**
- in the move-constructor : you *empty* the argument
- Note that the argument cannot be **`const`**

adding a move-assignment to your `IntStack` 1/2

```
class IntStack {
public:
   IntStack (int s);
   ~IntStack ();
   IntStack (const IntStack& ri );
   IntStack& operator= (const IntStack&);

   IntStack& operator= (IntStack&& ri );     // HERE !!

   void push (int e);
    int pop ();
   bool is_empty () const;
   bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
inline IntStack& IntStack::operator= (IntStack&& ri)
  {

     // the move-assignment's body

  }
```

```
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

How will you program the move-assignment ?

adding a move-assignment to your `IntStack` 1/2

```
class IntStack {
public:
   IntStack ( int  s);
   ~IntStack  ();
   IntStack  (const IntStack&  ri );
   IntStack& operator= (const IntStack&);

   IntStack& operator= (IntStack&& ri );      // HERE !!

   void push (int  e);
    int  pop ();
   bool is_empty () const;
   bool  is_full   ()  const;
private:
  int  size;
  int  top;
  int* tab;
};
```

```
inline  IntStack& IntStack::operator= (IntStack&& ri)
   {

      // the move-assignment's body

   }
```

```
#include "intstack.h"
int  main () {
  IntStack  s1 = IntStack{300};
}
```

How will you program the move-assignment ?

- the two objects exist
- you check for a potential self-affectation
- you must simply swap their content !
- and you return **`*this`**

adding a move-assignment to your `IntStack` 2/2

```cpp
#include <algorithm>  // for the std::swap function
class IntStack {
public:
   IntStack (int s);
   ~IntStack ();
   IntStack (const IntStack& ri);
   IntStack& operator= (const IntStack&);

   IntStack& operator= (IntStack&& ri);     // HERE !!

   void push (int e);
    int pop ();
   bool is_empty () const;
   bool is_full () const;
private:
  int size;
  int top;
  int* tab;
};
```

```cpp
inline IntStack& IntStack::operator= (IntStack&& ri) {
   if (this != &ri) {
      std::swap(size, ri.size);
      std::swap(top, ri.top);
      std::swap(tab, ri.tab);
   }
   return *this;
}
```

```cpp
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

How will you program the move-assignment ?

adding a move-assignment to your `IntStack` 2/2

```
#include <algorithm>  // for the std::swap function
class IntStack {
public:
   IntStack (int s);
   ~IntStack ();
   IntStack (const IntStack& ri);
   IntStack& operator= (const IntStack&);

   IntStack& operator= (IntStack&& ri);     // HERE !!

   void push (int e);
    int pop ();
   bool is_empty () const;
   bool is_full  () const;
private:
  int size;
  int top;
  int* tab;
};
```

```
inline IntStack& IntStack::operator= (IntStack&& ri) {
   if (this != &ri) {
      std::swap(size, ri.size);
      std::swap(top, ri.top);
      std::swap(tab, ri.tab);
   }
   return *this;
}
```

```
#include "intstack.h"
int main () {
  IntStack s1 = IntStack{300};
}
```

How will you program the move-assignment ?

- the two objects exist
- you check for a potential self-affectation
- you must simply swap their content !
- and you return **`*this`**

example of move-construction and move-assignment

move will only be used when the *moved* object cannot be used again

```cpp
class X {
public:
  X ()  {}
  ~X ()  {}

  X (const X&);
  X (X&&);

  X& operator= (const X&);
  X& operator= (X&&);
};
```

```cpp
X foo  (X var)  {
  return  var;
}
```

```cpp
int  main () {
  X f1;                    // construction of the object f1
  f1  = function(f1);      // copy-construction of f1 in var
                           // move-construction of var in a temporary
                           // move-assigment of the temporary in f1
                           // destruction of var
                           // destruction of the temporary
}
```

Note that : *the copy-construction of var in the temporary* and *the copy-assignment of the temporary in f1* have been replaced by their **move** counterpart

a matched set of constructors, destructors and copy/move operations

if your **destructor** performs **nontrivial tasks** : the class surely needs to implement the **full set of functions**

```
class X {
  public:
  X (Sometype);
  X ();                    // default constructor
  ~X ();                   // destructor (clean up)
  X (const X&);            // copy constructor
  X (X&&);                 // move constructor
  X& operator= (const X&); // copy assignment (clean up target and copy)
  X& operator= (X&&);      // move assignment (clean up target and move)
  // ...
};
```

overloading functions on left-value and right-value argument

move can also be done for normal functions !

```
#include "intstack.h"
void Fct (IntStack& s)  {}   // used when Fct is called with a left-value
void Fct (IntStack&& s) {}   // used when Fct is called with a right-value

IntStack bar (); // a function returning a right-value
IntStack& foo (); // a function returning a left-value

int main () {
  IntStack obj{12};
  Fct (IntStack{11});  // IntStack{11} is a right-value: Fct (IntStack &&) is called
  Fct (obj);           // obj is a left-value:           Fct (IntStack &) is called
  Fct (bar ());        // bar() is a right-value:        Fct (IntStack &&) is called
  Fct (foo ());        // foo() is a left-value:         Fct (IntStack &) is called
}
```

example of a lot of copy for a simple swap !

```cpp
class IntStack {
public:
  IntStack ( int );

  IntStack (const IntStack&);
  IntStack (IntStack&&);

  IntStack& operator= (const IntStack&);
  IntStack& operator= (IntStack&&);
  // an the other members ...
};
```

```cpp
void swap(IntStack& s1, IntStack& s2) {
  IntStack aux{s1};
  s1 = s2;
  s2 = aux;
}
```

```cpp
int main () {
  IntStack a{10};
  IntStack b{20};
  swap(a, b);
}
```

what is the execution of this program ?

example of a lot of copy for a simple swap !

```
class IntStack {
public:
  IntStack ( int );

  IntStack (const IntStack&);
  IntStack (IntStack&&);

  IntStack& operator= (const IntStack&);
  IntStack& operator= (IntStack&&);
  // an the other members ...
};
```

```
void swap(IntStack& s1, IntStack& s2) {
  IntStack aux{s1};
  s1 = s2;
  s2 = aux;
}
```

```
int main () {
  IntStack a{10};
  IntStack b{20};
  swap(a, b);
}
```

what is the execution of this program ?

- construction of the object **a**
- construction of the object **b**
- copy-construction of a in **aux**
- assignment of **b** in **a**
- assignment of **aux** in **b**
- destruction of **aux**
- destruction of **b**
- destruction of **a**

replace the copies by moves

```
class IntStack {
public:
  IntStack (int);

  IntStack (const IntStack&);
  IntStack (IntStack&&);

  IntStack& operator= (const IntStack&);
  IntStack& operator= (IntStack&&);
  // an the other members ...
};
```

```
void swap(IntStack& s1, IntStack& s2) {
  IntStack aux {std::move(s1)};
  s1 = std::move(s2);
  s2 = std::move(aux);
}
```

```
int main () {
  IntStack a{10};
  IntStack b{20};
  swap(a, b);
}
```

- construction of the object **`a`**
- construction of the object **`b`**
- move-construction of a in **`aux`**
- move-assignment of **`b`** in **`a`**
- move-assignment of **`aux`** in **`b`**
- destruction of **`aux`**
- destruction of **`b`**
- destruction of **`a`**

`std::move(x)` tells the compiler that the object **`x`** is to be considered like a **right-value** even if it might not be : **move-construction** and **move-assignment** will be called

Technical aspects of object's life cycle

```
struct X {
  X ();
  ~X ();

  X (const X&);
  X& operator= (const X&);
};
```

```
X function (X var) {
  return var;
}
```

```
int main () {
  X f1;                    // construction of f1
  f1 = function(f1);       // copy−construction of f1 in var
                           // copy−construction of var in a temporary
                           //     (returned by function)
                           // copy−assignment of the temporary in f1
                           // destruction of var
                           // destruction of the temporary
}
                           // destruction of f1
```

Derived Classes (1)

classes are user-defined types

remember that a class :

- **implements** a **user-defined** type
- with a set of **data members**
- and with a set of **member functions**
- the members you want the outside to **see** and **use** must be **`public`** (your public interface)
- the members required for your implementation details must be **`private`**

extending classes

but sometimes, the **set** of members becomes **insufficient**

you need to **add** more **data** members and/or more **member functions** to some code

but how will you extend a code if you don't want to touch the **code** because it is already **developped** and **debugged**

or how will you do if not all the objects of your user-defined type have exactly the same features

- there is a set of general properties common to all the objects
- and sub-sets of objects have different specific properties

Extend an Existing Code

the first way to extend an existing code in C++ is by using composition :

- embed an **object** of the existing type in your new **class** as a **data member**
- of course, the new class accesses only the object's public part of the embed object

the second way to do this is with inheritance :

- create a **new class** by **extending** an **existing** class
- the extension will access the `public` and `protected` part of the **extended** class)

Embed Objects in your Class with Composition

i want to create a new user-defined type named **`Vehicle`** composed of : one engine, four wheels, two doors (with window), one trunk

```
class Engine {};
```

```
class Wheel {};
```

```
class Window {};
class Door {
  Window window;
};
```

```
class Trunk {};
```

```
class Vehicle {
private:
  Engine engine;
  Wheel wheels[4];
  Door doors [2];
  Trunk trunk;
};
```

```
int main () {
  Vehicle v;
}
```

Composition

make embedded **objects private** to **prevent** the outside from **changing** them without your **permission**

the embedded **object** becomes **part** of the **underlying implementation** of your **class**

if you only want to **use** the **features** of **existing classes** inside a **new class :** use **composition**

before an object of the new class is constructed, all the embedded objects are previously constructed

```
class Vehicle {
private:
  Engine engine;
  Wheel wheels[4];
  Door doors [2];
  Trunk trunk;
};
```

```
int main () {
  Vehicle v;
}
```

in the order of their declaration, are created :

- one `Engine` object
- an array of four `Wheel` objects
- an array of two `Door`
- one `Trunk`
- and one `Vehicle`

Composition or Inheritance ? Composition ? ?

now i want to create two new user-defined types **Car** and **Truck**, what do you think of these types **Car** and **Truck** ?

```cpp
class Vehicle {
private:
  Engine engine;
  Wheel wheels[4];
  Door doors [2];
  Trunk trunk;
};
```

```cpp
class Car {
  Vehicle vehicle;
};
```

```cpp
class Truck {
  Vehicle vehicle;
};
```

Car and **Truck** are both composed of **Vehicle**, but **Vehicle** and **Car** and **Truck** have nothing in common ! They cannot be manipulated together, for example put in the same container !

```cpp
#include <list>
int main () {
  Vehicle v1, v2, v3;
  std:: list <Vehicle*> vehicles {&v1, &v2, &v3};
  Car c1, c2, c3, c4;
  std:: list <Car*> cars {&c1, &c2, &c3, &c4};
  Truck t1 , t2;
  std:: list <Truck*> trucks {&t1, &t2};
}
```

Composition or Inheritance ? Inheritance

because **`Car`** and **`Truck`** do **not** contain a **`Vehicle`** but they **are** a **`Vehicle`**, use **derivation** (inheritance)

```
#include <list>
class Vehicle {};
class Car : public Vehicle {
};
class Truck : public Vehicle {
};
```

```
int main() {
  Vehicle v;
  Car c;
  Truck t;
}
```

`Vehicle` is called a direct base class for the classes **`Car`** and **`Truck`**

`Car` and **`Truck`** are called **derived** classes

What are the types of objects of derived classes ?

```cpp
class Vehicle {
};
class Car : public Vehicle {
};
class Truck : public Vehicle {
};
```

```cpp
int main () {
  Vehicle v;
  Car c;
  Truck t;
}
```

- **v** is composed of a [Vehicle] object, the type of **v** is **Vehicle**
- **c** is composed of a [[Vehicle] [Car]] object, the types of **c** are **Vehicle** and **Car**
- **t** is composed of [[Vehicle] [Truck]] object, the types of **t** is **Vehicle** and **Truck**

Notice that we use a **public** derivation !

How has the base class been constructed ?

```
class Vehicle {
};
class Car : public Vehicle {
};
class Truck : public Vehicle {
};
```

```
int main () {
  Vehicle v;
  Car c;
  Truck t;
}
```

- **v** is | Vehicle |
- **c** is | Vehicle | Car |
- **t** is | Vehicle | Truck |

the **Vehicle** *sub-object* of a **Car** an a **Truck** object, must have been constructed !

Do you see how ?

How has the base class been constructed ?

```cpp
class Vehicle {
};
class Car : public Vehicle {
};
class Truck : public Vehicle {
};
```

```cpp
int main () {
  Vehicle v;
  Car c;
  Truck t;
}
```

- **v** is $\boxed{\text{Vehicle}}$
- **c** is $\boxed{\boxed{\text{Vehicle}}\ \boxed{\text{Car}}}$
- **t** is $\boxed{\boxed{\text{Vehicle}}\ \boxed{\text{Truck}}}$

the **Vehicle** *sub-object* of a **Car** an a **Truck** object, must have been constructed !

Do you see how ?

it has been done by a call to the implicitly-defined **Vehicle** default constructor

Passing arguments to base class constructor

supose you need to pass arguments to the base-class **`Vehicle`** constructor

when you enter the constructor of **`Car`** and **`Truck`** the sub-object of type **`Vehicle`** exists

Where can you pass the arguments to the base-class constructor ?

Passing arguments to base class constructor

supose you need to pass arguments to the base-class **`Vehicle`** constructor

when you enter the constructor of **`Car`** and **`Truck`** the sub-object of type **`Vehicle`** exists

Where can you pass the arguments to the base-class constructor ?

```cpp
#include <list>
class Vehicle {
  int number;
public:
  Vehicle (int n) : number{n} {}
};
class Car : public Vehicle {
public:
  Car (int n) : Vehicle {n} {}
};
class Truck : public Vehicle {
public:
  Truck (int n) : Vehicle {n} {}
};
```

```cpp
int main () {
  Vehicle v1 {1324}, v2 {84752};
  Car c1     {6573}, c2 {6574};
  Truck t1   {43},   t2 {325},   t3 {25467};

  std::list<Vehicle*> vehicles
      {&v1, &v2, &c1, &t1, &t2, &t3};
}
```

the initialization list is the only place to pass arguments to base class constructors

accessing data members from derived classes (1/3)

What happens when you try to access a base-class member inside a derived-class member function ?

accessing data members from derived classes (1/3)

What happens when you try to access a base-class member inside a derived-class member function ?

```
class Vehicle {
private:
   int number;
public:
  Vehicle (int n) : number {n} {}
};
class Car : public Vehicle {
public:
  Car (int n) : Vehicle {n} {    // OK Vehicle::Vehicle(int) is public
    number++;                    // ERROR Vehicle::number is private !
  }
};
```

accessing data members from derived classes (2/3)

if you want to access a base-class member inside derived-class member functions but not from everywhere else : you must qualify it **`protected`**

```
class Vehicle {
protected:                              // notice !!
  int number;
public:
  Vehicle (int n) : number {n} {}
};
class Car : public Vehicle {
public:
  Car (int n) : Vehicle {n} {
    number++;                           // OK !
  }
};
```

accessing data members from derived classes (3/3)

when a member of a **class** is :

- **`private`** : it can only be used in member functions and **`friends`** of the class
- **`protected`** : it can be used by member functions and **`friends`** of the class and by member functions and friends of the derived classes
- **`public`** :it can be used everywhere

remember that the **`protected`** qualifier gives access to your internal implementation details !

do not qualify as **`protected`** all members of base-class but only those you need access to in derived-classes !

Manipuling derived class objects

because objects of type **`Car`** and **`Truck`** have the type **`Vehicle`** in common, they can be manipulated *together* and, for example, put in the same container !

```
#include <list>
class Vehicle {};
class Car : public Vehicle {
};
class Truck : public Vehicle {
};
```

```
int main () {
  Vehicle v1, v2;
  Car c1;
  Truck t1, t2, t3;
  std::list<Vehicle*> vehicles {&v1, &v2, &c1, &t1, &t2, &t3};
}
```

Notice that, here, you are using **upcasting** that is : *the conversion of pointers to derived classes* ***`&c1, &t1, &t2, &t3`*** *towards pointers to a common base class (the list contains pointers to* ***`Vehicle`****)*

this is the first step towards *c++ polymorphism*

What is Upcasting ?

upcasting is when you refer to an object of a derived class with a pointer (or a reference) to a base class

```
class Vehicle {};
class Car : public Vehicle {};
class Truck : public Vehicle {};
```

```
int main () {
  Car c;
  Vehicle* pv1 = &c;
  Vehicle& rv1 = c;

  Truck t;
  Vehicle* pv2 = &t;
  Vehicle& rv2 = t;
}
```

when a derived class has a **`public`** base class :

- when manipulated throught pointers or references
- objects of derived class types can be treated as objects of base-class

every thing is correct : an object of type **`Car`** or **`Truck`** is also an object of type **`Vehicle`**

what is happening here ?

```
class Vehicle {};
class Car : public Vehicle {};
class Truck : public Vehicle {};
```

```
int main () {
  Car c;
  Vehicle v1 = c;
  Truck t;
  Vehicle v2 = t;
}
```

what are **v1** and **v2** ?

what is happening here ?

```cpp
class Vehicle {};
class Car : public Vehicle {};
class Truck : public Vehicle {};
```

```cpp
int main () {
  Car c;
  Vehicle v1 = c;
  Truck t;
  Vehicle v2 = t;
}
```

what are **v1** and **v2** ?

they are objects of type **Vehicle**

Watch out ! It is not the same with objects

when a **derived class** has a **`public` base class** :

- when manipulated throught objects
- the base-class part of objects of derived-classes are copied

```
class Vehicle {};
class Car : public Vehicle {};
class Truck : public Vehicle {};
```

```
int main () {
  Car c;
  Vehicle v1 = c;
  Truck t;
  Vehicle v2 = t;
}
```

- **`c`** is composed of a | Vehicle | Car | object
- **`v1`** is composed of a | Vehicle | object : which is the copy [a] of **`Vehicle`** part of the **`c`** object
- **`t`** is composed of a | Vehicle | Truck | object
- **`v2`** is composed of | Vehicle | object : which is the copy of the **`Vehicle`** part of the **`t`** object

a. here, by a call of the implicitly-defined copy constructor of **`Car`**

What is Downcasting ?

it is when you pass *from a pointer to base-class to a pointer to a derived-class*

but because not all base-class objects can be treated as objects of derived-class !

```
class Vehicle {};
class Car : public Vehicle {};
class Truck : public Vehicle {};
```

```
int main () {
  Vehicle v;

  Car* pc = &v;        // ERROR downcasting: invalid conversion from 'Vehicle*' to 'Car*'
  Truck* pt = &v;      // ERROR downcasting: invalid conversion from 'Vehicle*' to 'Truck*'

  Car c;

  Vehicle* pv = &c;  // ok upcasting
  Car* pc = pc1;     //  still  ERROR downcasting: invalid conversion from 'Vehicle*' to 'Car*'
}
```

c++ refuses to implicitly convert a **base-class** object to a **derived-class** object : because a **`Vehicle`** may not be a **`Car`** or a **`Truck`**

order of constructors and destructors with respect to derivation 1/2

```
class A {
public:
  A () {}
  ~A () {}
};
```

```
class B {
public:
  B () {}
  ~B () {}
};
```

```
class C {
public:
  C () {}
  ~C () {}
};
```

```
class D {
public:
  D () {}
  ~D () {}
};
```

```
class E : public A, public B {
public:
  C c;
  D d;
  E () {}
  ~E () {}
};
```

```
int main () {
  E e;
}
```

What is the order of constructors and destructors in this code ?

order of constructors and destructors with respect to derivation 2/2

```
class A {
public:
  A () {}
  ~A () {}
};
```

```
class B {
public:
  B () {}
  ~B () {}
};
```

```
class C {
public:
  C () {}
  ~C () {}
};
```

```
class D {
public:
  D () {}
  ~D () {}
};
```

```
class E : public A, public B {
public:
  C c;
  D d;
  E () {}
  ~E () {}
};
```

```
int main () {
  E e;
  // calls A::A
  //       B::B
  //       C::C
  //       D::D
  //       E::E
}
  // calls E::~E
  //       D::~D
  //       C::~C
  //       B::~B
  //       A::~A
```

the construction starts at the very base class hierarchy and constructs at each level, base-class(es) and data members

Destruction is done in exactly the reverse order

What is happening here ?

```
#include <iostream>
#include < list >
class Vehicle {
public:
  void paint () {
    std::cout << "Painting␣Vehicle␣!\n";
  }
};
class Car : public Vehicle {
public:
  void paint () {
    std::cout << "Painting␣Car␣!\n";
  }
};
```

```
int main () {
  Vehicle v1, v2;
  Car c1, c2;

  std::list <Vehicle> vehicles {v1, v2, c1, c2};
  for (auto e: vehicles)
    e.paint ();
}
```

```
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
```

What is happening here ?

```
#include <iostream>
#include < list >
class Vehicle {
public:
  void paint () {
    std::cout << "Painting␣Vehicle␣!\n";
  }
};
class Car : public Vehicle {
public:
  void paint () {
    std::cout << "Painting␣Car␣!\n";
  }
};
```

```
int main () {
  Vehicle v1, v2;
  Car c1, c2;

  std::list <Vehicle> vehicles {v1, v2, c1, c2};
  for (auto e: vehicles)
    e.paint ();
}
```

```
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
```

only the **`Vehicle`** part of **`v1`**, **`v2`**, **`c1`** and **`c2`** are copied ! so the type of **`e`** is always exactly **`Vehicle`**

What is happening here ?

```
#include <iostream>
#include < list >
class Vehicle {
public:
  void paint () {
    std::cout << "Painting␣Vehicle␣!\n";
  }
};
class Car : public Vehicle {
public:
  void paint () {
    std::cout << "Painting␣Car␣!\n";
  }
};
```

```
int main () {
  Vehicle v1, v2;
  Car c1, c2;

  std:: list <Vehicle*> vehicles {&v1, &v2, &c1, &c2};
  for (auto e: vehicles)
    e.paint ();
}
```

```
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
```

What is happening here ?

```
#include <iostream>
#include < list >
class Vehicle {
public:
  void paint () {
    std::cout << "Painting Vehicle !\n";
  }
};
class Car : public Vehicle {
public:
  void paint () {
    std::cout << "Painting Car !\n";
  }
};
```

```
int main () {
  Vehicle v1, v2;
  Car c1, c2;

  std:: list <Vehicle*> vehicles {&v1, &v2, &c1, &c2};
  for (auto e: vehicles)
    e.paint ();
}
```

```
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
Painting Vehicle !
```

by default the c++ compiler uses a static-binding

in the list **`&c1`** has been upcasted in an object of the type **`Vehicle*`** but in the list the exact type **`c1`** is **`Car`**

but still, in this code, the compiler will bind **`e->paint()`** to **`Vehicle::paint`**

dynamic binding

if you want the compiler to call the function **`Car::paint`** when objects of type **`Car`** have been upcasted in objects of type **`Vehicle*`**

```
void foo (Vehicle* pv) {
  pv->paint();
}
int main () {
  Car c;
  foo(&c);
}
```

```
Painting Car !
```

you must tell the compiler that the binding will only be known at compile-time (dynamic-binding)

when the binding function-call / function-definition can only be known at runtime : specify this functions as been **`virtual`**

dynamic binding

the compiler activates for **`virtual`** [a] member functions the mechanism of (late) binding i.e. binding at execution-time

a. the second step towards polymorphism

dynamic binding is a more expensive mechanism (in time and space) than static binding [a]

a. otherwise c++ member functions will be virtual by default

do not qualify a member function as virtual if you do not need to

What is happening here ?

```
#include <iostream>
#include < list >
class Vehicle {
public:
  virtual void paint () {
    std::cout << "Painting␣Vehicle␣!\n";
  }
};
class Car : public Vehicle {
public:
  void paint () {
    std::cout << "Painting␣Car␣!\n";
  }
};
```

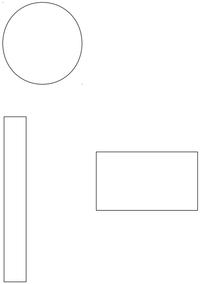
```
int main () {
  Vehicle v1, v2;
  Car c1, c2;

  std:: list <Vehicle*> vehicles {&v1, &v2, &c1, &c2};
  for (auto e: vehicles)
    e->paint();
}
```

```
Painting Vehicle !
Painting Vehicle !
Painting Car !
Painting Car !
```

you are using dynamic-binding on the function **`paint`**

the over-simplified editor of graphic shapes

[TP] implement a class hierarchy for an over-simplified editor of graphic shapes



Requirements :

- a shape can be a circle, a rectangle, a square
- a shape has a position
- a shape can be moved
- a shape can compute its area
- the editor has a list of graphic shapes
- the editor can add a shape in the list
- the editor has a member function **`move`** that searches for the graphic shape at a position (given in the arguments list) and returns it (nullptr otherwise)
- the editor has a member function **`compute_area`** that searches for the graphic shape at a position (given in the arguments list) and returns its area or an error message otherwise
- you must be abble to extend your code to other shapes (an ellipse, ...) without modifying the existing classes

implement a class hierarchy for arithmetic expressions

[TP] implement a class hierarchy for arithmetic expressions

- implement a hierarchy of classes to hold, to evaluate and to print arithmetic expressions
- do not implement parser of arithmetic expressions, you just have to design the set of classes
- your code must be written in such a way the main function below compiles and gives the same results as mine

```
#include <iostream>
#include <iostream>
#include "my_expr.hh"
int main () {
  my_constant c1{12};
  c1.print (std::cout);   // 12
  c1.eval ();
  my_constant c2 {13};
  c2.print (std::cout);  // 13
  c2.eval ();
  my_plus p1 {c1, c2};
  p1.print (std::cout);   // (12+13)
  p1.eval ();
  my_mult m1 {c1, c2};
  m1.print(std::cout);   // (12*13)
  m1.eval();
  my_constant c3 {156};
  c3.print (std::cout);   // 156
  c3.eval ();
  my_unary_minus u1 {c3};
  u1.print (std::cout);   // -156
  u1.eval ();
  my_plus p2 {m1, u1};
  p2.print (std::cout);   // ((12*13)+-156)
  p2.eval ();
  my_divide d1 {c1, p2};
  d1.print (std::cout);   // (12/((12*13)+-156))
  d1.eval ();
    // terminate called after throwing an instance of 'zero_divide'
    // what(): zero divide
    // 1213(12+13)(12...12/((12*13)+-156))Aborted (core dumped)
}
```