C++ course

Classes and Objects

Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr
Translated and lightly adapted from the Cécile Braunstein course

Autumn 2023

SORBONNE UNIVERSITÉ

C++ course
Classes and Objects
Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr
Translated and lightly adapted from the Cécile Braunstein course
Autumn 2023

General Information

Notes

Practicals $\times 6$	50%
Final exam	50%

References

- B. STROUSTRUP, Programming: principles and practice using C++, Pearson Education,2011
- A. KOENIG, B. MOO, Accelerated C++, Addison Wesley 2000
- S. MEYER, Effective C++, Addison Wesley 2005
- D. SLOBODAN, C++ for absolute beginners, Apress 2022

Web resources

- `http://www.cplusplus.com`
- `http://www.cppreference.com`
- `https://duckduckgo.com/`

- @meetingcpp
- @Scott_Meyer
- @c_plus_plus
- @CppCast

SORBONNE UNIVERSITÉ

General Information
Notes
Practicals ×6 50%
Final exam 50%
References
B. STROUSTRUP, Programming: principles and practice using C++, Pearson Education,2011
A. KOENIG, B. MOO, Accelerated C++, Addison Wesley 2000
S. MEYER, Effective C++, Addison Wesley 2005
D. SLOBODAN, C++ for absolute beginners, Apress 2022
Web resources
http://www.cplusplus.com
http://www.cppreference.com
https://duckduckgo.com/
@meetingcpp
@Scott_Meyer
@c_plus_plus
@CppCast

Why C++ ?

C++

- Middle-level language
- Developed by Bjarne Stroustrup in 1979 at Bell Labs
- First view as an extension of C (*C with classes*)

Philosophy

- C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
- C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
- C++ is object language it implies :
  - Re-use
  - Modularity
  - Maintainability

SORBONNE UNIVERSITÉ

Why C++ ?
C++
Middle-level language
Developed by Bjarne Stroustrup in 1979 at Bell Labs
First view as an extension of C (*C with classes*)
Philosophy
C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
C++ is object language it implies :
Re-use
Modularity
Maintainability

Scott Meyers Effective C++ Language still in evolution C++11/C++14/C++17/C++20/C++23

C++ standard

The core and the standard libraries. JTC1/SC22/WG21: ISO group for standardization

The Core language is alive

- 1998: C++98
- 2003: C++0x
- 2011 : C++11
- C++14, C++20
- C++23

C++ standard

The core and the standard libraries. JTC1/SC22/WG21: ISO group for standardization

The Core language is alive

- 1998: C++98
- 2003: C++0x
- 2011 : C++11
- C++14, C++20
- C++23

The Core Guidelines

The Core guidelines

From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*

Some guidelines wise examples

- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments

The Core Guidelines

The Core guidelines
From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*

Some guidelines wise examples
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments

The Core Guidelines

The Core guidelines

From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*

Some guidelines wise examples

- F.2: A function should perform a single logical operation

  Reason A function that performs a single operation is simpler to understand, test, and reuse.
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments

The Core Guidelines
The Core guidelines
From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*
Some guidelines wise examples
- F.2: A function should perform a single logical operation
  Reason A function that performs a single operation is simpler to understand, test, and reuse.
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments

The Core Guidelines

The Core guidelines

From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*

Some guidelines wise examples

- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple

  Reason Large functions are hard to read, more likely to contain complex code, and more likely to have variables in larger than minimal scopes. Functions with complex control structures are more likely to be long and more likely to hide logical errors
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments

The Core Guidelines
The Core guidelines
From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*
Some guidelines wise examples
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
  Reason Large functions are hard to read, more likely to contain complex code, and more likely to have variables in larger than minimal scopes. Functions with complex control structures are more likely to be long and more likely to hide logical errors
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments

The Core Guidelines

The Core guidelines

From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*

Some guidelines wise examples

- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code

  Reason Compilers do not read comments. Comments are less precise than code. Comments are not updated as consistently as code.
- NL.2: State intent in comments

The Core Guidelines
The Core guidelines
From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*
Some guidelines wise examples
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code
Reason Compilers do not read comments. Comments are less precise than code. Comments are not updated as consistently as code.
- NL.2: State intent in comments

The Core Guidelines

The Core guidelines

From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*

Some guidelines wise examples

- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments

  Reason: Code says what is done, not what is supposed to be done. Often intent can be stated more clearly and concisely than the implementation.

The Core Guidelines
The Core guidelines
From Bjarne Stroustrup and Herb Sutter : Best practices for C++14 and later.
Help designers to write *Modern C++*
Some guidelines wise examples
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- NL.1: Don't say in comments what can be clearly stated in code
- NL.2: State intent in comments
Reason Code says what is done, not what is supposed to be done. Often intent can be stated more clearly and concisely than the implementation.

When to use it ?

- Want to be fast and need of abstraction
- System programming
- Low-level programming

But it can be hard

- Complex code
- Handling the memory
- Segmentation fault
- Mix C/C++
- . . .

When to use it ?

- Want to be fast and need of abstraction
- System programming
- Low-level programming

But it can be hard

- Complex code
- Handling the memory
- Segmentation fault
- Mix C/C++
- . . .

Python vs C++ : testing of speed

Python

```python
import time
starting = time.time()
i = 0
while i < 1000000000:
    i += 1

seconds = time.time() - starting
print("D : ", seconds, "s")
```

Python vs C++ : testing of speed

Python

```python
import time
starting = time.time()
i = 0
while i < 1000000000:
    i += 1

seconds = time.time() - starting
print("D : ", seconds, "s")
```

C++

```cpp
#include <chrono>
#include <iostream>
using namespace std::chrono;
int main() {
  auto starting = system_clock::now();
  size_t n = 0;
  while (n < 1000000000)
    ++n;
  auto delay = (system_clock::now() -
        starting);
  auto usecs = duration_cast<
        microseconds>(delay).count();
  std::cout << "D : " << usecs / 1E6
        << "s with n :" << n << std::
        endl;
  return 0;
}
```

SORBONNE UNIVERSITÉ

Python vs C++ : testing of speed

Python vs C++ : testing of speed

Python

(D : 40.092409s)

```python
import time
starting = time.time()
i = 0
while i < 1000000000:
    i += 1

seconds = time.time() - starting
print("D : ", seconds, "s")
```

C++

```cpp
#include <chrono>
#include <iostream>
using namespace std::chrono;
int main() {
  auto starting = system_clock::now();
  size_t n = 0;
  while (n < 1000000000)
    ++n;
  auto delay = (system_clock::now() -
        starting);
  auto usecs = duration_cast<
        microseconds>(delay).count();
  std::cout << "D : " << usecs / 1E6
        << "s with n :" << n << std::
        endl;
  return 0;
}
```

Python vs C++ : testing of speed

Python

(D : 40.092409s)

```python
import time
starting = time.time()
i = 0
while i < 1000000000:
    i += 1

seconds = time.time() - starting
print("D : ", seconds, "s")
```

C++

(D : 0.602283s)

```cpp
#include <chrono>
#include <iostream>
using namespace std::chrono;
int main() {
  auto starting = system_clock::now();
  size_t n = 0;
  while (n < 1000000000)
    ++n;
  auto delay = (system_clock::now() -
        starting);
  auto usecs = duration_cast<
        microseconds>(delay).count();
  std::cout << "D : " << usecs / 1E6
        << "s with n :" << n << std::
        endl;
  return 0;
}
```


Python vs C++ : testing of speed
Python
(D : 40.092409s)
C++
(D : 0.602283s)

Python Comparison

	C++	Python
Output	natif	bytecode pyc
Compatibility	C	-
Rapidity	Very fast	slow
Complexity	+++	+
Memory	explicite (free/delete)	garbage collector
Documentation	-	pydoc
Librairies	STL	extensive
Error	(exceptions)	exceptions
Tools	+	++

SORBONNE UNIVERSITÉ

Python Comparison

	C++	Python
Output	natif	bytecode pyc
Compatibility	C	-
Rapidity	Very fast	slow
Complexity	+++	+
Memory	explicite (free/delete)	garbage collector
Documentation	-	pydoc
Librairies	STL	extensive
Error	(exceptions)	exceptions
Tools	+	++

C vs C++

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  clock_t start, end;
  start = clock();
  int n = 0;
  while (n < 1000000000)
    ++n;
  end = clock();
  printf ("time=%f\n", (double)(end -
        start) / CLOCKS_PER_SEC);
  return 0;
}
```

C++

```cpp
#include <chrono>
#include <iostream>
using namespace std::chrono;
int main() {
  auto starting = system_clock::now();
  size_t n = 0;
  while (n < 1000000000)
    ++n;
  auto delay = (system_clock::now() -
        starting);
  auto usecs = duration_cast<
        microseconds>(delay).count();
  std::cout << "D : " << usecs / 1E6
        << "s with n :" << n << std::
        endl;
  return 0;
}
```


C vs C++
C
C++

Part I

The very first example

Hello world !

helloworld.cpp

```cpp
// The first programm

#include <iostream>

int main()
{
  std::cout << "Hello, world !" << std::endl;
  return 0;
}
```


Hello world !
helloworld.cpp
// The first programm
#include <iostream>
int main()
{
std::cout << "Hello, world !" << std::endl;
return 0;
}

Call the compiler

```
g++ –Wall –g helloworld.cpp –o hello
```

Compile Options

g++ accepts most options as gcc

- Wall : all warnings
- g : include debug code
- o : specify the output file name (`a.out` by default)

SORBONNE UNIVERSITÉ

Call the compiler

g++ –Wall –g helloworld.cpp –o hello

Compile Options
g++ accepts most options as gcc
- Wall : all warnings
- g : include debug code
- o : specify the output file name (a.out by default)

By default main return 0 if OK,
equivalent to first leaving the function normally (which destroys the objects with automatic storage duration) and then calling std::exit with the same argument as the argument of the return. (std::exit then destroys static objects and terminates the program)

Comments

Line comment

```
// The first program
```

Block comment

```
/*
The first program could be written
in a block comment
*/
```

Rationale

Comments
Line comment
// The first program
Block comment
/*
The first program could be written
in a block comment
*/
Rationale

Comments : Some examples 1/3

Not sure they are useful

```
// variable "v" must be initialized

// variable "v" must be used only by function "f()"

// call function "init()" before calling any other function in this file

// call function "cleanup()" at the end of your program
// don't use function "weird()"

// function "f(int ...)" takes two or three arguments
```

or

```
a = b+c; // a becomes b+c
count++; // increment the counter
```


Comments : Some examples 1/3
Not sure they are useful
// variable "v" must be initialized
// variable "v" must be used only by function "f()"
// call function "init()" before calling any other function in this file
// call function "cleanup()" at the end of your program
// don't use function "weird()"
// function "f(int ...)" takes two or three arguments
or
a = b+c; // a becomes b+c
count++; // increment the counter

Comments : examples 2/3

More useful

```
//      tbl.c:  Implementation of the symbol table.

/*
        Gaussian elimination with  partial  pivoting.
        See Ralston: "A  first  course  ..."  pg 411.
*/

//      scan(p,n,c)  requires  that  p points  to  an array  of  at  least  n elements

// sort(p,q)  sorts  the  elements of the  sequence [p:q) using < for  comparison.

// Revised to handle invalid  dates.  Bjarne Stroustrup,  Feb 29 2013
```


Comments : examples 2/3
More useful
// tbl.c: Implementation of the symbol table.
/*
Gaussian elimination with partial pivoting.
See Ralston: "A first course ..." pg 411.
*/
// scan(p,n,c) requires that p points to an array of at least n elements
// sort(p,q) sorts the elements of the sequence [p:q) using < for comparison.
// Revised to handle invalid dates. Bjarne Stroustrup, Feb 29 2013

Comments : examples 3/3

Be careful : Not nested comments

```
/*
remove expensive check
 if  (check(p,q))  error ("bad p q")  /*  should never happen */
*/
```


Comments : examples 3/3
Be careful : Not nested comments
/*
remove expensive check
if (check(p,q)) error ("bad p q") /* should never happen */
*/

Comments : Rationale of good comments of Bjarne Stoutrup

A good comment states what a piece of code is supposed to do (**the intent of the code**), whereas the code (only) states what it does (in terms of how it does it). Preferably, a comment is expressed at a **suitably high level of abstraction** so that it is easy for a human to understand without delving into minute details.

Preferences, comments for :

- source file : common declarations , references to manuals, authors , general hints for maintenance, etc...
- class, template, and namespace
- nontrivial function : stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
- global and namespace variable and constant
- A few comments where the code is nonobvious and/or nonportable
- Very little else

Comments : Rationale of good comments of Bjarne Stoutrup

A good comment states what a piece of code is supposed to do (**the intent of the code**), whereas the code (only) states what it does (in terms of how it does it). Preferably, a comment is expressed at a **suitably high level of abstraction** so that it is easy for a human to understand without delving into minute details.

Preferences, comments for :
• source file : common declarations , references to manuals, authors , general hints for maintenance, etc...
• class, template, and namespace
• nontrivial function : stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
• global and namespace variable and constant
• A few comments where the code is nonobvious and/or nonportable
• Very little else

Comments : Rationale of good comments of Bjarne Stoutrup

A good comment states what a piece of code is supposed to do (**the intent of the code**), whereas the code (only) states what it does (in terms of how it does it). Preferably, a comment is expressed at a **suitably high level of abstraction** so that it is easy for a human to understand without delving into minute details.

Preferences, comments for :

- source file : common declarations , references to manuals, authors , general hints for maintenance, etc...
- class, template, and namespace
- nontrivial function : stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
- global and namespace variable and constant
- A few comments where the code is nonobvious and/or nonportable
- Very little else

SORBONNE UNIVERSITÉ

Comments : Rationale of good comments of Bjarne Stoutrup

A good comment states what a piece of code is supposed to do (the intent of the code), whereas the code (only) states what it does (in terms of how it does it). Preferably, a comment is expressed at a suitably high level of abstraction so that it is easy for a human to understand without delving into minute details.

Preferences, comments for :
• source file : common declarations , references to manuals, authors , general hints for maintenance, etc...
• class, template, and namespace
• nontrivial function : stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
• global and namespace variable and constant
• A few comments where the code is nonobvious and/or nonportable
• Very little else

Comments : Rationale of good comments of Bjarne Stoutrup

A good comment states what a piece of code is supposed to do (**the intent of the code**), whereas the code (only) states what it does (in terms of how it does it). Preferably, a comment is expressed at a **suitably high level of abstraction** so that it is easy for a human to understand without delving into minute details.

Preferences, comments for :

- source file : common declarations , references to manuals, authors , general hints for maintenance, etc...
- class, template, and namespace
- nontrivial function : stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
- global and namespace variable and constant
- A few comments where the code is nonobvious and/or nonportable
- Very little else

Comments : Rationale of good comments of Bjarne Stoutrup

A good comment states what a piece of code is supposed to do (**the intent of the code**), whereas the code (only) states what it does (in terms of how it does it). Preferably, a comment is expressed at a **suitably high level of abstraction** so that it is easy for a human to understand without delving into minute details.

Preferences, comments for :
• source file : common declarations , references to manuals, authors , general hints for maintenance, etc...
• class, template, and namespace
• nontrivial function : stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
• global and namespace variable and constant
• A few comments where the code is nonobvious and/or nonportable
• Very little else

Program details

#include

Many fundamentals facilities are part of standard library rather than core language

```
#include <iostream>
```

#include directive + angle brackets refers to standard header

main function

- Every C++ program must contain a function named `main`. When we run the program, the implementation call this function.
- The result of this function is an integer to tell the implementation if the program ran successfully Convention :

$$0 : \textit{success} \quad | \quad \neq 0 : \textit{fail}$$

SORBONNE UNIVERSITÉ

Program details
#include
Many fundamentals facilities are part of standard library rather than core language
#include <iostream>
#include directive + angle brackets refers to standard header
main function
Every C++ program must contain a function named main. When we run the program, the implementation call this function.
The result of this function is an integer to tell the implementation if the program ran successfully Convention :
0 : success | ≠ 0 : fail

Directive en C
cstdio ...

Using the standard library for output

```
std::cout << "Hello, world !" << std::endl;
```

- << : output stream operator
- std:: : namespace `std`
- std::cout : standard output stream
- std::endl : stream manipulator(end of line)

SORBONNE UNIVERSITÉ

Using the standard library for output

std::cout << "Hello, world !" << std::endl;

- << : output stream operator
- std:: : namespace std
- std::cout : standard output stream
- std::endl : stream manipulator(end of line)

std::endl : finit la ligne courante de la sortie, si le programme produit d'autre sortie, elles seront sur une nouvelle ligne
name space
Comment c'est en C?

Namespace

Purpose

- Collection of identifier (variable name, type name . . .)
- Avoid name conflict :

```
int cout = 2;
std::cout << cout << std::endl;
```

Declaration

```
namespace myNamespace
{
  int a, b;
}
```

Usage:

```
myNamespace::a
myNamespace::b
```


espace de nommage
Package

Scope example

```
// namespaces
#include <iostream>
using namespace std;
namespace foo
{
  int value() { return 5; }
}
using namespace foo;

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}
using namespace bar ;

int main () {
  cout << foo::value() << endl;
  cout << bar::value() << endl;
  cout << bar::pi << endl;
}
```


Scope example

exemple/namespace.cpp

Using namespace

Tell the compiler which name you are using.

(These lines should be included in the general part of your program)

- Refer to the a specific name of the standard library

```
using std::cout;
```

- Refer to all the names of a namespace

```
using namespace std;
```

SORBONNE UNIVERSITÉ

Using namespace
Tell the compiler which name you are using.
(These lines should be included in the general part of your program)
Refer to the a specific name of the standard library
using std::cout;
Refer to all the names of a namespace
using namespace std;

exemple/namespace_lsb.cpp

```
using std::cout;
using std::endl;
cout << "Hello !"<< endl;
```

Using example

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

namespace first
{
  int x = 5;
  int y = 10;
}
namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}
int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

SORBONNE UNIVERSITÉ

Using example

```
#include &lt;iostream&gt;
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}
namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}
int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

revenir sur l'exemple précédent pour using namespace

Expressions in C++ : Type

- A variable is an object that has a name.
- An object is a part of the memory that has a type.
- Every object, expression and function has a type.
- Types specify properties of data and operations on that data.

Primitive types

Type	bool	char	int	float	double	void	wchar_t
Modifier	signed		unsigned		short	long	long long

To improve the code re-use it is important to use the right type at the right place !

SORBONNE UNIVERSITÉ

Expressions in C++ : Type
- A variable is an object that has a name.
- An object is a part of the memory that has a type.
- Every object, expression and function has a type.
- Types specify properties of data and operations on that data.
Primitive types
Type bool | char | int | float | double | void | wchar_t
Modifier signed | unsigned | short | long | long long
To improve the code re-use it is important to use the right type at the right place !

- A type of an entity determines its behavior.
- Types can be thought of as ways of structuring
- accessing memory
- and defining operations that can be performed on objects of a type.
- char16_t/char32_t c++11

Expressions in C++ : Variable definition

Local variable

- Variable can be define anywhere in the program.
- local variable are destroyed when an end of block is reached.

```
{
  std::string  name; // var creation
  std::cin  >> name // var  life
  std::cout << "Hello " << name << std::endl;
} //  var death
```

- Variable has a type and an interface

How to define a variable

type-name name;	(definition)
type-name name = value;	(definition + initialization)
type-name name(args);	(definition + initialization)

Expressions in C++ : Variable definition
Local variable
Variable can be define anywhere in the program.
local variable are destroyed when an end of block is reached.
{ std::string name; // var creation std::cin >> name // var life std::cout << "Hello " << name << std::endl; } // var death
Variable has a type and an interface
How to define a variable
type-name name; (definition)
type-name name = value; (definition + initialization)
type-name name(args); (definition + initialization)

- Interface : collection of operation available on an object of that type
- Standard library says that every string object starts out with the initial value.
- Initialization : if object default else undefined

-> Highlight C differences

Expressions in C++ : Declaration vs. Definition

Declaration

Tells the compiler about the name and the type of something

```cpp
extern int x;                  // object declaration
size_t numDigit(int number); // function declaration;
class Widget;                  // Class declaration
```

Definition

Provides the compiler with details :

- set size of memory,
- code body,
- initialization,
- ...

Expressions in C++ : Declaration vs. Definition
Declaration
Tells the compiler about the name and the type of something
extern int x; // object declaration
size_t numDigit(int number); // function declaration;
class Widget; // Class declaration
Definition
Provides the compiler with details :
set size of memory,
code body,
initialization,
...

Expressions in C++ : Variable default-initialization

```
#include <iostream>
#include <string>
using namespace std;
int main(){
  int a = 2;
  int b(4);
  int c;
  cout << a << "  " << b << "  "
       << c << endl;

  string name;
  string surname("Max");
  cout << name << "  " <<
    surname << endl;
  return 0;
}
```

Rules

- Class type
  - Always initialized
  - Implicit initialization → call default constructor
  - string : implicitly empty ("\0")
- Primitive type
  - No implicit initialization
  - Variable may be undefined

SORBONNE UNIVERSITÉ

Expressions in C++ : Variable default-initialization

- Initialisation : si object defaut sinon undefined -> consists of whatever we have in the memory
- Standard library says that every string object starts out with the initial value.
- undefined whatever there is in memory at this place at this time

-> Highlight C differences

Expressions in C++ : Constant

```
const unsigned int size_max = 15 ;
```

Purpose

Keyword **const** :

- Part of a variable's definition
- The variable must be initialized as part of its definition

Use :

- Promise that the value of the variable is unchanged during its lifetime
- Make program easier to understand : A name give more information than a value
- May be used as global parameters

ES.9: *Avoid ALL_CAPS names*

Expressions in C++ : Constant
const unsigned int size_max = 15 ;
Purpose
Keyword const :
Part of a variable's definition
The variable must be initialized as part of its definition
Use :
Promise that the value of the variable is unchanged during its lifetime
Make program easier to understand : A name give more information than a value
May be used as global parameters
ES.9: Avoid ALL_CAPS names

prefer to #define for debug purpose and we can make it depend on a class

Help compiler to detect usage errors

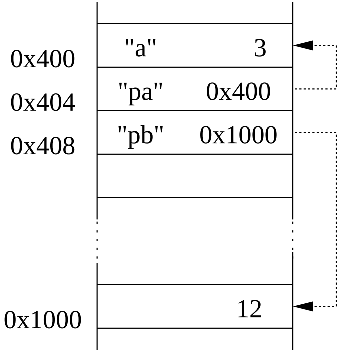
Pointers

Definition

A pointer is a value that represents the address of an object.

Every distinct object has a unique address. It's the the part of the computer's memory that contains the object.

```cpp
int main()
{
    int a = 3;
    int *pa ;
    int *pb ;

    pa = &a;
    pb = new int;
    *pb = 12;
    return 0;
}
```



SORBONNE UNIVERSITÉ

Pointers
Definition
A pointer is a value that represents the address of an object.
Every distinct object has a unique address. It's the the part of the computer's memory that contains the object.

Pointers usage

Operators on pointer

`&x`	: address operator
`*px`	: dereference operator
`T* p`	: declaration of a pointer to `T` (`*p` has a type `T`)
`nullptr`	: constant value, differs from every pointer to any object

Common issues

- segmentation fault
- double free corruption
- memory leaks

→ from c++11 smart pointers std :: unique_ptr and std :: shared_ptr

ES.42: *Keep use of pointers simple and straightforward*

Pointers usage

Operators on pointer
&x : address operator
*px : dereference operator
T* p : declaration of a pointer to T (*p has a type T)
nullptr : constant value, differs from every pointer to any object

Common issues
- segmentation fault
- double free corruption
- memory leaks

→ from c++11 smart pointers std :: unique_ptr and std :: shared_ptr
ES.42: Keep use of pointers simple and straightforward

RAII : Resource Acquisition Is Initialization

P.8:: *Don't leak any ressources*

One way is to make use of std :: unique_ptr and std :: shared_ptr (but not only)

std :: unique_ptr

- A std :: unique_ptr does not share its pointer: it can't be copied.
- The pointed object is destroyed when the unique_pointer goes out of scope.

std :: shared_pointer

- Several std :: shared_ptr objects may own the same object.
- The object is destroyed and its memory deallocated when the last remaining shared_ptr owning the object is destroyed.

2023-08-09

MSR MU5EEB11 : C++ course
└Pointers
 └Pointer's definition
  └RAII : Resource Acquisition Is Initialization

RAII : Resource Acquisition Is Initialization

P.8:: *Don't leak any ressources*

One way is to make use of std :: unique_ptr and std :: shared_ptr (but not only)

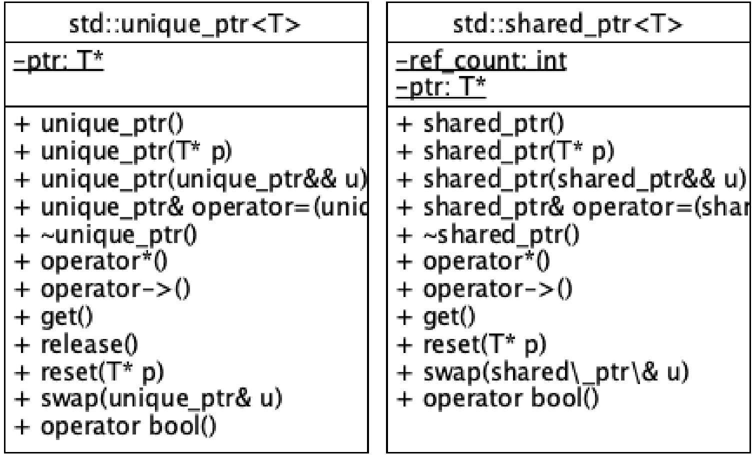
std :: unique_ptr

- A std :: unique_ptr does not share its pointer: it can't be copied.
- The pointed object is destroyed when the unique_pointer goes out of scope.

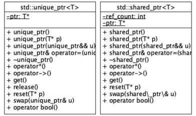
std :: shared_pointer

- Several std :: shared_ptr objects may own the same object.
- The object is destroyed and its memory deallocated when the last remaining shared_ptr owning the object is destroyed.

Smart pointers



Smart pointers



Operations on pointers

Exercise

What is the output of this program. We assume that
`&x = 0xbf84e7b8`

```
#include <iostream>
using namespace std;
int main() {
  int x{5}; // equivalent of "int x = 5;" or "int x(5);"
  int *p = &x;
  cout << "x = " << x << endl;
  cout << "p = " << p << " ; *p = " << *p << endl;

  *p = 6;
  p = p + 1;
  cout << "x = " << x << endl;
  cout << "p = " << p << " ; *p = " << *p << endl;

  return 0;
}
```

SORBONNE UNIVERSITÉ

Operations on pointers
Exercise
What is the output of this program. We assume that
&x = 0xbf84e7b8
#include <iostream>
using namespace std;
int main() {
int x{5}; // equivalent of "int x = 5;" or "int x(5);"
int *p = &x;
cout << "x = " << x << endl;
cout << "p = " << p << " ; *p = " << *p << endl;
*p = 6;
p = p + 1;
cout << "x = " << x << endl;
cout << "p = " << p << " ; *p = " << *p << endl;
return 0;
}

Constant and pointers

```cpp
char greeting[] = "Hello";
char * p = greeting              // non-const pointer,
                                 // non-const data

const char * p = greeting        // non-const pointer,
                                 // const data

char * const p = greeting        // const pointer,
                                 // non-const data

const char * const p = greeting  // const pointer,
                                 // const data
```


Constant and pointers

```cpp
char greeting[] = "Hello";
char * p = greeting              // non-const pointer,
                                 // non-const data

const char * p = greeting        // non-const pointer,
                                 // const data

char * const p = greeting        // const pointer,
                                 // non-const data

const char * const p = greeting  // const pointer,
                                 // const data
```

Arrays

Definition

- Part of the core language
- Sequence of one ore more objects of the same size
- The number of elements must be known at compile time

An array is not a class type

Good use

```cpp
double coords[3];
```

```cpp
const size_t NDim = 3;
double coords[NDim];
```

- The constant is known at compile time.
- Better for documentation purpose.

SORBONNE UNIVERSITÉ

Arrays
Definition
- Part of the core language
- Sequence of one ore more objects of the same size
- The number of elements must be known at compile time
An array is not a class type
Good use
double coords[3]; | const size_t NDim = 3; double coords[NDim];
- The constant is known at compile time.
- Better for documentation purpose.

- Arrays doesn't grow dynamically as in the standard library
- no members no size_type to name an appropriate type to deal with the size of Ann array.
- size_t in deffest
- for documentation purpose

Array initialization

```
const int DIM = 3;
double tab[DIM] = {1,2,3};

double number[] {1,2,3,4,5,6};

const int month_length[] = {
31, 28, 31, 30, 31, 30,
31, 31, 30, 31, 30, 31
};
```

= is not mandatory since c++11 for initialization of simple and array types

Number of elements

- The size may be implicit :

```
size_t n= sizeof(number)/sizeof(*number);
```

But always known at compile time

Array initialization
const int DIM = 3;
double tab[DIM] = {1,2,3};
double number[] {1,2,3,4,5,6};
const int month_length[] = {
31, 28, 31, 30, 31, 30,
31, 31, 30, 31, 30, 31
};
= is not mandatory since c++11 for initialization of simple and array types
Number of elements
The size may be implicit :
size_t n= sizeof(number)/sizeof(*number);
But always known at compile time

La taille du tableau doit etre connue a la compilation cst/def

Attention à la declaration et le = optionnel

Memory management

Three kinds

1. Automatic management: system's job
2. Static allocation: once and only once
3. Dynamic allocation: with respect to our needs

Memory management
Three kinds
1. Automatic management: system's job
2. Static allocation: once and only once
3. Dynamic allocation: with respect to our needs

Automatic memory management

Local variables

- The program allocates memory when it encounters the definition of the variable
- The program deallocates that memory at the end of the block containing the definition.

➛ Any pointers to this variable become invalid

```
int* invalid_pointer ()
{
    int x;
    return &x; // never !
}
```


Automatic memory management
Local variables
The program allocates memory when it encounters the definition of the variable
The program deallocates that memory at the end of the block containing the definition.
Any pointers to this variable become invalid
int* invalid_pointer ()
{
int x;
return &x; // never !
}

qd un pointeur est desaloue il devient invalide -> au programmeur de verifier ces trucs la seg fault

Static allocation

```cpp
int y;
int * pointer_to_static ()
{
    static int x;
    return &x;
}
```

Global/static variables

- x,y are allocated once and only once before the function call.
- x,y are deallocated only at the end of the run.
- x,y are initialized only once: the first time the program run encounters the definition.
- The function always return the same address.

SORBONNE UNIVERSITÉ

Static allocation

```cpp
int y;
int * pointer_to_static ()
{
    static int x;
    return &x;
}
```

Global/static variables

- x,y are allocated once and only once before the function call.
- x,y are deallocated only at the end of the run.
- x,y are initialized only once: the first time the program run encounters the definition.
- The function always return the same address.

The pointer is valid as long as the program run
The default-initialization for a static local or global variable is 0. Not the case for local var

Life time example

var_life.cc

```
int a = 1;
void f()
{
  int b = 1;
  static int c = a;
  cout << " a = " << a++
       << " b = " << b++
       << " c = " << c << endl;
  c = c + 2;
}

int main()
{
  while( a < 4) f();
  return 0;
}
```


Life time example
var_life.cc

Dynamic allocation

Allocate **new**

```
int* p = new int(42);
```

- Allocate new objects of type **int**
- Initialize the object to 42
- Cause `p` to point to that object

The object stays around until it is deleted or the program ends.

Deallocate **delete**

```
delete p;
```

- Frees space memory used by `*p`
- Invalids `p`
- **delete** only object created by **new**

Deleting a zero pointer has no effect.

Dynamic allocation
Allocate new
int* p = new int(42);
Allocate new objects of type int
Initialize the object to 42
Cause p to point to that object
The object stays around until it is deleted or the program ends.
Deallocate delete
delete p;
Frees space memory used by *p
Invalids p
delete only object created by new
Deleting a zero pointer has no effect.

delete can be forget but no good for memory leak

Dynamic allocation example

var_life_dyn.cc

```
class mine
{
  int  m;
public:
  mine(int x) :m(x){cout << "m(" << m << ") created" << endl;};
  ~mine(){cout << "m(" << m << ") destroyed" << endl;};
};
void f ()
{
  mine  m(42);
  mine * p = new mine(24);
  cout<< "END OF F" <<endl;

}
int  main()
{
   f () ;
   cout<< "AFTER RETURN OF F" <<endl;
   cout<< "END OF MAIN" <<endl;
  return 0;
}
```


Dynamic allocation example
var_life_dyn.cc

exemple/var_life_dyn.cc

Allocating and deallocating an array

```
T* p = new T[n]
delete[] p
```

Allocation

- Allocates and default-initializes an array of n places
- Returns a pointer to the first element in the array

Deallocation

- Destroys the objects in the array
- Frees the memory used to hold the array
- Invalids the pointer p;

Allocating and deallocating an array
T* p = new T[n]
delete[] p
Allocation
Allocates and default-initializes an array of n places
Returns a pointer to the first element in the array
Deallocation
Destroys the objects in the array
Frees the memory used to hold the array
Invalids the pointer p;

le n peut etre inconnu a la compil mais la taille du type doit etre connu.
=> pour plusieurs dimensions le nb de colonne doit être constante

Multidimensional arrays

Allocate

```
int n = 4;
int (*M)[3]=new int[n][3];
// n lines of 3 columns
```

```
int n2 = 4;
type** M = new type*[n] ;
for(int i=0 ; i< n ; ++ i)
{
    M[i] = new type[n2] ;
}
// n lines of  n2 columns
```

Deallocate

```
delete[] M;
```

```
for(int i=0 ; i< n ; ++ i)
{
    delete[] M[i] ;
}
delete[] M;
```


Multidimensional arrays
Allocate
int n = 4;
int (*M)[3]=new int[n][3];
// n lines of 3 columns
int n2 = 4;
type** M = new type*[n] ;
for(int i=0 ; i< n ; ++ i)
{
M[i] = new type[n2] ;
}
// n lines of n2 columns
Deallocate
delete[] M;
for(int i=0 ; i< n ; ++ i)
{
delete[] M[i] ;
}
delete[] M;

References

```cpp
int i;
int &r = i;

int j = 2;
r = j;
```

New in C++

- A reference is a pointer self-dereferenced
- It acts as a synonym for the refered variable
- It's an address but after initialization all operation affect the pointed variable

Useful ? Yes !

- Give a specific name to no-name element (table element)
- Use in a parameter list of function.

SORBONNE UNIVERSITÉ

References

int i;
int &r = i;

int j = 2;
r = j;

New in C++
- A reference is a pointer self-dereferenced
- It acts as a synonym for the refered variable
- It's an address but after initialization all operation affect the pointed variable

Useful ? Yes !
- Give a specific name to no-name element (table element)
- Use in a parameter list of function.

attention : The operator & :
in a type declaration = a reference
in an expression = an address il n'existe pas de ref de ref c'est la meme chose que definir un autre synonyme.
Tout ce qui est fait sur r est fait sur i

Si on defini un nonconst ref on ne peut pas le faire pointer sur un const ca demande des permissions que const ne permet pas.

References vs. Pointers

```
#include<iostream>
using namespace std;
void increment(int& v)
{
    v++;
}
int main(){
 int a = 3 ;  int* pa;
 int & ra = a;
 pa = &a ;  ra = 4;
 increment(a);

 cout << "a  = " << a <<" &a = " << &a<< endl;
 cout << "*pa = " << *pa << " pa = " << pa << endl;
 cout << "ra = " << ra <<" &ra = "<< &ra << endl;
 return 0;
}
```


exemple/ref_point_exemple.cpp

References - other examples

Explain the following lines :

```cpp
double d;
const double d_const = 4.0;
double &a = d;
const double &b = d;
double &c = d_const;
const double &c_const = d_const;
```


References - other examples
Explain the following lines :
double d;
const double d_const = 4.0;
double &a = d;
const double &b = d;
double &c = d_const;
const double &c_const = d_const;

References - other examples

Explain the following lines :

```cpp
double d;
const double d_const = 4.0;
double &a = d;
const double &b = d;
double &c = d_const;
const double &c_const = d_const;
```

Example

```cpp
double d; // declare a double
const double d_const = 4.0; // declare a const double
double &a = d; // a is a synomym for d
const double &b = d; // b is a read–only synonym for d
double &c = d_const; // This is not possible
const double &c_const = d_const; // c_const is a synonym for d_const
```

SORBONNE UNIVERSITÉ

References - other examples

Explain the following lines :

```cpp
double d;
const double d_const = 4.0;
double &a = d;
const double &b = d;
double &c = d_const;
const double &c_const = d_const;
```

Example

```cpp
double d; // declare a double
const double d_const = 4.0; // declare a const double
double &a = d; // a is a synomym for d
const double &b = d; // b is a read–only synonym for d
double &c = d_const; // This is not possible
const double &c_const = d_const; // c_const is a synonym for d_const
```

Parameters

Example

Computing student's grade

```
double grade(double midterm, double final, double homework)
{
  return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

Parameters list

Behaves like local variables to the function :

- Calling the function : create the variables
- Returning from the function : destroy the variables

Parameters

Example
Computing student's grade
double grade(double midterm, double final, double homework)
{
return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}

Parameters list
Behaves like local variables to the function :
Calling the function : create the variables
Returning from the function : destroy the variables

Call by value

```cpp
std::cout << "Your final grade is : " << setprecision(3)
          << grade(midterm,final,sum/count)
          << setprecision(prec) << std::endl;
```

Arguments

- Arguments can be a variable or an expression.
- Each argument is used to initialize the corresponding parameters
- The parameters take a copy of the value of the argument

Call by value

```cpp
std::cout << "Your final grade is : " << setprecision(3)
          << grade(midterm,final,sum/count)
          << setprecision(prec) << std::endl;
```

Arguments

- Arguments can be a variable or an expression.
- Each argument is used to initialize the corresponding parameters
- The parameters take a copy of the value of the argument

Call by reference

We want to have a function that returns two values at once.

```
int function_f(int a,int& b)
{
   r = a + b
   b = b + 1;

   return r;
}
```

Reference

- Fast : only the address is in `b`.
- No copy
- The function will modify `b`
- The compiler manages the operators "*" and "&"

SORBONNE UNIVERSITÉ

Call by reference
We want to have a function that returns two values at once.
Reference
Fast : only the address is in b.
No copy
The function will modify b
The compiler manages the operators "*" and "&"

si pas de lvalue on pourrais ranger un truc dans qqchose qui serait detruit à la fin de la fonction. Ca revient a ranger un truc dans qqchose auquel on a pas acces

exemple/function_call.cc

Call by `const` reference

```cpp
int function_f(int a,const int& b)
{
    r = a + b

    return r;
}
```

`const`

- Direct access to the argument
- No copy of the argument
- Promise we will not change the value

Con.1: *By default, make objects immutable*

Con.3: *By default, pass pointers and references to consts*

Call by const reference
int function_f(int a,const int& b)
{
r = a + b
return r;
}
const
Direct access to the argument
No copy of the argument
Promise we will not change the value
Con.1: By default, make objects immutable
Con.3: By default, pass pointers and references to consts

Resume

Call by	const ref	value	ref
	`void f(const string &a)`	`void f(string a)`	`void f(string &a)`
modification of `a`	No	local	with side effect
accepted values	All	All	non-temporary
advantages	security no copy	simple	more general no copy

Resume

Call by	const ref	value	ref
	`void f(const string &a)`	`void f(string a)`	`void f(string &a)`
modification of `a`	No	local	with side effect
accepted values	All	All	non-temporary
advantages	security no copy	simple	more general no copy

Part II

Object Oriented Programming

Part II

Object Oriented Programming

Organizing programs and data

Thinking big

To keep larger programs manageable, we need break it into independents named parts.
Fundamental ways of organizing program :

- Functions
- Data structure
- Class : combine Functions and data structure

And then ...

- Divide program into files
- Compile separately
- Write Makefile

SORBONNE UNIVERSITÉ

Organizing programs and data

Thinking big
To keep larger programs manageable, we need break it into independents named parts.
Fundamental ways of organizing program :
- Functions
- Data structure
- Class : combine Functions and data structure

And then ...
- Divide program into files
- Compile separately
- Write Makefile

Programmation oriented object (POO)

Advantages

- Re-use
- Modularity
- Maintainability

Oriented object language

Before :

- Data more or less well organized
- Functions and computation applied on these data
- A program is a following of affectation and computation

POO :

- Modules (*classes*) representing data and functions
- A program is a set of *objects* interacting by calling their own functions(*methods*)

Programmation oriented object (POO)
Advantages
Re-use
Modularity
Maintainability
Oriented object language
Before :
Data more or less well organized
Functions and computation applied on these data
A program is a following of affectation and computation
POO :
Modules (classes) representing data and functions
A program is a set of objects interacting by calling their own functions(methods)

Concepts

Objects

An object is a recognizable element characterized by its structure (*attributes*) and its behavior (*methods*)

➛ Object = Class instance

Class

Groups and creates objects with the same properties (method and attributes).

Class members :

- Attributes : define the domain of value
- Methods : define behavior ; set of function modifying the state of an object

A class has got at least two methods (create and delete) - *may be implicit*

Concepts
Objects
An object is a recognizable element characterized by its structure (attributes) and its behavior (methods)
➛ Object = Class instance
Class
Groups and creates objects with the same properties (method and attributes).
Class members :
Attributes : define the domain of value
Methods : define behavior ; set of function modifying the state of an object
A class has got at least two methods (create and delete) - may be implicit

une classe = abstraction, un objet = entitee concrete

Information hiding

C.9: Minimize exposure of members

Purpose

Restrict access to a class by its interface

- Put constraints for the use and the interaction between objects.
- Programmer see only a part of the object corresponding to its behavior
- Help updates and changes for a class.

Class has two parts

- An interface : access for external users,
- Internal data and internal implementation.

Information hiding
C.9: Minimize exposure of members
Purpose
Restrict access to a class by its interface
Put constraints for the use and the interaction between objects.
Programmer see only a part of the object corresponding to its behavior
Help updates and changes for a class.
Class has two parts
An interface : access for external users,
Internal data and internal implementation.

Encapsulation
L'interface ne doit jamais être modifiée. Elle doit être suffisante pour manipuler les objets de la classe, mais ne doit pas contenir tous ses membres car alors la classe ne pourrait pas être modifiée.

Encapsulation. Information hiding. Minimize the chance of unintended access. This simplifies maintenance.

Defining new types in C++

```cpp
class Rectangle{
    double _h;
    double _w;
public:
    std::istream& read(std::istream&);
    double area() const;
};
```

An object Rectangle is made of memory composed by :

- 2 double numbers
- 2 functions
- default constructor and destructor.

Usually written in a header file.

Create interface

Our Goal :

- Hiding implementation details
- Users can access only through functions

Defining new types in C++

class Rectangle{ double _h; double _w; public: std::istream& read(std::istream&); double area() const; };

An object Rectangle is made of memory composed by :
- 2 double numbers
- 2 functions
- default constructor and destructor.

Usually written in a header file.

Create interface
Our Goal :
- Hiding implementation details
- Users can access only through functions

New style

```cpp
#include <iostream>

class Rectangle{
  double h;
  double w;
public:
  std::istream& read(std::istream &in){  in  >> h >> w; return in;}
  double area() const {return h * w;}
};

int  main(){
  Rectangle my_rect;
  my_rect.read(std::cin);
  std::cout << "Area:  " << my_rect.area() << std::endl;
  return 0;
}
```


New style

```cpp
#include <iostream>

class Rectangle{
  double h;
  double w;
public:
  std::istream& read(std::istream &in){  in  >> h >> w; return in;}
  double area() const {return h * w;}
};

int  main(){
  Rectangle my_rect;
  my_rect.read(std::cin);
  std::cout << "Area:  " << my_rect.area() << std::endl;
  return 0;
}
```

Protection - Data Encapsulation

```
class Rectangle{
public:
   // interface
     void set_rectangle(double,double);
     bool is_higher(const Rectangle& r) const {return h > r.h;}
     double area() const;
     std :: istream& read(std::istream&);

private:
   //implementation
     double _h;
     double _w;
};
Rectangle p,q;
```

SORBONNE UNIVERSITÉ

Protection - Data Encapsulation

class Rectangle{
public:
// interface
void set_rectangle(double,double);
bool is_higher(const Rectangle& r) const {return h > r.h;}
double area() const;
std :: istream& read(std::istream&);
private:
//implementation
double _h;
double _w;
};
Rectangle p,q;

Protection label

Each protection label defines the accessibility of all members that follow the label.

labels

They can appear in any order

- `private` : Inaccessible members from outside
- `public` : accessible members from outside

`struct` or `class` ?

There is no difference except :

- default protection : private for a class ; public for struct.
- by convention : struct for simple data structure

Protection label

Each protection label defines the accessibility of all members that follow the label.

labels

They can appear in any order

- `private` : Inaccessible members from outside
- `public` : accessible members from outside

`struct` or `class` ?

There is no difference except :

- default protection : private for a class ; public for struct.
- by convention : struct for simple data structure

Member functions - Definition

`read`

```
istream& Rectangle::read(istream& in)
{
    in >> _h >> _w;
    return in;
}
```

Usually implemented in the source files

Particularities

- The name of the function `Rectangle::read`
- No object `Rectangle` in parameters list
- Direct access to data elements of our object

Member functions - Definition
read
istream& Rectangle::read(istream& in)
{
in >> _h >> _w;
return in;
}
Usually implemented in the source files
Particularities
The name of the function Rectangle::read
No object Rectangle in parameters list
Direct access to data elements of our object

- on dit qu’on est en train de definir la fonction de la classe correspondante
- fonction membre de l’objet
- this is not useful

Member functions

`area`

```
double Rectangle::area() const
{
  return _h*_w;
}
```

What's new ?

- `area` is a member of `Rectangle` : implicit reference to the object
- and `const` ?

Member functions
area
double Rectangle::area() const
{
return _h*_w;
}
What's new ?
area is a member of Rectangle : implicit reference to the object
and const ?

Const member function

```
double Rectangle::area() const {...}  //new
double area(const Rectangle&) {...} //old
```

Const

- In the old version we ensure that the grade function do not change the parameter
- In the new version, the function is qualified as `const`
- `area` can be applied to a `const` or no`const` object
- `read` cannot be call by a `const` object

Con.2: *By default, make member functions const*

Const member function

double Rectangle::area() const {...} //new
double area(const Rectangle&) {...} //old

Const
In the old version we ensure that the grade function do not change the parameter
In the new version, the function is qualified as const
area can be applied to a const or noconst object
read cannot be call by a const object
Con.2: By default, make member functions const

create const when we call
nom de fonction avec const

Member functions

`is_higher`

```
bool is_higher(const Rectangle& r) const {return _h > r._h;}
```

Inline function

- To avoid function call overhead, we can *inline* funciton
- Ask the compiler to replace the call by the code if it's possible.

SORBONNE UNIVERSITÉ

Member functions
is_higher
bool is_higher(const Rectangle& r) const {return _h > r._h;}
Inline function
- To avoid function call overhead, we can *inline* funciton
- Ask the compiler to replace the call by the code if it's possible.

Life cycle of an object

Run of the constructor for derived object

1. Allocating memory space for the entire object (base-class + class members)
2. Calling the base-class constructor to initialize the base-class part of the object
3. Initializing the members following the declaration order.
4. Executing the body of the constructor, if any

Constructors of the base-class are always called.

Life cycle of an object
Run of the constructor for derived object
1. Allocating memory space for the entire object (base-class + class members)
2. Calling the base-class constructor to initialize the base-class part of the object
3. Initializing the members following the declaration order.
4. Executing the body of the constructor, if any
Constructors of the base-class are always called.

Constructor

Definition

- Special member functions that defines how object are initialized.
- If no constructor defined the compiler will synthesized one for us.
- They have the same name as the name of the class itself.
- They have no return type and no return instruction.

```cpp
class Rectangle{
    Rectangle();  //construct an empty object
    Rectangle(std::istream&);  //construct by reading a stream as before
    Rectangle(double h, double w); //construct with given  initial  value
};
```


Constructor
Definition
- Special member functions that defines how object are initialized.
- If no constructor defined the compiler will synthesized one for us.
- They have the same name as the name of the class itself.
- They have no return type and no return instruction.
class Rectangle{
Rectangle(); //construct an empty object
Rectangle(std::istream&); //construct by reading a stream as before
Rectangle(double h, double w); //construct with given initial value
};

constructor don't need to allocate memory for the object, it's done before.

Call the constructor

When an object is created, a call to the constructor is always performed.
How to call it ?

```
//Basic form
Rectangle a(2,3);
Rectangle b = Rectangle(5,7);

// Constructor with only one parameter
Rectangle c = cin; //   Rectangle c = Rectangle(cin)

// Dynamic allocation initialisation is not mandatory
Rectangle* d = new Rectangle(1,5);

// For anonymous object
cout << Rectangle(3,4).is_higher(Rectangle(2,7)) << endl;

//Default constructor
Rectangle e; // Rectangle e = Rectangle()
Rectangle f[10] // 10 calls of Rectangle()
```

SORBONNE UNIVERSITÉ

Call the constructor
When an object is created, a call to the constructor is always performed.
How to call it ?

More example add copy constructor
constructeur.cpp rectangle_constr.cpp

The default constructor

Implementation

The one without argument.

```
Rectangle::Rectangle():_h(0),_w(0) {}
```

Constructor initializer

When we create a new class object :

1. The implementation allocate memory to hold the object
2. It initializes the object using initial values as specified in an initializer list
3. It executes the constructor body

The default constructor
Implementation
The one without argument.
Rectangle::Rectangle():_h(0),_w(0) {}
Constructor initializer
When we create a new class object :
1. The implementation allocate memory to hold the object
2. It initializes the object using initial values as specified in an initializer list
3. It executes the constructor body

- All data are initialized
- define in the body make the work twice

Calling the constructor initializer

A not so good version :

```
Segment::Segment(int x1, int y1,
                 int x2, int y2,
                 int w){
    start = Point(x1,y1);
    end = Point(x2,y2);
    width = w;
}
```

A better one:

```
Segment::Segment(int x1, int y1,
                 int x2, int y2,
                 int w):start (x1,y1),
        end(x2,y2), width(w){}
```

When should I use constructor initializer ?

- Members object don't have default constructor.
- Constant members.
- Reference members.

Calling the constructor initializer

A not so good version :

Segment::Segment(int x1, int y1, int x2, int y2, int w){ start = Point(x1,y1); end = Point(x2,y2); width = w; }

A better one:

Segment::Segment(int x1, int y1, int x2, int y2, int w):start (x1,y1), end(x2,y2), width(w){}

When should I use constructor initializer ?

- Members object don't have default constructor.
- Constant members.
- Reference members.

Members already created before the call of the body.

Can be local variable

Copy Constructor

```
Rectangle a(1,2);
Rectangle b = a.scale(2);
Rectangle c = b;
Rectangle d(b);
cout << b.is_higher(c)  << endl;
```

Explicit or implicit copies are controlled by the copy constructor.

Copy constructor

- Exists to initialize a new object of the same type
- Define what a copy means (including function member)
- Does not change the initial object

```
Rectangle(const Rectangle& r);
```

SORBONNE UNIVERSITÉ

Copy Constructor

Rectangle a(1,2);
Rectangle b = a.scale(2);
Rectangle c = b;
Rectangle d(b);
cout << b.is_higher(c) << endl;

Explicit or implicit copies are controlled by the copy constructor.

Copy constructor
- Exists to initialize a new object of the same type
- Define what a copy means (including function member)
- Does not change the initial object

Rectangle(const Rectangle& r);

Call by copy to

- parameter object
- reference
- const

The compiler may construct one for you Copy elision for return value optimization (check wikipedia)

How a copy constructor works ?

The compilator may synthesises one for us

- Each members are just copied out
- If there is object member their copy constructor is called
- Otherwise it is a simple "bit to bit" memory copy.

Our own copy constructor

Completely useless for `rectangle` !
So let’s take another example.

How a copy constructor works ?
The compilator may synthesises one for us
- Each members are just copied out
- If there is object member their copy constructor is called
- Otherwise it is a simple "bit to bit" memory copy.
Our own copy constructor
Completely useless for `rectangle` !
So let’s take another example.

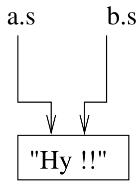
How a copy constructor works ?

string1.h

```cpp
#include <string.h>
#include <cstring>
class OurString {
     char * s;
public:
     OurString(char * s_new);
     ~OurString(){delete[]  s;}
};

OurString::OurString(char * s_new)
{
          s = new char[strlen(s_new)+1];
          strcpy(s,s_new);
}
```

string1.cpp

```cpp
#include "string1.h"
int  test  (OurString s)
{
  return 2;
}
int  main()
{
  OurString a("Hy  !!");
  OurString b = a;
  test(a);
}
```



SORBONNE UNIVERSITÉ

What we call a copy constructor differs from affectation

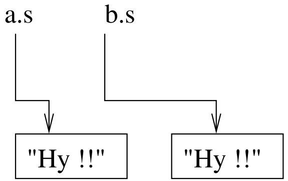
How a copy constructor works ?

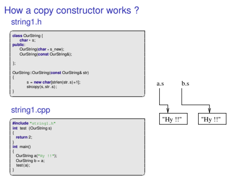
string1.h

```
class OurString {
      char * s;
public:
      OurString(char * s_new);
      OurString(const OurString&);

};

OurString::OurString(const OurString& str)
{
            s = new char[strlen(str.s)+1];
            strcopy(s, str.s);
}
```

string1.cpp

```
#include "string1.h"
int test (OurString s)
{
   return 2;
}
int main()
{
   OurString a("Hy !!");
   OurString b = a;
   test(a);
}
```





What we call a copy constructor differs from affectation

Destructor

```
class Rectangle{
 ~Rectangle();
};
```

Definition

- Free the allocated memory
- Only one in a class
- Can be synthesized if it doesn't exist

SORBONNE UNIVERSITÉ

Destructor

```
class Rectangle{
 ~Rectangle();
};
```

Definition

- Free the allocated memory
- Only one in a class
- Can be synthesized if it doesn't exist

Synthesized Constructor

- If you don't write any constructor ; C++ might automatically synthesize a default constructor for you
  - the default constructor is one that takes no arguments and that initializes all member variables to 0-equivalents (0, NULL, false, ..)
  - C++ does this iff your class has no const or reference data members
- If you don't define your own copy constructor, C++ will synthesize one for you
  - it will do a shallow copy of all of the fields (i.e., member variables) of your class
  - sometimes the right thing, sometimes the wrong thing

Synthesized Constructor

- If you don't write any constructor ; C++ might automatically synthesize a default constructor for you
  - the default constructor is one that takes no arguments and that initializes all member variables to 0-equivalents (0, NULL, false, ..)
  - C++ does this iff your class has no const or reference data members
- If you don't define your own copy constructor, C++ will synthesize one for you
  - it will do a shallow copy of all of the fields (i.e., member variables) of your class
  - sometimes the right thing, sometimes the wrong thing

Test yourself

Write a class DoubleArray and its copy constructor that deals with the preceding example!

Test yourself

Write a class DoubleArray and its copy constructor that deals with the preceding example!

Overloaded functions

Same name but different

- Two functions/methods may have the same name
- But their signature have to be different
- The compiler resolves the choice
- If the compiler fails an error diagnostic is produced

SORBONNE UNIVERSITÉ

Overloaded functions
Same name but different
- Two functions/methods may have the same name
- But their signature have to be different
- The compiler resolves the choice
- If the compiler fails an error diagnostic is produced

cas de changer un type genre homework

- Learn to read compiler insults
- Highlight differences with C.

Overloaded functions - example

```
#include <iostream>
#include <string>
double grade(double mid, double final, double hw){
  return 0.2 * mid + 0.4 * final + 0.4 * hw;
}
double grade(double mid, double final, double hw1, double hw2){
  return 0.2 * mid + 0.4 * final + 0.4 * ((hw1+hw2)/2);
}
int main(){
    double x;
    x = grade(10,15,14);
    std::cout << x << std::endl;
    x = grade(10,15,14,20);
    std::cout << x << std::endl;
    return 0;
}
```


Overloaded functions - example

Overloaded operators

The effect of an operator depends on the type of its operands.

Example

```
#include<iostream>
#include<string>
int main()
{    // Example 1
     int a = 2;
     int b = 3;
     std::cout << a + b << std::endl;

     // Example 2
     std::string s = "Hello,";
     std::cout << s + "World !" << std::endl;

     return 0;
}
```

SORBONNE UNIVERSITÉ

Overloaded operators
The effect of an operator depends on the type of its operands.
Example

Attention modify only when it make sense with meaning too

Our own overloaded operators

As a member

```
class Point{
  int x;
  int y;
public:
  Point(int a,int b){x=a;y=b;};

  Point operator+(const Point& a){
    return Point(x + a.x , y + a.y);
  };
```

OR

As a non-member

```
Point operator+(const Point& b,
                const Point& a){
  return Point(b.getx() + a.getx() ,
               b.gety() + a.gety());
}
```

```
int main()
{
    Point p1(3,4);
    Point p2(7,6);
    p1 = p1 + p2;

    cout << "(" << p1.x <<"," ;
    cout << p1.y << ")"<< endl;
    return 0;
}
```

Result ?

Our own overloaded operators

As a member

class Point{ int x; int y; public: Point(int a,int b){x=a;y=b;}; Point operator+(const Point& a){ return Point(x + a.x , y + a.y); };

OR

As a non-member

Point operator+(const Point& b, const Point& a){ return Point(b.getx() + a.getx() , b.gety() + a.gety()); }

int main() { Point p1(3,4); Point p2(7,6); p1 = p1 + p2; cout << "(" << p1.x <<"," ; cout << p1.y << ")"<< endl; return 0; }

Result ?

Certains pas le choix =, -> [] ()

si left ope can’t be modified by our class

Write its own operators

Generalities

An operator is used in expressions.
An expression returns a result and may have some side effects.
➢ It can be defined as a function.

Structure

```
return_type operator@ (argument_list){
  Opertor body
}
```

Restrictions

- The operators **::** (scope resolution), . (member access), **.*** (member access through pointer to member), and **?:** (ternary conditional) cannot be overloaded.
- New operators cannot be created

For the side effect

- Operator »
- Modifier Point

Type de retour Attention au const

Copy assignment Operator **operator=**

Should be a member of the class

Assignment behavior

```
int x, y , z;
x = y = z = 15;
```

- The assignment is right-associative and returns a reference to its left-hand argument.
- All members should be copied

```
Point& operator=(const Point& p){
  copy(p);
  return *this;
}
```


Copy assignment Operator operator=
Should be a member of the class
Assignment behavior
int x, y , z;
x = y = z = 15;
The assignment is right-associative and returns a reference to its left-hand argument.
All members should be copied
Point& operator=(const Point& p){
copy(p);
return *this;
}

Warning the compiler may construct one for you

Synthesized assignment operator

- If you don't overload the assignment operator, C++ will synthesize one for you
  - it will do a shallow copy of all of the fields (i.e., member variables) of your class
  - sometimes the right thing, sometimes the wrong thing

Synthesized assignment operator

- If you don't overload the assignment operator, C++ will synthesize one for you
  - it will do a shallow copy of all of the fields (i.e., member variables) of your class
  - sometimes the right thing, sometimes the wrong thing

Default class behaviours

C.20: *If you can avoid defining default operations, do*

The rules of three (five since c++11)

if a class defines any of the following then it should probably explicitly define all three:

- destructor
- copy constructor
- copy assignment operator
- the move constructor
- the move assignment operator

SORBONNE UNIVERSITÉ

Default class behaviours
C.20: If you can avoid defining default operations, do
The rules of three (five since c++11)
if a class defines any of the following then it should probably explicitly define all three:
- destructor
- copy constructor
- copy assignment operator
- the move constructor
- the move assignment operator

Static members

```
class Rectangle{
public:
 static int _nb_rectangle;
 Rectangle(){_nb_rectangle++;};
 Rectangle(double a,double b):_h(a),_w(b){_nb_rectangle++;};
private:
 double _h;
 double _w;
};
```

How it works ?

- Share by all objects of the same type.
- There exists one and only one memory part for a given class.
- Initializing : in the global part of a program (for public and private members)
- Calling : p.nbRectangle or Rectangle::nbRectangle

SORBONNE UNIVERSITÉ

Static members

class Rectangle{
public:
 static int _nb_rectangle;
 Rectangle(){_nb_rectangle++;};
 Rectangle(double a,double b):_h(a),_w(b){_nb_rectangle++;};
private:
 double _h;
 double _w;
};

How it works ?
Share by all objects of the same type.
There exists one and only one memory part for a given class.
Initializing : in the global part of a program (for public and private members)
Calling : p.nbRectangle or Rectangle::nbRectangle

static function no this just about static members

Static member functions

Differ from ordinary functions

- Do not operate on a object of the class type.
- Associate to the class, not to a particular object.
- Access only static members.

```
class Rectangle{
 double h;
 double w;
 static Rectangle * _first_rect ;
public:
 static int nbRectangle;
 Rectangle(){nbRectangle++;};
 Rectangle(double a, double b):h(a),w(b){nbRectangle++;};
 static void show_first(){ cout << "(" <<
     _first_rect ->h<<"," << _first_rect->w<<")"<<endl;}
};
```


- under the Namespace scope

Static use

```
#include <iostream>
using namespace std;
int Rectangle::nb_rectangle = 0;
Rectangle * Rectangle:: first_rect = new Rectangle(2,2);

int main(){
    Rectangle p(3,4) ;
  cout <<   Rectangle::nb_rectangle << endl;
    Rectangle::show_first() ;
    p.show_first() ;
    return 0;
}
```


Static use

#include <iostream>
using namespace std;
int Rectangle::nb_rectangle = 0;
Rectangle * Rectangle:: first_rect = new Rectangle(2,2);

int main(){
Rectangle p(3,4) ;
cout << Rectangle::nb_rectangle << endl;
Rectangle::show_first() ;
p.show_first() ;
return 0;
}

rectangle.h

```cpp
#include <iostream>
class Rectangle{
public:
    Rectangle();
    Rectangle(std::istream&);
    Rectangle(const Rectangle&r);
    Rectangle(double, double);
    Rectangle scale(double);
    void set_rectangle(double,double);
    bool is_higher(const Rectangle& r) const {return _h > r._h;};
    std :: istream& read(std::istream&);
    double area() const;
  friend std :: ostream& operator<< (std::ostream& out,const Rectangle& r);
private:
    double _h;
    double _w;
};
```

SORBONNE UNIVERSITÉ

Static use

```cpp
#include <iostream>
using namespace std;
int Rectangle::nb_rectangle = 0;
Rectangle * Rectangle:: first_rect = new Rectangle(2,2);

int main(){
    Rectangle p(3,4) ;
  cout <<  Rectangle::nb_rectangle << endl;
    Rectangle::show_first() ;
    p.show_first() ;
    return 0;
}
```

rectangle.cpp

```cpp
#include "rectangle.h"

using namespace std;

Rectangle::Rectangle():_h(0),_w(0){}

Rectangle::Rectangle(double x, double y):_h(x),_w(y){}

Rectangle::Rectangle(std::istream& in)
{
    read(in);
}

std::istream& Rectangle::read(std::istream& in){
    in >> _h >> _w;
    return in;
}
double Rectangle::area() const
{
    return _h*_w;
    Rectangle r = cin;
    //r.read(cin);
    //Rectangle l = r.scale(2);
    cout << r.area()<< endl;
    //cout << l << endl;
    return 0;
```


Static use

```cpp
#include <iostream>
using namespace std;
int Rectangle::nb_rectangle = 0;
Rectangle * Rectangle:: first_rect = new Rectangle(2,2);

int main(){
    Rectangle p(3,4) ;
  cout <<  Rectangle::nb_rectangle << endl;
    Rectangle::show_first() ;
    p.show_first() ;
    return 0;
}
```

```
}
```



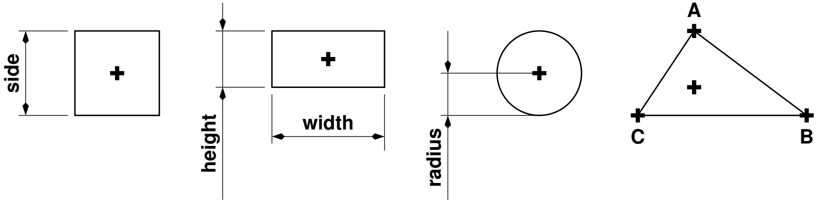
```
#include <iostream>
using namespace std;
int Rectangle::nb_rectangle = 0;
Rectangle * Rectangle:: first_rect = new Rectangle(2,2);

int main(){
    Rectangle p(3,4) ;
  cout <<   Rectangle::nb_rectangle << endl;
    Rectangle::show_first() ;
    p.show_first() ;
    return 0;
}
```

Part III

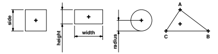
Inheritance

Part III
Inheritance

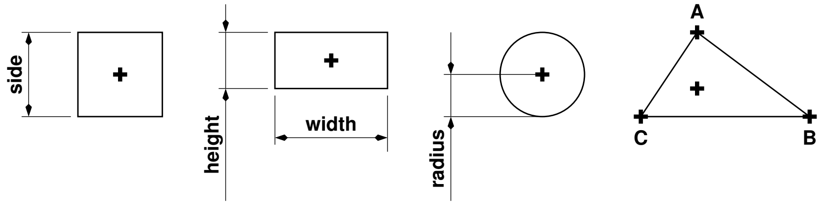
Basic cases



Basic cases



Basic cases



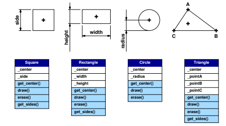
Square
_center
_side
get_center()
draw()
erase()
get_sides()

Rectangle
_center
_width
_height
get_center()
draw()
erase()
get_sides()

Circle
_center
_radius
get_center()
draw()
erase()

Triangle
_center
_pointA
_pointB
_pointC
get_center()
draw()
erase()
get_sides()

Basic cases



Look more closer

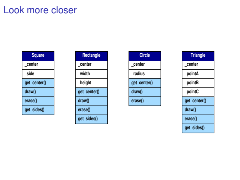
Square
_center
_side
get_center()
draw()
erase()
get_sides()

Rectangle
_center
_width
_height
get_center()
draw()
erase()
get_sides()

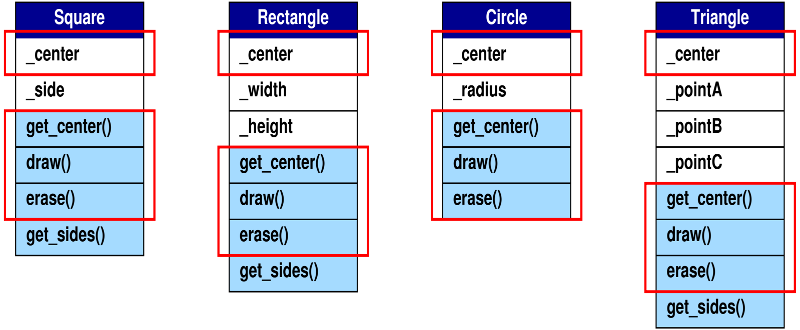
Circle
_center
_radius
get_center()
draw()
erase()

Triangle
_center
_pointA
_pointB
_pointC
get_center()
draw()
erase()
get_sides()

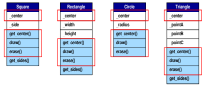
SORBONNE UNIVERSITÉ



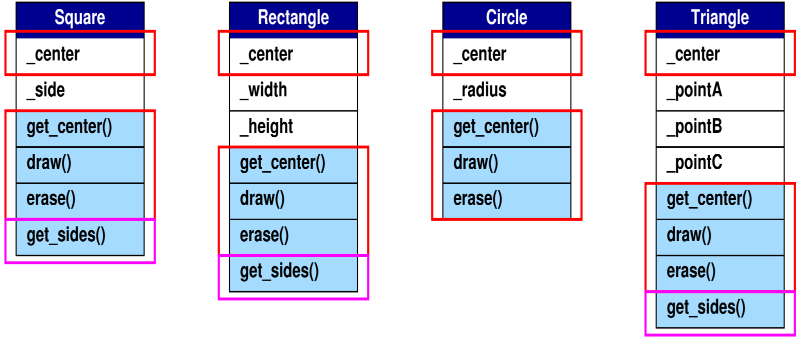
Look more closer



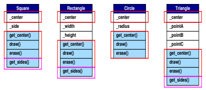
Look more closer



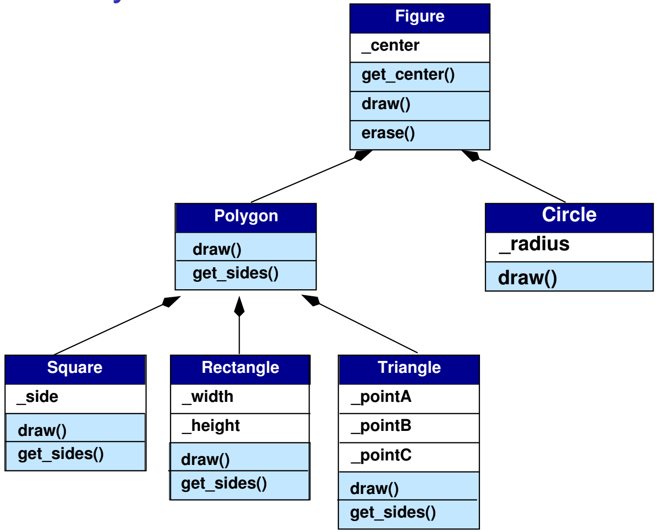
Look more closer

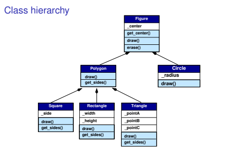


Look more closer



Class hierarchy





Every members of figure are members of the derived class except the constructor, the destructor, assignment operator

Can redefine function

Defining class hierarchy

```cpp
class Figure {
 private :
 Point _center;

 public :
 Figure(Point& center);

 Point& get_center();
 void draw() const;
 void erase();

};
```

```cpp
#include "figure.h"

class Circle: public Figure{
    private:
        double _radius;

    public:
        Circle () ;
        void draw() const;
};
```

Inheritance limit

The constructor, the destructor, assignment operator of Figure are not members of the derived class.

define function

- invariant
- with default behavior
- abstract

Public inheritance

Let $\mathcal{B}$ and $\mathcal{C}$ be two classes such that $\mathcal{C}$ derived from $\mathcal{B}$ *publicaly*.

`private` and `public`

- `private` members of $\mathcal{B}$: Only class $\mathcal{B}$ may access to these members
- `public` members of $\mathcal{B}$: Everyone may access to these members

What the compiler will say about that ?

```
void Circle::Draw(){
    std::cout << "center : ";
    std::cout << " center :" << _center << " radius : " << _radius << std::
        endl;
}
```

```
figure.h: In member function 'void Circle::Draw()':
figure.h:5: error: 'Point Figure::_center' is private
circle.cc:8: error: inside the context of 'class Circle'
```

Protection revisited

```
class Figure {
 protected :
 Point _center;

 public :
 Figure(Point& center);

 Point& get_center();
 void draw();
 void erase();

};
```

`protected`

- $\mathcal{B}$ and $\mathcal{C}$ have access to these members
- They are still part of the interface
- Users of class $\mathcal{C}$ can not have direct access to these members

Protection revisited

```
class Figure {
 protected :
 Point _center;

 public :
 Figure(Point& center);

 Point& get_center();
 void draw();
 void erase();

};
```

`protected`

- $\mathcal{B}$ and $\mathcal{C}$ have access to these members
- They are still part of the interface
- Users of class $\mathcal{C}$ can not have direct access to these members

Composition of protection

3 types of inheritance

- `public` : Like the definition of a sub-type.
- `private` or `protected` : Hide details of the implementation

Change access to the class members

		Members of the base class		
		public	protected	private
Derived class	public	public	protected	no access
	protected	protected	protected	no access
	private	private	private	no access

Composition of protection
3 types of inheritance
public : Like the definition of a sub-type.
private or protected : Hide details of the implementation
Change access to the class members

		Members of the base class		
		public	protected	private
Derived class	public	public	protected	no access
	protected	protected	protected	no access
	private	private	private	no access

Constructor

Run of the constructor for derived object

1. Allocating memory space for the entire object (base-class + derived-class members)
2. Calling the base-class constructor to initialize the base-class part of the object
3. Initializing the members of the derived class as directed by the constructor initializer
4. Executing the body of the constructor, if any

Constructors of the base-class are always called.

Constructor

Run of the constructor for derived object

1. Allocating memory space for the entire object (base-class + derived-class members)
2. Calling the base-class constructor to initialize the base-class part of the object
3. Initializing the members of the derived class as directed by the constructor initializer
4. Executing the body of the constructor, if any

Constructors of the base-class are always called.

Destructor

Run of the destructor for derived object

1. Executing the body of the destructor, if any
2. Destroying the members of the derived class as directed by the destructor in the opposite order
3. Calling the base-class destructor
4. Deallocating memory space for the entire object (base-class + derived-class members)

Destructors of the base-class are always called.

SORBONNE UNIVERSITÉ

Destructor

Run of the destructor for derived object

1. Executing the body of the destructor, if any
2. Destroying the members of the derived class as directed by the destructor in the opposite order
3. Calling the base-class destructor
4. Deallocating memory space for the entire object (base-class + derived-class members)

Destructors of the base-class are always called.

Constructors

base-class

```
Figure::Figure(){std::cout<<"Default Figure" << std::endl;}

Figure::Figure(Point& center):_center(center){
std::cout<<"Figure with center" << std::endl;
}
```

derived class

```
Circle::Circle():_radius(0){std::cout<<"Default Circle" << std::endl;}

Circle::Circle(Point c,double r):Figure(c),_radius(r){
std::cout<<"Circle init" << std::endl;
}
```

SORBONNE UNIVERSITÉ

Constructors
base-class
Figure::Figure(){std::cout<<"Default Figure" << std::endl;}
Figure::Figure(Point& center):_center(center){
std::cout<<"Figure with center" << std::endl;
}
derived class
Circle::Circle():_radius(0){std::cout<<"Default Circle" << std::endl;}
Circle::Circle(Point c,double r):Figure(c),_radius(r){
std::cout<<"Circle init" << std::endl;
}

Example - use

Constructor

```
Figure f1(p);
Figure f2(p1);

Circle c1(p,3);
Circle g2(p,4);
```

Function call

```
bool compare(const Figure& s1, const Figure& s2){
  return s1.get_center() < s2.get_center();
}

compare(f1,f2);
compare(c1,c2);
compare(c,f);
```

SORBONNE UNIVERSITÉ

Example - use

Constructor

Figure f1(p);
Figure f2(p1);
Circle c1(p,3);
Circle g2(p,4);

Function call

bool compare(const Figure& s1, const Figure& s2){
return s1.get_center() < s2.get_center();
}
compare(f1,f2);
compare(c1,c2);
compare(c,f);

1- example for the call
2- OK because we are referring to the part of stage that is student + figure + example pointer

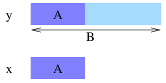
Static cast

Type known at compile time

```
class A {...} ;
class B : public A {...};
```

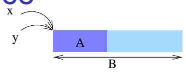
Object

```
B y;
A x = y;
```



Pointer and reference

```
B* y;
A* x = y;
```



Static cast
Type known at compile time
class A {...} ;
class B : public A {...};
Object
B y;
A x = y;
Pointer and reference
B* y;
A* x = y;

Dynamic cast 1

Type known at run time

```
class A {...} ;
class B : public A {...};
```

Only for references pointers

```
B x;
A y = x;
A *ptry = &x;
A &refy = x;
```

- the static type of `*ptry` and `refy` is `A`
- the dynamic type of `*ptry` and `refy` is `B`

Dynamic cast 1

Type known at run time

class A {...} ;
class B : public A {...};

Only for references pointers

B x;
A y = x;
A *ptry = &x;
A &refy = x;

- the static type of *ptry and refy is A
- the dynamic type of *ptry and refy is B

Dynamic cast 2

Syntax

`dynamic_cast<T*>(p)`

- `p` is a pointer
- Transform the type of `p` in `T`
- If it's not possible returns `NULL`

`dynamic_cast<T&>(p)`

- `p` is a reference
- Transform the type of `p` in `T`
- If it's not possible raise an exception

Dynamic cast 2
Syntax
dynamic_cast<T*>(p)
p is a pointer
Transform the type of p in T
If it's not possible returns NULL
dynamic_cast<T&>(p)
p is a reference
Transform the type of p in T
If it's not possible raise an exception

An other example

Comparing grade

Sometimes, we really want to know the real type at run time.

```
void draw_picture(const Figure& s1, const Figure& s2)
{
  s1.draw();
  s2.draw();
}

Figure e1,e2;
Circle  s1,s2;
draw_picture(e1,e2);
draw_picture(s1,s2);
```

How to be sure that the right method `draw()` is used ?

SORBONNE UNIVERSITÉ

An other example

Comparing grade

Sometimes, we really want to know the real type at run time.

void draw_picture(const Figure& s1, const Figure& s2)
{
s1.draw();
s2.draw();
}

Figure e1,e2;
Circle s1,s2;
draw_picture(e1,e2);
draw_picture(s1,s2);

How to be sure that the right method draw() is used ?

Polymorphism

For references and pointers, sometimes we want to know at which class the object really belongs ?

Definition

Polymorphism defines the notion that the behavior of an object does not't have to be known at compile time. The real object type may be known at run time.

What for ?

- Build container with heterogeneous types inside
- For the destructor
- Re-use the code for an other application

Polymorphism

For references and pointers, sometimes we want to know at which class the object really belongs ?

Definition
Polymorphism defines the notion that the behavior of an object does not't have to be known at compile time. The real object type may be known at run time.

What for ?
- Build container with heterogeneous types inside
- For the destructor
- Re-use the code for an other application

Re-use code example with if and type name

`virtual` function

```
class Figure{
public :
  virtual  void draw() const;
  // ...};
```

Virtual function

We can declare function that can be redefine in derived class.
As before, so what ?

- Calling a function that depends on the actual type of an object
- Making this decision at run time

How ?

- Keyword `virtual` used only inside the class definition
- When it's inherited, no need to repeat this keyword

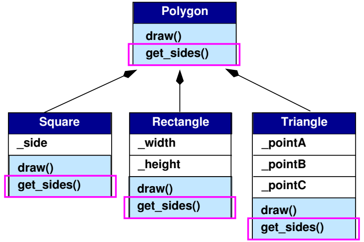
➛ A destructor has to be virtual

SORBONNE UNIVERSITÉ

`virtual` function
class Figure{ public : virtual void draw() const; // ...};
Virtual function
We can declare function that can be redefine in derived class.
As before, so what ?
Calling a function that depends on the actual type of an object
Making this decision at run time
How ?
Keyword virtual used only inside the class definition
When it's inherited, no need to repeat this keyword
➛ A destructor has to be virtual

The compiler do not decide which function is really called
at run time choose the function and look at the actual type of the object
example with draw explain ::

Abstract class



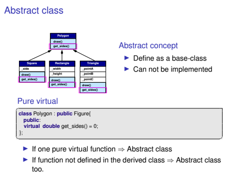
Abstract concept

- Define as a base-class
- Can not be implemented

Pure virtual

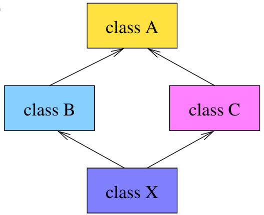
```
class Polygon : public Figure{
  public:
  virtual double get_sides() = 0;
};
```

- If one pure virtual function ⇒ Abstract class
- If function not defined in the derived class ⇒ Abstract class too.



No sense to be implemented, have sense only when implemented
Example with YX, Y, Z and the diifferent possibilities

Multiple inheritance



Derived from many classes

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class X :  public B, public C { /* ... */ };
```

- The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.
- A derived class can inherit an indirect base-class more than once

Leads to ambiguities

SORBONNE UNIVERSITÉ

Multiple inheritance
Derived from many classes
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class X : public B, public C { /* ... */ };
The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.
A derived class can inherit an indirect base-class more than once
Leads to ambiguities

Resolving ambiguities

Members with same names from different classes

- C++ compilers resolves some ambiguities by choosing the minimal path to a member
- Use the scope operator `A::function`

Two same members from different class

- Sometimes it's the correct behavior
- Virtual inheritance

```
class A { /* ... */ };
class B : public virtual A{ /* ... */ };
class C : public virtual A{ /* ... */ };
class X : public B, public C { /* ... */ };
```

SORBONNE UNIVERSITÉ

Resolving ambiguities
Members with same names from different classes
C++ compilers resolves some ambiguities by choosing the minimal path to a member
Use the scope operator A::function
Two same members from different class
Sometimes it's the correct behavior
Virtual inheritance
class A { /* ... */ };
class B : public virtual A{ /* ... */ };
class C : public virtual A{ /* ... */ };
class X : public B, public C { /* ... */ };

Part IV

The Stream Library

Part IV
The Stream Library

I/O stream

Read and write

Stream library

The iostream library is an object-oriented library that provides input and output functionality using streams.

- Input/output is implemented entirely in the library
- No language features supports I/O

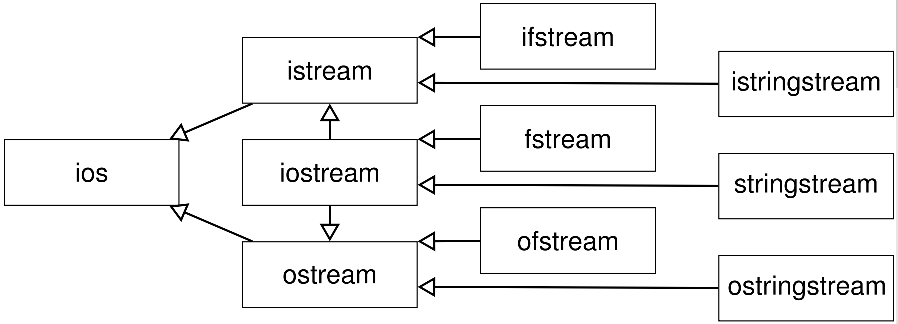
Stream definition

- Represent a device on which input and output operations are performed.
- Can be represented as a source or destination of characters of indefinite length
- Associated generally to a physical source or destination of characters(disk file,keyboard,console)

I/O stream
Read and write
Stream library
The iostream library is an object-oriented library that provides input and output functionality using streams.
Input/output is implemented entirely in the library
No language features supports I/O
Stream definition
Represent a device on which input and output operations are performed.
Can be represented as a source or destination of characters of indefinite length
Associated generally to a physical source or destination of characters(disk file,keyboard,console)

so the characters gotten or written to/from our abstraction called stream are physically input/output
to the physical device.
For example, file streams are C++ objects to manipulate and interact with files;
Once a file stream is used to open a file, any input or output operation performed
on that stream is physically reflected in the file.

Class hierarchy



Using inheritance

- Basic functions are defined only once
- Same operators/functions used for all kind of stream
- Your own classes can be derived easily that look and behave like the standard ones.

Class hierarchy
Using inheritance
- Basic functions are defined only once
- Same operators/functions used for all kind of stream
- Your own classes can be derived easily that look and behave like the standard ones.

The class stream

What's inside ?

- Formatting in formations (format flags, field with, precision ...)
- State information (error state flags)
- Types (flags types, stream size ...)
- Operations (!)
- Members functions (set/get flags, floating-point precision ...)

The class stream

What's inside ?

- Formatting in formations (format flags, field with, precision ...)
- State information (error state flags)
- Types (flags types, stream size ...)
- Operations (!)
- Members functions (set/get flags, floating-point precision ...)

The class stream - example

```
#include<iostream>
using namespace std;
int  main()
{
  double f = 3.14159;
  cout.precision(10);
  cout << f  << endl;
  cout.setf(ios::fixed);    //  floatfield  set to  fixed
  cout << f  << endl;
  cout.flags(  ios::right  |  ios::hex |  ios::showbase );
  cout.width (10);
  cout << 100 << endl;
  cout.unsetf  (  ios_base::showbase | ios::hex);
  cout.width (10);
  cout.  fill  ('>');
  cout << 100 << endl;
  return 0;
}
```


The class stream - example

exemple/iostream.cc

```
  3.14159
3.1415900000
      0x64
>>>>>>> 100
```

The class input and output stream

<iostream>

<ostream> / write

- << : insert data with format operator
- put/write : put character/write block of data
- tellp/seekp : get/set position of the put pointer

➛ cout, cerr, clog are instantiations of this class.

<istream> / read

- >> : extract data with format operator
- get/getline : get data from stream
- tellg/seekg : get/set position of the get pointer

➛ cin is an instantiation of this class.

fstream only adds open and close file member function.

The class input and output stream
<iostream>
<ostream> / write
<< : insert data with format operator
put/write : put character/write block of data
tellp/seekp : get/set position of the put pointer
cout, cerr, clog are instantiations of this class.
<istream> / read
>> : extract data with format operator
get/getline : get data from stream
tellg/seekg : get/set position of the get pointer
cin is an instantiation of this class.
fstream only adds open and close file member function.

- operator so can be overloaded
- cerr write directly/ urgent comment
- clog running comment of what he's doing

Manipulators

Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators.

Some examples

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main () {
  cout << showbase << hex;
  cout << uppercase << 77 << "\t" << nouppercase << 77 << endl;

  double f = 3.14159;
  cout << setprecision(5) << f << "\t" << setprecision(7) << f << endl;
  return 0;
}
```

SORBONNE UNIVERSITÉ

Manipulators

Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators.

Some examples

#include <iostream>
#include <iomanip>
using namespace std;
int main () {
cout << showbase << hex;
cout << uppercase << 77 << "\t" << nouppercase << 77 << endl;
double f = 3.14159;
cout << setprecision(5) << f << "\t" << setprecision(7) << f << endl;
return 0;
}

C++ course

Classes and Objects

Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr
Translated and lightly adapted from the Cécile Braunstein course

Autumn 2023

C++ course
Classes and Objects
Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr
Translated and lightly adapted from the Cécile Braunstein course
Autumn 2023