Object-Oriented Methodology for Software Development with PELICANS

March 15, 2010

INSTITUT DE RADIOPROTECTION ET DE SÛRETÉ NUCLÉAIRE
DIRECTION DE PRÉVENTION DES ACCIDENTS MAJEURS
SERVICE D'ETUDES ET DE MODELISATION DES INCENDIES, DU CONFINEMENT ET DU CORIUM

Centre d'Études de Cadarache - Bâtiment 702 - BP3-13115 SAINT PAUL LEZ DURANCE CEDEX
Téléphone : (33) 4.42.19.96.48 - Fax : (33) 4.42.19.91.66 - E-mail : bruno.piar@irsn.fr

Introduction

PELICANS is a **C++** application framework with a set of integrated reusable components, designed to simplify the task of developing applications of numerical mathematics and scientific computing, particularly those concerning partial differential equations and initial boundary value problems.

PELICANS defines a "prescriptive" approach, without being unnecessarily restrictive, to building applications. It provides both standard component types with a great deal of built-in behavior and standard structures for code.

The standard components can be combined in many ways to build a variety of applications. They can be extended by customizing them or by building new ones with a similar structure. They represent code application developers don't have to write, because it is already done for them.

The standard code structures, or framework, provide pre-built applications with all their generic features already implemented. Developers use object-oriented techniques like inheritance to build on the generic application and tailor components to the new application.

PELICANS allows developers to focus on problems unique to the application at hand. Elements commonly used are handled in the framework. This increases the speed of development, reduces the amount of new code written, enables wide-scale software reuse and reduces application maintenance costs.

Object Oriented Programming

The implementation language of the PELICANS framework is **C++**. This report describes **C++** techniques for instantiating the framework and using the components.

C++ weaknesses concerning Component-Based Development are well known. The main one is probably that it does not allow *seamless development* (a software development process which uses a uniform method and notation throughout all activities, such as problem modeling and analysis, design, implementation and maintenance). In particular, framework plug-points are not represented differently from the rest of the system, which has the undesirable consequence that there is no indication of where the application developer should write the customizing code, or what code should be written. Moreover **C++** does not provide any facility for dealing with clear *component specifications*. Some support of the Design By Contract methodology, although not a guarantee, would be a good way to make pieces of software, developed at different times by different people, work together successfully. The ignorance of the documentation issues is another drawback of **C++**: indeed, if the software and its documentation are treated as separate entities, it is difficult to guarantee that they will remain compatible when things start changing. Keeping everything at the same place, although not a guarantee, would be a good way to help maintain this compatibility.

Besides these weaknesses, **C++** exhibits some pleasant features among which the most noticeable are its support for the basic object oriented techniques (namely inheritance, polymorphism and dynamic binding), its relative efficiency for scientific calculations, the availability of quality free compilers for a wide variety of hardware platforms and a nowadays favorable audience among engineers and researchers of various background.

A specific **C++** usage methodology has been adopted in the development of PELICANS. The present report explains how that methodology influences the way PELICANS-based applications should be written. The spirit of this approach is threefold:

- include a support for Design By Contract, which is considered to be of major importance for Component-Based Development;
- include a support for Self-Documentation (as much as possible, information about a module appears in the module itself rather than externally);
- among the **C++** capabilities, ignore those whose use is considered delicate and tricky in order to maintain the accessibility of PELICANS to non **C++** experienced programmers.

The Quest for Quality

computer programming

the art of making a computer do what you want it to do

software engineering

the production of **quality** software

scientific software architectures designed to survive change

- computer program: network of interacting modules.
- specificity of scientific software development: ever changing requirements.
- symptoms of poor design: hard to change, easy to break, hard to reuse.
- cause of poor design: **improper dependencies** of software modules.

Objective: define and apply principles and guidelines governing **simple** and **adaptable designs**.

Discipline and creativity: what quality software requires is **egoful design** with **egoless expression**

Where Did These Ideas Come From ?

All the foregoing material relies on the expertise of others who have struggled with, and solved the related problems.

- Bertrand Meyer

  Object-Oriented Software Construction, Prentice Hall, 1997.
  http://archive.eiffel.com
- Robert C. Martin

  Agile Software Development, Prentice Hall, 2003
  http://www.objectmentor.com
- Andrew Koenig & Barbara E. Moo

  Accelerated **C++**, Addison-Wesley, 2000.
- Scott Meyers

  Effective **C++**, More Effective **C++**, Addison-Wesley, 1998, 1996.
- Herb Sutter

  Exceptional **C++**, Addison-Wesley, 2000.
- Bjarne Stroustrup

  The **C++** Programming Language, Addison-Wesley, 2000

Who Should Read this Report ?

This report has been written for developers of applications based on PELICANS, that is for those who are using some PELICANS component or for those who are instantiating the PELICANS framework for customization to their specific problem. Since it explains how PELICANS looks like to a user and how its capabilities should be understood, this report should also appeal to the PELICANS development team whose one of the tasks is to guarantee that the externally visible features fulfill the requirements exposed here.

Chapter I

A First Look at **C++**

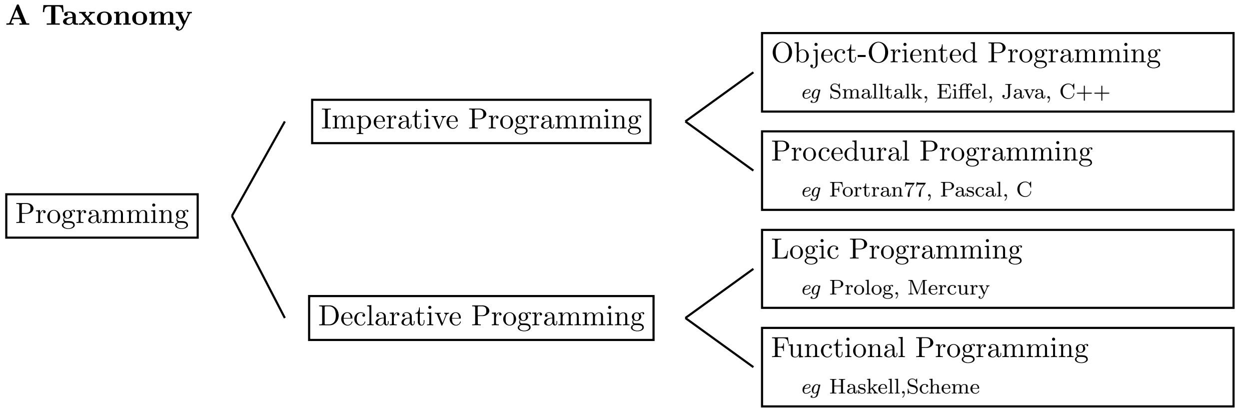
I.1 Programming and **C++**

I.1.1 Programming Paradigms

Paradigms

- A programming *paradigm* is an approach to programming:
  - what is a program ?
  - how are programs designed and written ?
  - how are the goals of programs achieved by their execution ?
- A *paradigm* is an **idea in pure form**.
- A *paradigm* is **realized** in a *language* by a set of constructs and by methodologies for using them.
- *Languages* sometimes combine several *paradigms*.

A Taxonomy



I.1.2 Imperative Programming

Memory Cells

- *Imperative programming* is based on a simple construct: the *memory cell*.
- A *memory cell* is a *container* of *values*, whose contents can be changed.
- A *memory cell* is an **abstraction** of a *memory location*.

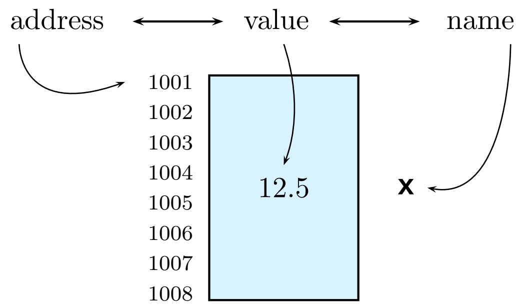
  The abstraction hides the size bounds and the physical address space details that apply to real memory.

- The *values* stored in *memory cells* are data such as integers, real numbers or addresses of other cells.

  A *value* is an **abstraction** that hides the details of binary representations used by computer hardware.

Illustration in C When talking about *memory cells*, we are dealing with **three** levels:

```
double x = 12.5 ;
```



- A program written using an *imperative programming language* is executed by following an **ordered sequence** of instructions.
- These instructions are usually termed *statements*.

  A *statement* is an **abstraction** that hides the details of *machine-level instructions* (for which the word "instruction" is reserved).
- *Statements* are able to manipulate *values* that are *stored*.
- A program describes the **steps of a process** in terms of a *program state* that is changed by the *statements*.
- The *program state* is divided into *control* and *data*.

  Control : in which step is the process ?

  Data : what are the *stored values* ?
- The order in which *statements* are executed can be managed using *control structures*.
- A program *execution* corresponds to a sequence of *states*.

I.1.3 Object-Oriented Programming

- In *Object-Oriented Programming (OOP)*, *values* and *statements* are packaged into *objects*.
- *OOP* shifts the emphasis from *values* as passive elements acted on by *statements* to active elements, the *object*, interacting with their environment.

> **OBJECT MOTO**
> Ask not first what a system does. Ask what it does it to.

- The notions of *class* and *inheritance* are introduced.

  Using them has a major impact on the programming style.

  ⇒ *OOP* is considered a **separate paradigm**.
- Yet, like in *imperative paradigm*, the notion of *program state* remains central, and a program execution still corresponds to a sequence of states.

  ⇒ *OOP* still deals with the two constructs: *memory cells* and *statements*.

I.1.4 The **C++** Language: Major and Missing Features

- Not everyone agrees about what *Object-Oriented (OO)* means.

  ...*inheritance*, *polymorphism*, *encapsulation*, *exception handling*, *templates*...

- **C++** is a multiparadigm language:
  - **C++** supports many *OO* features.
  - It is possible to write completely non-*OO* programs in **C++**, and many people do.

    (*generic programming* using the **powerful** abstraction mechanism of *templates*)

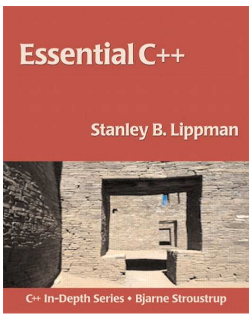
- **C++** lacks major features for *Component-Based Development*:
  - it does not permit *seamless development*;
  - it provides no facility to deal with *component specifications*

    (no native support of *Design by Contract*);
  - it ignores the *documentation* issues.

- **C++** has some pleasant features:
  - standardized since 1998 (ISO/IEC 14882:1998)
  - relative efficiency for scientific calculations
  - availability of quality free compilers for a wide variety of hardware platforms

> No language is the be-all and end-all. Today, I'm using **C++** as my primary programming language; tomorrow, I'll use whatever best suits what I'm doing them. [Herb Sutter]

I.1.5 About **C++** Books and Courses...



"Essential C++ is an excellent introduction to C++ for experienced programmers, a wonderfully deep book for so wonderfully brief a book, and a unique book that storms the heights of theory without exhausting the interest of the reader."
J.J. Woehr, *Dr Dobb's Electronic Review of Computer Books.*

"I'm often asked to recommend the **best** C++ book [...]. There are dozens [...]. In the future, however, I'll have no trouble coming up with the **worst** C++ book I've seen. If there's a worse one out there than Essential C++ I hope I never encounter it. Stanley Lippman confronts his readers with a stream of silly examples, grotesquely misguided coding techniques, opaque writing, and just plain errors. This book has no redeeming qualities and is appropriate for no audience."
C.Weisert, *Information Disciplines, Inc.*

> **Conclusion:** Not everyone agrees about ...
> Each viewpoint has its passionate defenders ...

I.2 Getting Started with C++

I.2.1 A First Program - Hello World

```
// a small C++ program
//    A. Koenig, B. Moo
//    Accelerated C++, Addison-Wesley, 2000

#include <iostream>

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

I.2.2 A Second Program - Write a Greeting

```
// ask for a person's name, and greet the person
//    A. Koenig, B. Moo
//    Accelerated C++, Addison-Wesley, 2000

#include <iostream>
#include <string>

int main( void )
{
   // ask for the person's name
   std::cout « "Please enter your first name: " ;

   // read the name
   std::string name ;    // define name
   std::cin » name ;    // read into name

   // write a greeting
   std::cout « "Hello, " « name « "!" « std::endl ;
   return 0 ;
}
```

I.2.3 Steps from Edition to Execution

- The text of a **C++** program consists of one or more *source files*.
- **C++** programs typically go through 6 phases to be executed:
  Edition, Preprocessing, Compilation, Link, Load, Execution

Edition

source files are created in an *editor* and stored on disk.

Preprocessing

For each *source file*, a *preprocessor*

- examines all lines beginning with a **#** (first character other than *white space*);
- performs the corresponding actions and macro replacements;
- produces a preprocessed version, called *translation unit*.

Compilation

The *compiler* translates each *translation unit* into an *object files* (**.o**) made of machine language code.

Link

A *linker* produces an *execution file* with no missing pieces:

- it takes all the independent *object files* and links them together into the execution file;
- it searches libraries to resolve external references, and
  - links needed library routines from static archives (typically with the **.a** extension) into the execution file (like any other object file);
  - includes a reference to needed library routines from *dynamic libraries* (also called *shared libraries*, typically with the **.so** extension) into the execution file.

Load

A *loader* reads instructions from the *execution file* and stores them into *memory* for subsequent execution

Execution

A *computer*, under the control of its CPU, executes the program **one instruction at a time**. When a reference to a dynamic library routine is first executed, the routine in loaded into memory.

I.2.4 Lexical Conventions

Tokens

- A *translation unit* (*i.e.* a preprocessed **C++** source file) is a sequence of *tokens* (or *lexical elements*).
- Blanks, tabs, newlines, formfeeds and comments (collectively *white space*) are ignored except as they serve to separate tokens.
- A *token* is the smallest element of a **C++** program that is meaningful to the compiler.
- There are 5 kinds of *tokens*: *identifiers*, *keywords*, *literals*, *operators*, *punctuators*.

Identifiers

- arbitrarily long sequence of letters and digits
- first character: letter (underscore _ counts as a letter)
- Upper- and lower-case letters are different:
  MyObject is a different identifier than **myobject**
- identifiers beginning with a single underscore (_) or containing a double underscore (__) are reserved to **C++** implementations and standard libraries

Keywords

- Keywords are words reserved as part of the language.
- They cannot be used by the programmer to name things.
- They consist of lowercase only.
- They have special meaning to the compiler.

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

Operators and Punctuators

{	}	[	]	#	##	(	)	
<:	:>	<%	%>	%:	%:%:	;	:	...
new	delete	?	::	.	.*			
+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	,	->*	->
and	and_eq	bitand	bitor	compl	not	not_eq	or	or_eq
xor	xor_eq							

Literals

- The term *literal* generally designates those *tokens* that are called *constants* in ISO C.
- There are several kinds of literals: integer literals, character literals, floating literals, string literals and boolean literals.

I.2.5 Objects, Functions

Objects

- An *object* is a part of the computer's memory that has a *type*.
- The *type of an object* specifies a *data structure* and the meaning of *operations* that make sense for the data structure.
- All objects of a given type occupy the same amount of storage.
- A *variable* is an *object* that has a *name*.

  It is possible to have objects that do not have names.
- A *reference* is an **alternative** *name* for a *variable*.

Functions

- A *function* is a piece of program
  - that has a *name*,
  - that another part of the program can *call*, *ie* cause to run.
- The *type of a function* is given by its *return* type and the type of its *parameters*.

I.3 Types and Declarations

I.3.1 Names

Names are used to **denote** program values or components.

- An *entity* is a value, object, subobject, base class object, array element, variable, function, instance of a function, enumerator, type, class member, template or namespace.
- A *name* is an *identifier* that denotes an *entity* (or a *label*... ignored thereafter).
- Before a name can be used, it must be declared: every name is introduced by a *declaration*.
- A *declaration* tells the compiler that a name exists and how it is to be interpreted.
- A *definition* is a unique specification of the entity denoted by the name to which it refers. It provides information that allows the compiler to allocate memory for objects or generate code for functions.
- Some *declarations* are also *definitions*.
- In *declarations* that are not *definitions*: the entity they refer to must be defined elsewhere.
- Names may be declared multiple times but must be defined only once.
- A *declaration* introduces a *name* into a *scope*: the *scope* of a *name* is the part of the program text where that name can be used.

I.3.2 Examples of Declarations and Definitions

```
std::string s ;                     //  defines s
int count ;                         //  defines count
const double pi = 3.14 ;            //  defines pi
int f( int x ) { return x+a ; }     //  defines  f  and defines  x
struct S { int a ; int b ; } ;      //  defines  S,  S::a, and  S::b
class X {                           //  defines  X
public:
    X( void ) : x(0) {}             //  defines a constructor of  X
    void f( double x ) ;            //  declares function member f
private:
    static int y ;                  //  declares static data member  y
    int x ;                         //  defines nonstatic data member  x
} ;
int X::y = 1;                       //  defines  X::y
enum Move{ up, down } ;             //  defines  Move, up and down
X anX ;                             //  defines  anX
extern const int a ;                //  declares  a
int f( int x ) ;                    //  declares  f
class S ;                           //  declares  S
typedef int Int ;                   //  declares  Int
extern X anotherX ;                 //  declares  anotherX
using N::d ;                        //  declares  N::d
```

I.3.3 Types

- *Types* describe *objects*, *references* or *functions*.
- Among the *types*, we distinguish:
  - those built into the core of the language (*eg* **double**);
  - those that are defined outside the core of the language (*eg* **std::ostream**).
- *Types* can be *cv-unqualified* or *cv-qualified*.
- Each *cv-unqualified type* has three corresponding *cv-qualified* versions:
  - a *const-qualified* version;
  - a *volatile-qualified* version;
  - a *const-volatile-qualified* version.
- There are two kinds of *cv-unqualified types*:
  - *fundamental* types;
  - *compound* types.

I.3.4 Fundamental Types

- four signed integer types: **signed char**, **short int**, **int**, **long int**
- four unsigned integer types: **unsigned char**, **unsigned short int**, **unsigned int**, **unsigned long int**
- the type **wchar_t**
- the type **bool**
- three floating point types: **float**, **double**, **long double**
- the type **void**

I.3.5 Compound Types

- arrays of objects of a given type
- functions
- pointers to void, pointer to objects of a given type, pointers to functions of a given type
- references to object of a given type or references to functions of a given type
- classes containing a sequence of *members* of different types
- unions
- enumerations
- pointers to non-static class members

I.4 Expressions and Statements

I.4.1 Expressions

- An *expression* is a sequence of *operators*, *operands* and *punctuators* that specifies a computation.
- The implementation *evaluates* an *expression* when it performs the defined computations.
- The *evaluation* of an *expression* is based on the operators that the expression contains, and the context in which the operators are used.
- The *evaluation* of an *expression*
  - yields a *result*,
  - may cause *side effects* (may affect the state of the implementation in a way that is not directly part of the result).
- A *condition* is an *expression* that yields a truth value.
- Understanding complicated *expressions* requires understanding:
  - how the operators group with the operands;
  - how the operands will be converted to other type, if at all;
  - the order in which the operands are evaluated.

I.4.2 Operators

- The effect of an *operator* depends on the *type* of its *operands*.
- When an *operator* has different meanings for different types of its *operands*, we say that the operator is *overloaded*.
- Each *operator* has the following characteristics:
  - *Valence*: number of operands.
  - *Operand types*.
  - *Return type*.
  - *Precedence*: priority for grouping different kinds of operators with their operands.
  - *Associativity*: left-to-right or right-to-left order for grouping operands to operators that have the same precedence.

- *Overloading* allows to programmer to define
  - the operand types
  - the return type
  - the behavior

  but not
  - the precedence
  - the valence
  - the associativity

I.4.3 Examples of Expressions

- Consider:

```
std::string n ; // definition, implicit initialization
std::cin » n ;
std::string h = "Hello " ; // definition, explicit initial value
std::string r( 3, '*' ) ;  // definition, initialization according
                           // to its type and the given arguments
```

$$\underbrace{\texttt{r = h + n + "!" + r ;}}_{\text{expression } e}$$

The `=` operator has lower precedence than arithmetic operators.

⇒ e means `r = ( h + n + "!" + r )`

The `+` operator is left-associative : when `+` appears twice or more in the same expression, each `+` will use as much of the expression as it can for its left operand.

⇒ `h + n + "!" + r` means `( ( h + n ) + "!" ) + r`

- Consider the expression

  `a = b = c`

The `=` operator is right-associative : when `=` appears twice or more in the same expression, each `=` will use as much of the expression as it can for its right operand.

⇒ `a = b = c` means `a = ( b = c )`

I.4.4 Table of Operators, Grouped by Precedence

In the following, when several operators are grouped together, they share the same *precedence* and *associativity* (L: left-to-right; R: right-to-left).

<table>
<tr><td>C::m</td><td>the member m from class C</td><td></td></tr>
<tr><td>N::m</td><td>the member m from namespace N</td><td></td></tr>
<tr><td>::m</td><td>the name m at global scope</td><td></td></tr>
<tr><td>x[y]</td><td>the element in object x indexed by y; lvalue</td><td>L</td></tr>
<tr><td>x->y</td><td>the member y of the object pointed by x; lvalue</td><td>L</td></tr>
<tr><td>x.y</td><td>the member y of object x; lvalue if x is an lvalue</td><td>L</td></tr>
<tr><td>f(args)</td><td>calls function f passing args as argument(s)</td><td>L</td></tr>
<tr><td>x++</td><td>increments the lvalue x; yields the original value of x</td><td>L</td></tr>
<tr><td>x--</td><td>decrements the lvalue x; yields the original value of x</td><td>L</td></tr>
<tr><td>*x</td><td>dereferences the pointer x; yields the object pointed to; lvalue</td><td>R</td></tr>
<tr><td>&x</td><td>the address of the object x; yields a pointers</td><td>R</td></tr>
<tr><td>-x</td><td>unary minus; may be applied only to expressions of numeric type</td><td>R</td></tr>
<tr><td>!x</td><td>logical negation; if x is zero, then !x is true, otherwise false</td><td>R</td></tr>
<tr><td>~x</td><td>ones complement of x; x must be an integral type</td><td>R</td></tr>
<tr><td>++x</td><td>increments the lvalue x; yields the incremented value; lvalue</td><td>R</td></tr>
<tr><td>--x</td><td>decrements the lvalue x; yields the decremented value; lvalue</td><td>R</td></tr>
<tr><td>sizeof(e)</td><td>the number of bytes, as a size_t, consumed by expression e</td><td>R</td></tr>
<tr><td>sizeof(T)</td><td>the number of bytes, as a size_t, consumed by objects of type T</td><td>R</td></tr>
<tr><td>T(args)</td><td>contructs a T object from args</td><td>R</td></tr>
<tr><td>new T</td><td>allocates a new, default-initialized object of type T</td><td>R</td></tr>
<tr><td>new T(args)</td><td>allocates a new object of type T initialized by args</td><td>R</td></tr>
<tr><td>new T[n]</td><td>allocates an array of n default-initialized objects of type T</td><td>R</td></tr>
<tr><td>delete p</td><td>frees object pointed to by p</td><td>R</td></tr>
<tr><td>delete [] p</td><td>frees the array of objects pointed to by p</td><td>R</td></tr>
<tr><td>x * y</td><td>product of x and y</td><td>L</td></tr>
<tr><td>x / y</td><td>quotient of x and y</td><td>L</td></tr>
<tr><td>x % y</td><td>x-((x/y)*y)</td><td>L</td></tr>
<tr><td>x + y</td><td>sum of x and y, if both operands are numeric</td><td>L</td></tr>
<tr><td>x - y</td><td>result of substracting y from x if operands are numeric.</td><td>L</td></tr>
<tr><td>x shiftop y</td><td>shiftop : >> or <<</td><td>L</td></tr>
<tr><td>x <i>relop</i> y</td><td><i>relop</i> : <, >, <= or >=</td><td>L</td></tr>
<tr><td>x == y</td><td>yields a bool indicating whether x equals y</td><td>L</td></tr>
<tr><td>x != y</td><td>yields a bool indication whether x is not equal y</td><td>L</td></tr>
<tr><td>x & y</td><td>bitwise and; x and y must be integral</td><td>L</td></tr>
<tr><td>x ^ y</td><td>bitwise exclusive or; x and y must be integral</td><td>L</td></tr>
<tr><td>x | y</td><td>bitwise or; x and y must be integral</td><td>L</td></tr>
<tr><td>x && y</td><td>yields a bool indicating whether both x and y are true
(evaluates y only if x is true)</td><td>L</td></tr>
<tr><td>x || y</td><td>yields a bool indicating whether either x or y are true
(evaluates y only if x is false)</td><td>L</td></tr>
<tr><td>x = y</td><td>assigns the value y to x; yields x, lvalue</td><td>R</td></tr>
<tr><td>x op= y</td><td>equivalent to x = x op y where op is an arithmetic, bitwise, or shift operator</td><td>R</td></tr>
<tr><td>x ? y1 : y2</td><td>yields y1 if x is true; y2 otherwise
(only one of y1 or y2 is evaluated)</td><td>R</td></tr>
<tr><td>throw x</td><td>signals an error by throwing value x (the type of x determines which handler will catch the error)</td><td>R</td></tr>
<tr><td>x , y</td><td>evaluates x, discard the result, then evaluates y; yields y</td><td>L</td></tr>
</table>

I.4.5 Order of Evaluation of the Operands

- Only four operators guarantee the order of evaluation of their operands:

`&&`	The right operand is evaluated only if the left operand is **`true`**.
`||`	The right operand is evaluated only if the left operand is **`false`**.
`? :`	Only one expression after the condition will be evaluated. The expression after the **`?`** is evaluated if the condition is **`true`**; otherwise, the expression after the **`:`** is evaluated.
`,`	The left operand is evaluated first.

- *Short-circuit evaluation* refers to the condition where operands of an expression are no longer evaluated since further evaluation cannot change the value of the expression. The (only) three operators **`&&`** , **`?:`** , **`||`** use a *short-circuit evaluation* strategy.
- For the other operators, order of evaluation of their operands is not guaranteed.

```
string s = ... ;
string::size_type i = 0 ;
while( i!=s.size() && isspace(s[i]) ) // relies on the short-circuit
   ++i ;                              // property of &&
if( i!=s.size() )
   cout « "first non whitespace character: " « s[i] « endl ;
```

I.4.6 Statements

- A *statement* is the minimal unit of structuring.
- *Statements* contrast with *expressions* in that
  - they do not return results;
  - they are solely executed for side effects.
- Every *statement* appears inside the *definition* of a function, where it forms part of what happens when that function is called.
- Most *statements* end with semicolons (main exception: the block).
- A *statement* is either:
  - a labeled statement;
  - an expression statement;
  - a compound statement (equivalently called block);
  - a selection statement;
  - an iteration statement;
  - a jump statement;
  - a declaration statement;
  - a try-block.

I.4.7 Labeled, Expression and Compound Statements

Labeled Statements

`case` *value* **`:`** *statement*

`default` **`:`** *statement*

- shall occur only in **`switch`** statements

Expression Statements

expression **;**

- evaluates *expression* for its side effects

Compound Statements or Blocks

{ *statement(s)* **}**

- executes the sequence of zero or more *statement(s)* in order
- may be used wherever a *statement* is expected.
- variables defined inside the braces have scope limited to the block.

I.4.8 Selection Statements

if(*condition* **)** *statement1*

- evaluates *condition* and executes *statement1* if *condition*'s evaluation yielded **true**

if(*condition* **)** *statement1* **else** *statement2*

- evaluates *condition* and executes *statement1* if *condition*'s evaluation yielded **true**; otherwise executes *statement2*

```
int main( void ) {
   int exit_code = 0 ;
   std::ifstream ff( "toto.txt" ) ;
   if( ff ) {
      /* ... */
      exit_code = 0 ;
   }
   else {
      std::cout « "unable to open toto.txt" « std::endl ;
      exit_code = 1 ;
   }
   return exit_code ;
}
```

if-**else** statement may be "glued" together.

```
if( nbr < 0 )
{
   cout « nbr « " is negative" « endl ;
}
else if( nbr > 0 )
{
   cout « nbr « " is positive" « endl ;
}
else
{
   cout « nbr « " is zero" « endl ;
}
```

switch(*expression* **)** *statement*

- *statement* : almost always a *block*
  - that includes *labeled statements* of the form

    case *value* **:** *stmt*

    where each *value* is an *integral constant-expression*
  - that may include one, and only one, *labeled statement* of the form

    default : *stmt*
- evaluates *expression* and jumps to the **case** label whose values matches the evaluation result, if any; otherwise passes control to the **default:** label if any, or to the point immediately after the entire **switch** statement
- **case** labels are just labels: control will flow from one to the next unless the programmer takes explicit action to prevent it from doing so, *eg* using a **break** *statement* before each **case** label after the first

```
ifstream ff( "expr.txt" ) ;
if( ff ) {
   double xl, xr ;
   char op ;
   ff » xl » xr » op ;
   switch( op ) {
      case '+' :
         cout « xl + xr « endl ;
         break ;
      case '-' :
         cout « xl - xr « endl ;
         break ;
      case '*' :
         cout « xl * xr « endl ;
         break ;
      case '/' :
         cout « xl / xr « endl ;
         break ;
      default :
         cout « "illegal operation" « endl ;
   }
}
```

I.4.9 Iteration Statements

while(*condition* **)** *statement*

- evaluates *condition* and executes *statement* as long as the evaluation result is **true**

do *statement* **while(** *condition*)

- executes *statement* and then evaluates *condition*; continues executing *statement* until condition is **false**.

```
std::ifstream ff( "nombres.txt" ) ;
if( ff )
{
   std::vector<double> vals ;
   double x = 0.0 ;
   while( ff » x )
   {
      vals.push_back( x ) ;
   }
   /* ... */
}
```

for(*init-statement* *condition* **;** *expression*) *statement*

- executes *init-statement* once on entry to the loop and then evaluates *condition*; if the evaluation result is **true**, executes *statement* and then evaluates *expression*; continues evaluating *condition*, followed by executing *statement* and evaluating *expression*, until the result of the evaluation of *condition* is **false**
- if *init-statement* is a *declaration*, then the *scope* of the *variable* is the *statement* only.
- equivalent *block statement*:

```
{
   init-statement
   while( condition ) {
      statement
      expression ;
   }
}
```

Example

- Let **vals** be a variable of type **std::vector<double>**
- Write the stored value onto **std::cout**:

```
for( vector<double>::size_type i = 0 ;
     i != vals.size() ;
     ++i )
{
   cout « vals[i] « endl ;
}
```

- Same output:

```
for( vector<double>::const_iterator it = vals.begin() ;
     it != vals.end() ;
     ++it )
{
   cout « *it « endl ;
}
```

- Equivalently:

```
{
   vector<double>::const_iterator it = vals.begin() ;
   for( ; it != vals.end() ; ++it )
   {
      cout « *it « endl ;
   }
}
```

I.4.10 Jump Statements

`break ;`

- jumps to the point immediately after the end of the nearest enclosing **`while`**, **`for`**, **`do`**, or **`switch`** statement

`continue ;`

- jumps back to the beginning of the next iteration (including the test) in the nearest enclosing **`while`**, **`for`**, **`do`**, or **`switch`** statement
- if **`for`**, the next iteration includes the *expression* in the **`for`**-*statement* as well.

```
for( int i=0 ; i!=8 ; ++i )
{
   if( i%3 == 0 ) continue ;
   cout « i « endl ;
}
```

⇝

```
1
2
4
5
7
```

`return` *expression* **`;`**

- evaluates *expression*, exits the function, returns the evaluation result to the function's caller
- *expression*: empty for functions declared **`void`**

```
int  f( ... ) { return ; }  // illegal: return value missing

int fac( int n ) { return (n>1) ? n*fac(n-1) : 1 ; }
```

I.4.11 Declaration Statements

Generalities

decl-specifiers [declarator [initializer]] [**,** *declarator [initializer]] ...* **;**

- **C++** inherits its declaration syntax from **C**.
- A declaration consists of:
  - a sequence *decl-specifiers*, that collectively specify one type (and other attributes),
  - followed by zero or more *declarators*, each optionally followed by an *initializer*.
- A declaration declares an *entity* for each *declarator*, giving that entity a *name* (through the *declarator*), and a type with other attributes (through the *decl-specifiers*).
- To understand any declaration, first locate the boundary between the specifiers and the declarators.
- Specifiers: *keywords* or *names* of *types*.

  ⇒ specifiers end just before the first symbol that isn't one of those.

$$\overbrace{\texttt{const char}}^{\text{specifiers}}\overbrace{\texttt{* const* const* cp}}^{\text{(only) declarator}}\ ;$$

$$\overbrace{\texttt{char}}^{\text{specifier}}\ \overbrace{\texttt{* kings[]}}^{\text{declarator}}\ \texttt{=}\ \overbrace{\texttt{\{ "Antigonus", "Seleucus", "Ptolemy" \}}}^{\text{initializer}}\ ;$$

Specifiers

decl-specifiers	:	{ *type-specifier* | *storage-specifier* | *other-decl-specifier* } ...
type-specifier	:	**char** | **wchar_t** | **bool** | **short** | **int** | **long** | **signed** | **unsigned** | **float** | **double** | **void** | *type-name* | **const** | **volatile**
type-name	:	*class-name* | *enum-name* | *typedef-name*
storage-specifier	:	**register** | **static** | **extern** | **mutable**
other-decl-specifier	:	**friend** | **inline** | **virtual** | **typedef**

- *decl-specifiers*: can appear in any order
- *type-specifier*: determines the type that underlies any declaration
- *storage-specifier*: determines the location and lifetime of a variable
- *other-decl-specifier*: defines properties that are not related to types

Declarators

- A *declarator* is composed of:
  - a *name* possibly qualified with nested *scope names* ;
  - optionaly some *declarator operator*.
- Most common *declarator operators*:

`*`	pointer	prefix
`* const`	constant pointer	prefix
`&`	reference	prefix
`[]`	array	postfix
`()`	function	postfix

- A **const** that is part of a *declarator* always follows a `*`

  ⇒ no ambiguity with the **const** *specifier*.
- `*`, `[]`, `()` were designed to mirror their use in expressions.
- The postfix *declarator operators* bind tighter than the prefix ones.

`int (*f)( char* c ) ;`	**f** : pointer to function taking a **char*** argument and returning an **int**
`int  *f ( char* c ) ;`	**f** : function taking a **char*** argument and returning a pointer to **int**

Examples

```
int list[20] ;                // vector of 20 int values
char* cp ;                    // pointer to char
double func( void ) ;         // function returning a double,
                              // with no argument

int* aptr[10] ;               // array of 10 pointers to int
int *aptr[10] ;               // same as before
int (*var)[5] ;               // pointer to vector of 5 int values

long *var( long, long ) ;     // function returning pointer to long
long (*var)( long, long ) ;   // pointer to function returning long

int* p, y ;      // pointer to int named p and int value named y
int *p, y ;      // same as before
int v[10], *p ; // vector of 10 int values and pointer to int
```

- You should not write this: `char *(*(*var)())[10] ;`
  but you may understand it: `char * ( * ( * var ) () ) [10]`

```
char   *   (   *   (   *   var   )   ()   )   [10]
 ↓     ↓       ↓       ↓    ↓        ↓         ↓
 7     6       4       2    1        3         5
```

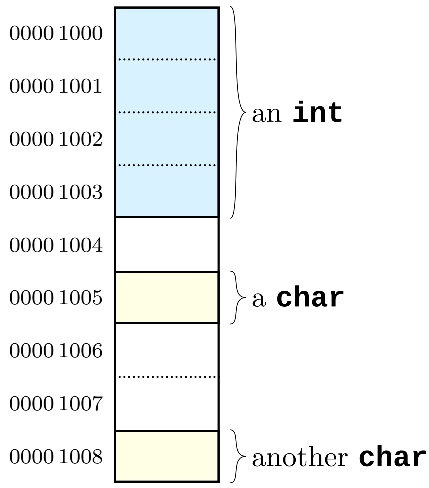
1 : indentifier **var** is declared as
2 : a pointer to
3 : a function returning
4 : a pointer to
5 : an array of 10 elements, which are
6 : pointers to
7 : char variables

I.5 Pointers and References

I.5.1 Pointers

Memory

- Memory consists of *bytes*.
- A *byte* can store an eight bit value.
- A *memory cell* is an abstraction of the range of *bytes* occupied by its stored *value*.



Objects in Memory

- Every *object* occupies a *memory cell* corresponding to a range of *bytes* determined by the *object type*.
- Every *object* has a unique *address* defined as the address of the first byte it uses

Pointers

- A *pointer* is a *value* that represents the *address* of an *object*.
- As with all **C++** *values*, pointers have *types*.

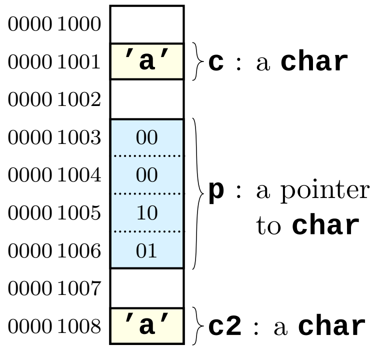
  The address of an object of type **T** has type “pointer to **T**”, written as **T*** .
- Pointers to different types are themselves different types (even though under the hood, they’re just memory addresses)

I.5.2 Two Important Pointer Operators

- If you can access an *object*, you can obtain its *address*, and vice versa.
- If **x** is an object, **&x** is the address of that object.
  - **&** is called the *address-of operator*.
- If **p** is the address of an object, ***p** is the object itself.
  - Refering to the *object* pointed to by a *pointer* is called *dereferencing* or *indirection*.
  - ***** is called the *dereference operator*.

- Pointers in memory:

```
char  c = 'a' ;

char* p = &c ;
// p holds the address of c

char c2 = *p ;
// c2 == 'a'
```



I.5.3 Values of Pointer Variables

- During program execution, a *pointer variable* can be in one of the following states:
  - contain no meaningfull value (uninitialized);
  - refer to an object;
  - contain the special *value* **0**.
- **0** acts as a pointer literal, indicating that the pointer doesn't refer to any object:
  - The literal **0** is the only integer value that can be converted to a pointer type.
  - No object has the address **0**.
- Assigning **0** to a *pointer* does **not** mean "make it point to the address **0** of the memory".
- The special value **0** is useful in comparisons.
- As with other built-in types, a local variable of type pointer has no meaningfull value until we give it one. To avoid uninitialized pointers, use the special value **0**.

```
int*   p1 = 0 ;       // pointer to int
char** p2 = 0 ;       // pointer to pointer to char
```

- The possible values of any pointer variable is:
  - the set of all memory addresses
  - along with the special value **0**.

I.5.4 Constants and Pointers

- A constant pointer is a pointer such that we cannot change the location to which it points.

```
char c = 'c' ;
char const d = 'd' ;
char* const p1 = &c ;
*p1 = 'cc' ; // legal: c == 'cc'
p1 = &d ;    // illegal
```

- A pointer to a constant value is a pointer object such that the value at the location to which the pointer points is considered constant.

```
char const* p2 = &d ;
p2 = &c    ; // legal
*p2 = 'e' ; // illegal: cannot change d
            // through indirection with p2
```

- Both the pointer and the referred object can be constants:

```
char const* const p3 = &c ;
p3 = &d ;   // illegal
*p3 = 'x' ; // illegal
```

I.5.5 Dynamic Memory Allocation

- The **new** operator is used to create dynamically allocated objects.
- A dynamically allocated object is deallocated with the **delete** object (if not, it stays around until the program ends).

expression	result and side effect of the evaluation
`new T`	- *Allocate* a new object of type **T**. - *Default-initialize* this newly allocated object. - Return a pointer to this unnamed object.
`new T(` *args* `)`	- *Allocate* a new object of type **T**. - *Initialize* this newly allocated object using *args*. - Return a pointer to this unnamed object.
`delete p`	**p must** point at a dynamically allocated object. - *Destroy* the object to which **p** points. - *Free* the memory used to hold ***p**. Remark: if the value of **p** is **0**, **delete p** has no effect.

Examples

`int* p = new int( 42 ) ;`	- An unnamed new object of type **int** is allocated, - its value is initialized to **42**. - A local pointer variable of name, **p** is created, - its value is initialized so as to point to the dynamically allocated object.
`++*p ; // *p is now 43`	The value of the new object is modified.
`delete p ;`	The space occupied by ***p** is freed and **p** becomes an invalid pointer that can no longer be used until a new value is assigned to it.

`int* create_int( void ) { return new int( 0 ) ; }`	- A function is defined that allocates an **int** object, initializes it to zero and returns a pointer to it. - It imposes on its called the responsibility of freeing the object at an appropriate time.

I.5.6 References

- A *reference* is an alternative *name* for an *object*.
- The notation **T&** means "reference to **T**" (where **T** is a type)

```
int i = 1 ;
int& r = i ;  // r and i now denote to the same int
int x = r ;   // x = 1
r = 2 ;       // now i = 2
```

- **T const&** is a type of references that may not be used to change the denoted value (of type **T**). Usually used to avoid cost of copying a parameter to a function.
- A reference must **always** refer to some object. It has to be initialized.

```
std::string& rs ; // illegal: references must be initialized

std::string s( "xyzzy" ) ;
std::string& rs = s ;   // ok, rs and s denote the same object
extern std::string& r ; // ok, r defined elsewhere
```

- A reference **always** refer to the object with which is was initialized.

```
string s1( "Nancy" ) ;
string s2( "Clancy" ) ;
string& r = s1 ;   // r and s1 denote the same object
r = s2 ;           // r and s1 still denote the same object
                   // but s1's value is now "Clancy"
```

I.6 Functions

I.6.1 Organizing Programs

- **C++** offers two fundamental ways of organizing large programs:
  - *functions* (sometimes called *subroutines* or *routines*);
  - *data structures*.
- In addition, **C++** let programmers **combine** *functions* and *data structures* into a **single notion** called a *class*.
- *Functions* defined as part of a *class* are called *function members* or *methods*.

> Since the organizational unit of PELICANS is the *class*, we will introduce *functions* only for the sake of preparing discussions on *function members*.

- A *function* is a piece of program
  - that has a *name*,
  - that another part of the program can *call*, *ie* cause to run.
- A function must be *declared* in *every source* file that uses it, and *defined* only once.

I.6.2 Function Declarations and Definitions

- A function cannot be called unless it has been previously *declared*.
- Every function that is called must be *defined* somewhere (once only).
- A function declaration
  - specifies the *return type*,
  - followed first by the function *name*,
  - next by a *parameter list* enclosed in **( )** (which gives the number and type of the arguments that must be supplied in a call),
  - finally, if it is also a *definition*, by the *function body* enclosed in **{ }**.
- **void** as a return type means that the function does not return a value.
- All declarations of a function should be consistent with their definition.

```
// a declaration
int fac( int n ) ;

// a definition
int fac( int n )
{
   return (n<2) ? 1 : n*fac(n-1) ;
}
```

I.6.3 Function Calls

Essentials

- A *function call* (or *function invocation*) is an *expression* defined as:
  - a *postfix expression* (generally a *function name*)
  - followed by parenthesis containing a possibly empty comma separated list of *expressions* called *arguments*.
- When a function is called:
  - store (called *activation record*) is set aside for its *parameters*, local *variables* and possibly other information (*eg* a pointer to the current statement being executed and a pointer to the invoking statement). ;
  - each parameter is initialized with the value yielded by the evaluation of its corresponding argument in the calling expression.
  - The *flow of control* is temporarily transfered to the invoked function: the next statement executed is the first one of the invoked function.
- After function completes its action:
  - The return value is used as the *result* of the *evaluation* of the *function call*.
  - The *activation record* memory is automatically released.
  - The *flow of control* is returned to the invoking function.

Further Explanations

- Calling a function involves **copying** the arguments' value into the parameters.

  $\Rightarrow$ **Call By Value**.
- The order of evaluation of the arguments is unspecified.

- *Parameters* behave like *variables* that are local to the function:
  - calling the function creates them with the corresponding argument value as an initial value;
  - returning from the function destroys them.
- The semantics of *argument passing* and *function value return* are identical to the semantics of *initialization*.
  - It is different from the semantics of *assignment*. This is important for:
    - `const` arguments;
    - reference arguments;
    - arguments of some user-defined types.
  - Implicit type conversions (standard and user-defined) are performed.

I.6.4 Functions Parameters

Non `const` Reference Parameters

```
void f( int val, int& ref )
{
   val++ ;
   ref++ ;
}
```

When `f()` is called,

- `val++` increments a local copy of the first actual argument;
- `ref++` increment the second actual argument.

For example:

```
int i = 1 ;
int j = 1 ;
f( i, j ) ; //  i==1 and j==2
```

will increment `j` but not `i` .

Remark

functions that modify arguments can make programs hard to read and should generally be avoided.

`const` Reference Parameters

- The copy of a "large" argument into a parameter may be noticeably inefficient.
- Saying that a *parameter* is a `const` *reference*
  - gives a **direct access** to the associated argument, **without copying** it;
  - promises that the *parameter*'s *value* won't change (which would otherwise change the *argument* too).

```
// the entire argument is copied into the parameter vec
double median( vector<double> vec )
{
   typedef vector<double>::size_type vec_sz ;
   vec_sz size = vec.size() ;
   sort( vec.begin(), vec.end() ) ;
   vec_sz mid = size/2 ;  // modifies vec
   return size % 2 == 0 ? (vec[mid]+vec[mid-1])/2 : vec[mid] ;
}

// the argument of type vector<double> will not be copied
// but directly accessed without modification
double grade( double mid, double final, vector<double> const& hw )
{
   return 0.2 * mid + 0.4 * final + 0.4 * median( hw ) ;
}
```

Three Kinds of Function Parameters

Let **T** be an *object type* (*eg* **std::vector<double>**) involved in the declaration of a *parameter* **p** of a function **foo**.

- If **p** is of type **T** :

  ```
  ... foo( ..., T p, ... )  ;
  double median( vector<double> vec ) ;
  ```

  - calling **foo** causes the *argument*'s value to be copied into **p**;
  - subsequent modification of **p** will not affect its associated argument.

- If **p** is of type **T const&**:

  ```
  ... foo( ..., T const& p, ... )  ;
  double grade( vector<double> const& vec ) ;
  ```

  - the **&** asks the implementation not no copy the *argument* associated to **p**,
  - the **const** promises that the program will not change **p**.

- If **p** is of type **T&**:

  ```
  ... foo( ..., T& p, ... )  ;
  istream& read_hw( istream& in, vector<double>& vec ) ;
  ```

  - the **&** asks the implementation not no copy the *argument* associated to **p**,
  - the **absence** of **const** means that the function **intends to change** the argument's value (by changing **p**).

I.6.5 Possible Errors in Value Return

The *activation record* is released after the function returns, so it is **unwise** to return a pointer or reference to a local variable.

C++ implementations are not required to diagnose the following errors:

$\Rightarrow$ you get what you get.

- Negative example: **don't do this!**

```
int* foo( void ) {
   int x ;
   // ...
   return &x ; // instant disaster!
}
```

- Negative example: **don't do this!**

```
int& foo( void ) {
   int x ;
   // ...
   return x ; // instant disaster!
}
```

I.6.6 The `main` Function

- A *program* shall contain a global function called **`main`**, which is the designated start of the program.
- There are two possibilities for the definition of **`main`**
  - if the program does not expect any arguments from the running environment:

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

  - if **`main`** is willing to accept a sequence of character strings as an argument from the running environment:

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

- **`main`** shall not be called from within the program.

I.7 Name Resolution

I.7.1 Scopes

- The *scope* of a *name* is the portion of the program text in which it may be used as an *unqualified name* to refer to the same *entity*.
- Defining the scope requires the following notions:
  - The *declarative region* of a name is the *construct* where it is declared.
  - The *point of declaration* of a *name* is the point in the software text after the complete *declarator* and before the *initializer*.
  - The *potential scope* of a *name* extends from its *point of declaration* to the end of its *declarative region*.
- The *scope* of a *name* is its *potential scope* unless the potential scope contains another declaration of the same name.

Illustrative Example (but don't do this!)

```
int k = 24 ;
int main( void )
{
   int i = k, k ;
   k = 42 ;
}
```

first **k**

- *declarative region*: the entire example
- *potential scope*: after the first **k** to the end
- scope: excludes the text between **,** and **}**

second **k**

- *declarative region*: all the text between **{** and **}**
- *potential scope*: excludes the declaration of **i**
- *scope*: same as its *potential scope*.

Six Kinds of Scope

- global scope (or file scope);
- local scope (or block scope);
- function prototype scope;
- function scope;
- class scope;
- namespace scope.

Global Scope

- The *potential scope* of a *name declared* outside any *function*, *class* or *namespace* extends from its *point of declaration* to the end of the *file* in which its *declaration* occurs.

Local Scope

- The *potential scope* of a *name declared* in a *block* extends from its *point of declaration* to the end of the *block* in which its *declaration* occurs.
- The *potential scope* of a *function parameter name* in a *function definition* extends from its *point of declaration* to the end of the outermost block of the function definition.
- *Names* declared in the *init-statement* of a **for** statement, and in the *condition* of **if**, **while**, **for**, and **switch** statements are local to the **if**, **while**, **for**, or **switch** statement.

Namespace Scope

- A *namespace* is a named *scope*.
- A *namespace* is open: you can add names to it from several *namespace declarations*.
- The *standard library* defines all its *names* in a namespace named **`std`**.

Class Scope

- A *class* is a *type* defined by a named *scope* that describes how *objects* of that *type* can be created and used.
- The *potential scope* of a *name* declared in a *class definition* is composed of:
  - the region of the *class definition* following the *name*'s *declarator*;
  - all *function member* bodies and default arguments in that class;
  - all *constructor initializers* in that class.

I.7.2 Name Hiding

- *Hiding* occurs when a *name* is *declared* in a *declarative region* whereas the same *name* was already *declared* in an **enclosing** *declarative region*.
- In that case, the *potential scope* of the *name declared* in the **inner** (contained) *declarative region* is excluded from the *scope* of the (same) *name declared* in the **outer** (containing) declarative region.
- *Name hiding* is unavoidable but should be **minimized**.

```
int x ;             // global x
void f( void )
{
   int x ;          // local x hides global x
   x = 1 ;          // assign to local x
   {
      int x ;       // hides first local x
      x = 2 ;       // assign to second local x
   }
   x = 3 ;          // assign to first local x
}
int* p = &x ;       // take address of global x
```

I.7.3 Scope-Resolution Operator

- The scope-resolution operator `::` is used to qualify names in order to specify which scope to use when that name is looked up.
- A name prefixed by the unary scope-resolution operator `::` is looked up in *global scope*, in the *translation unit* where it is used.

```
int x ;
void f( void ) {
   int x = 1 ; // hide global x
   ::x = 2 ;   // assign to global x
   x = 2 ;     // assign to local x
}
```

- Names used with the binary scope-resolution operator `::` are called *qualified names*.
  - To the left of the `::` is the (possibly qualified) name of a *scope*.
  - To the right of the `::` is a name that is defined in the *scope* named on the left.

`std::cout` means: “the name **`cout`** that is in the (namespace) scope **`std`**”

`PDE_LocalFE::N` means: “the name **`N`** that is in the (class) scope **`PDE_LocalFE`**”

I.7.4 `using` declarations

- A ***`using`**-declaration* is a *statement* of the form:

  `using` *namespace-name*`::`*name* **`;`**

  that defines *name* as a synonym for *namespace-name*`::`*name* into the *declarative region* in which the ***`using`**-declaration* appears.

```
#include <vector>  //STL vector; belongs to namespace std
void foo( void ) {
  using std::vector ;  // using declaration
  vector<int> vi ;     // instead of std::vector<int>
... } // the above using declaration goes out of scope here
```

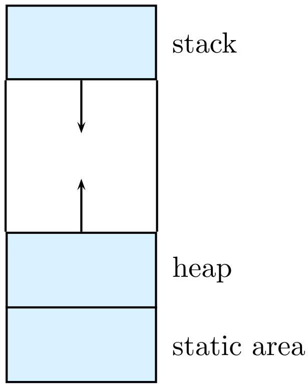
- A ***`using`**-directive* is a statement of the form:

  `using namespace` *namespace-name* **`;`**

  that instructs the compiler to recognize all names of the namespace *namespace-name*.

```
#include <vector>   // STL vector; belongs to namespace std
#include <iostream> // iostream declarations, in namespace std
void foo( void )  {
  using namespace std ; // using directive
  vector<int> vi ;      // instead od std::vector<int>
  vi.push_back( 10 ) ;
  cout « vi.begin() ;  // instead of std::cout
}
```

I.8 Storage and Memory Management



I.8.1 Memory Layout

- The part of the memory used by a program is devided into 3 parts
  - Static area: instruction of the program; static objects.
  - Stack: automatic objects.
  - Heap: dynamic objects.
- The stack and the heap grow towards each other, reducing the amount of unallocated memory

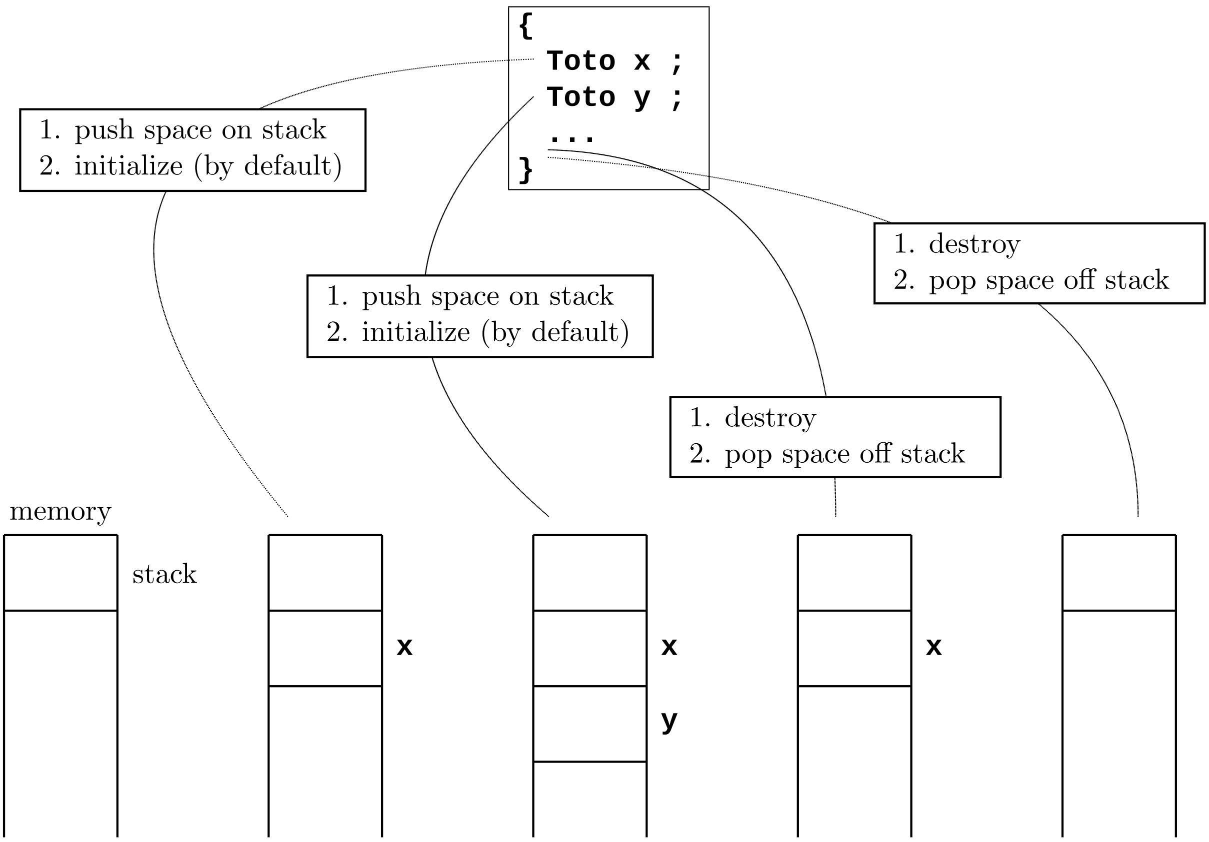
I.8.2 Static Area

- The size of any static object can be determined at compile time.
- Storage of static objects is allocated by the linker.
- Static objects are initialized at load time.
- Static objects are created just before the program begins to execute.
- Static objects are destroyed just before the program terminates.
- There is little or no control over the order in which static objects are created or destroyed.

I.8.3 The Stack

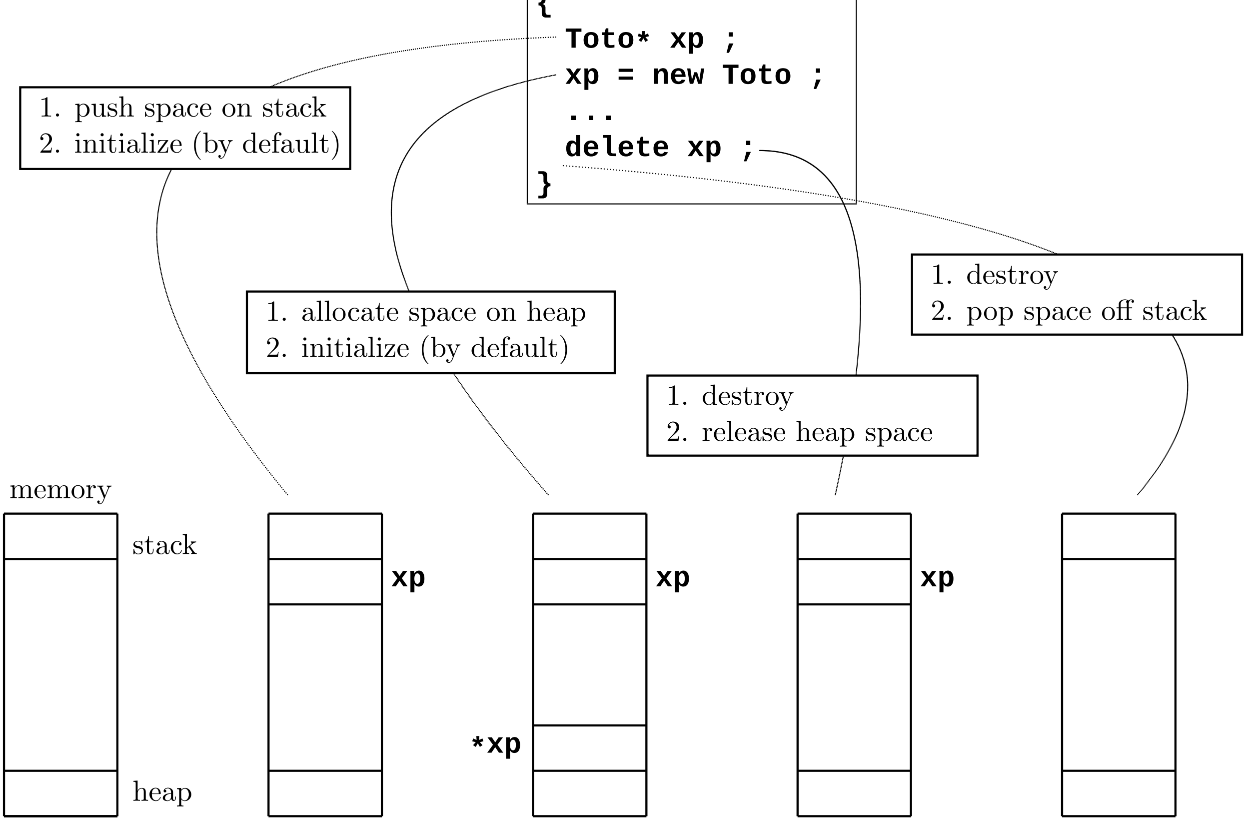
- Objects declared directly are created on the stack.

  Examples: formal parameters, return values, objects defined inside blocks.
- Objects are destroyed in reverse order of creation.
- Objects destruction is done automatically on exit of the block where they were created.



I.8.4 The Heap

- Objects created with **`new`** are placed on the heap.
- They will persist until explicitly destroyed.
- To refer to such objects, you need a pointer.
- Every object created with **`new`** should have a correponding **`delete`**, otherwise you will get *memory leaks*.



I.8.5 Construction and Destruction

Different ways an object can be created and gets destroyed afterwards:

- **named automatic object**: created each time its declaration is encountered in the execution of the program and destroyed each time the program exits the block in which it occurs
- **dynamic object**: created using the new operator and destroyed using the delete operator
- **nonstatic member object**, member of another class object: created and destroyed when the object of which it is a member is created and destroyed
- **array element**: created and destroyed when the array of which it is an element is created and destroyed
- **local static object**: created the first time its declaration is encountered in the execution of the program and destroyed once at the termination of the program
- **global, namespace, or class static object**: created once at load time and destroyed once at the termination of the program
- **temporary object**: created as part of the evaluation of an expression and destroyed at the end of the full expression in which it occurs
- object placed in memory obtained from a user-supplied function guided by arguments supplied in the allocation operation
- union member, which may not have a constructor or a destructor

Chapter II

An Approach to Object-Oriented Programming

II.1 Objects and Classes: A Presentation

II.1.1 Abstract Data Types

Type

- A *type* is a description of a set of individuals.
- The associated *type extension* (or simply *extension*) is the set of individuals.
- An *instance* of a type is a single member of its extension.

Abstract Data Type

- An *abstract data type* (ADT) is a *type* together with a *set of operations* applicable to its instances.
- The *set of operations* provide the one and the only access mechanism to the instances characteristics.

II.1.2 Classes

Class

- A *class* is an abstract data type equipped with a possibly partial implementation.

A class is a *software text* that exists independently of any execution.

Member

- A *member* of a class is the implementation of an operation of the associated Abstract Data Type.
- Members are classified:
  - *data members* or *attributes*: members represented by associating *data* with every instance of the class;
  - *function members* or *methods*: members represented by defining a certain computation applicable to all instances of the class or by defining a mechanism for delivering instances of the class.

II.1.3 Objects as Data Structures

Object

- An object is a *run-time data structure* made of zero or more *values*.
- A *field* is one of the *values* that make up an object.
- The *value* of an object is the set of all its fields.
- The *identity* is a property that uniquely identifies an object independently of its *value*.

Object Rule

- Every object O is an instance of some class C.
- Each field making up O is described by an attribute of C.

In other words, a class defines a certain mold and an object is a *data structure* built according to that mold.

II.1.4 Objects as Machines

Classification of Function Members

Function members are categorized as follows:

- A *delivery method* is a function member used to obtain an instance of the class.
- A *query* is a function member, other than a *delivery method*, returning a result.
- A *command* is a function member that does not return a result.

The notion of *delivery* mechanism is more abstract than the notion of instance *creation*: although it may end up being implemented as calling an allocation operator and subsequently returning its result, it does not have to. For example, the classes having a limited number of instances implement the delivery mechanism by selecting and returning an appropriate instance. From a conceptual perspective we may pretend that all of the instances of interest, for all times past, present and future, are already inscribed in the Great Book of Abstract Data Type Instances, and a delivery mechanism is just a way to obtain one of them.

Objects as Machines

An object can be viewed as a machine with an *abstract state*.

- The *abstract state* is always accessed indirectly through *commands* and *queries*: it is a pure abstraction.
- *Queries* provide information about the *abstract state*.
- *Commands* change the *abstract state*.
- The *abstract state* of an object is its visible properties as expressed by its non-secret queries.

> **Command-Query Separation Principle**
> Every *query* does not modify the *abstract state* of the current object.

The Command-Query Separation principle is a methodological precept, not a language constraint. This does not, however, diminish its importance. Thus, it ought to be applied with no exception.

II.1.5 Refering to Objects

Pointer/Reference

- a pointer/reference is a *run-time value* which is either *void* or *attached*
- if *attached*, a pointer/reference identifies a single object (it is then said to be attached to that particular object).

Variable

A *variable* is a *name* in a *software text* that denotes a *run-time value* (object or pointer/reference).

When talking about *variables*, we are dealing with two or three levels:

- a variable is an identifier in the software text;
- the variable denotes a run-time value, which is either an object or a pointer/reference;
- if the variable denotes a pointer/reference, that pointer/reference may get attached to an object.

II.1.6 Accessing and Manipulating Objects

All computation is achieved by *calling* certain *function members* on certain *objects*

Function Member Call

A *function member call* is **formally** expressed as $x.f(arg)$ (dot notation) where:

- x is a *variable* denoting an object O of class C ;
- f, is a *function member* of C whose arguments is arg.

x is called the *target* of the call, O the *target object*.

some object-oriented jargon: "passing to the object denoted by x the message f with arguments $args$"

II.1.7 Object-Oriented Systems

System

An *O-O system* is a set of *classes* that can be assembled to produce an executable result.

Three Levels in an O-O system

1. a *system* is a set of *clusters*;
2. a *cluster* is a set of *classes*;
3. a *class* is a set of *function members* and *data members*.

Static/Run-Time Structure

- The *static structure* of an *O-O system* is the **software text** of all its *classes*.
- At any time during its **execution**, an *O-O system* will have created a certain number of objects. The *run-time structure* is the organization of these objects and their relations.

II.1.8 Modularity

Module

- A *module* is a basic unit of software decomposition.

Modularity is a syntactic concept: decomposition into modules only affects the form of software texts, not what the software can do.

Modular Structure

- In functional programming, modular structures are based on *subroutines*.
- In object oriented programming, *classes* provide the basic form of modules.

The choice of a proper module structure is the key to achieving the aim of *reusability* and *extendibility*.

Client-Supplier

- A *client* of a piece of code is somebody (a programmer) or something (a module) that uses that piece of code. This latter is called *supplier*.

Clients are important; clients are the name of the game. If nobody uses the software you write, why write it ? Good software is "clientcentric": it resolves around clients.

Concept of a Main Program

- In functional programming, the *main program* is:
  - the fundamental top component of the system's architecture, and
  - the place where execution begins.
- In object oriented programming, the *main program* is **nothing more** than the place where execution begins.

II.1.9 From Classes to Components

A *component* is a **client oriented software** that:

1. may be used by other software elements (its *clients*);
2. may be used by clients without intervention of the component's developers;
3. includes a specification of all dependenies (hardware and software platform, versions, other components);
4. includes a precise *specification* of the functionalities it offers;
5. is usable on the sole basis of that specification;
6. is composable with other components;
7. can be integrated into a system quickly and smoothly.

Components are prebuild pieces of software that help building an application. In component technology, a system is decomposed into runtime elements, that can be built, analyzed, tested and maintained independently. The integration of available *off-the-shelf components* into applications can help to reduce development time, increase developer productivity together with ensuring high-quality criteria.

II.1.10 Object Based Programming – Summary

- abstract data types
- classes
- objects
- refering to objects
- accessing and manipulating objects
- object oriented systems
- modularity
- components

II.2 Objects and Classes: An Application in **C++**

II.2.1 **C++** User-Defined Types

- **C++ class**es are closely related to the notion of *class* defined previously.
- **class**es may define function members as well as data members, collectively called members.
- Protection labels control access to members:
  - **public** members are accessible to all *clients*;
  - **protected** members are accessible only to members of the class and of its derivatives (more on this later);
  - **private** members are accessible only to members of the class.

  Protection labels can appear in any order and multiple times within a class.
- In addition to **class**es, user-defined type can be defined as **struct**s. The only difference is in the default protection that applies to members: **public** for **struct**s and **private** for **Class**es.
- The construct

  ```
  class X { ... } ;
  ```

  is called a *class definition* because it **defines** a new type.

II.2.2 Partitioning Class and Member Definitions

- We recommend that *data members* be **private** with no exception.
- *Function members* can be defined inside or outside the *class definition*.
  - If defined inside, calls to them will be expanded inline.
  - We recommend to restrict inlining to optimization purposes, and hence to define *function members* **outside** the *class definition*.
- Outside the *class definition*, the *name* of a *member* must be qualified to indicate that is is from the *class scope*.
- We recommend that the implementation of a **C++** class **X** be partitioned into two or three files:
  - a *header file*, say **X.hh**, containing the *class definition*;
  - a *source file*, say **X.cc**, containing the definition of non-**inline** *class members*;
  - a *source file*, say **X.icc**, containing the definition of **inline** *class members*;
- We recommend that:
  - The header file contains a mechanism that prevents multiple inclusion of that file.
  - The **public**, **protected** and **private** sections appear in that order (thus avoiding the defaults).
  - The header file uses fully *qualified names*.

<table>
<tr><th>file X.hh</th><th>file X.cc</th></tr>
<tr><td>

```
#ifndef X_HH
#define X_HH

class X
{
   public:
      ...
   protected:
      ...
   private:
      ...
      // attributes
      ...
} ;

#endif
```

</td><td>

```
#include <X.hh>



// definition of function member
// foo of class X
... X::foo( ... ) ...
{
   ...
}
```

</td></tr>
</table>

II.2.3 Function Members

- Within a *function member definition*, the *names* of *members* are implicitly bound to the *target object* on which this *function member* is *called*.
- The **`this`** keyword, valid only inside a *function member*, denotes a pointer to the *target object* on which this *function member* is *called*.
- A *function member* that is defined **`const`** (by inserting the **`const`** keyword after the parameter list) is not allowed to change the *state* of the *target object* on which this *function member* is *called*.

  ⇒ **`const`** function members are used to implement *queries*.

<table>
<tr><th>in file X.hh</th><th>in file X.cc</th></tr>
<tr><td>

```
class X
{
   public:
      std::string const&
         name( void ) const ;
   private:
      std::string MY_NAME ;
} ;
```

</td><td>

```

std::string const&
X::name( void ) const
{
   return MY_NAME ;
}
```

</td></tr>
</table>

II.2.4 Special Members

- *Constructors* are **special** *function members* that define how objects of the type are **initialized**. They have
  - the same *name* as the class;
  - no return value.

  A class can define multiple constructors.

- A *constructor initializer list* is a comma separated list of *member-name(value)* pairs.
  - Each *member-name* is initialized from the associated *value*.
  - *Data members* that are not initialized in the *initializer list* are implicitly initialized.
  - *Data members* are initialized is the order of declaration in the *class definition*.
- When we create a new class object, the following steps happen in sequence:
  1. The implementation allocates memory to hold the object.
  2. It initializes the object using initial values as specified in the *initializer list*.
  3. It executes the constructor body.
- The role of the *constructors* in a *class* is to ensure that an object of the type is correctly **initialized** on completion of **any** creation statement.
- The constructor that takes no argument is known as the *default constructor*.

```
X( void ) ;
```

- A class controls what happens when objects of the type are copied, explicitly or implicitly, through the *copy constructor*.

```
X( X const& other ) ;
```

- A *destructor* is a **special** *function member* that controls what happens when objects of the type are destroyed.

```
~X( void ) ;
```

  It has
  - the same *name* as the class, prefixed by a tilde (**~**)
  - no argument and no return value.
- A class controls what happens when objects of the type are assigned through the special member operator called the *assignment operator*.

```
X& operator=( X const& other ) ;
```

  It is essential that the assignment operator deals correctly with self-assignment.
- Special care must be taken when implementing copy, assigment and destruction facilities in classes that allocate ressources in a constructor.

II.2.5 Assignment is not Initialization

- Assignment always obliterates a previous value.
- Assignment happens **only** when using the **=** *operator* in an *expression*.
- Initialization
  - never involves a preexisting value;
  - involves creating a new object and giving it a value at the same time.
- Initialization happens
  - in variable *declarations*;
  - for *function parameters* on entry to a *function*;
  - for the *return value* of a *function* on return from the *function*;
  - in *constructor initializers*.

```
string nn( 10, 'x' ) ;      // initialization
string nn( "Hello" ) ;      // initialization
string nn = "Hello" ;       // initialization
nn = "Hello" ;              // assignment
```

Initialization and *assignment* cause different operations to run.

- *Constructors* **always** control *initialization*.
- The **`operator=`** function member **always** control *assignment*.

II.2.6 Member Access Operators

postfix-expression **.** *name*

- *postfix-expression*: represents a *value* of **`struct`**, **`class`**, or **`union`** type.
- *name*: names a member of the specified structure, union, or class.
- value of the operation: that of *name* and is an *l-value* if *postfix-expression* is an l-value.

postfix-expression **->** *name*

- *postfix-expression*: represents a *pointer* to a **`struct`**, **`class`**, or **`union`** type.
- *name*: names a member of the specified structure, union, or class.
- value of the operation: that of *name* and is an *l-value*.
- The expressions *e*`->`*member* and `(*`*e*`).`*member* yield identical results (except when the operators `->` or `*` are overloaded).

```
string n = "Niels Stroustrup";
string s = n.substr( 6, 10 );
assert( s == "Stroustrup" );
n.replace( 0, 5, "N." );
assert( n == "N. Stroustrup" );
```

```
string* n = new string("PELICANS");
string s = n->substr( 2, 6 );
assert( s == "LICANS" );
n->replace( 0, 2, "pe" );
assert( *n == "peLICANS" );
delete n;
```

II.2.7 Use of Class Member Names

The name of a *class member* has *class scope* and can only be used:

- in a *member function* of that class;
- in a *member function* of a class derived from that class;
- after the **.** (dot) operator applied to an instance of that class;
- after the **.** (dot) operator applied to an instance of a class derived from that class, as long as the derived class does not hide the name;
- after the **->** (dereference) operator applied to a pointer to an instance of that class;
- after the **->** (dereference) operator applied to a pointer to an instance of a class derived from that class, as long as the derived class does not hide the name;
- after the **::** (scope-resolution) operator applied to the name of that class;
- after the **::** (scope-resolution) operator applied to a class derived from that class.

II.2.8 Static Members

Static Members

- **static** members exist as members of the class, rather than as a feature in each object of the class.
- The names of **static** members are within the scope of their class (⇒ minimization of names that are globally defined).

Static Data Members

- There is a single instance of each **static** data member for the entire class.
- **static** data members must be initialized, usually in the source file that implements the class function members. Because they are initialized outside the class definition, you must fully qualify the name when you initialize it:

  ```
  double Toto::xx = 10.2 ;
  ```

  says that the **static** member named **xx** from the class **Toto** has type **double** and is given the initial value **10.2**

Static Function Members

- Unlike other function members, **static** function members are associated with the class, not with a particular object: they do not operate on an object of the class type.
- The **this** keyword is not available in a **static** function member.
- **static** function members may access only **static** data members.

Static Member Access

- A **static** *member* can be referred to without mentionning an object.
- Instead, its *name* is *qualified* by the name of the class.

```
PDE_DomainAndFields* dom = PDE_DomainAndFields::create( 0, ee ) ;
PEL_LocalEquation* leq = PDE_LocalEquation::create( 0 ) ;
PEL_Error::object()->raise_plain( "invalid data" ) ;
GE_Color const* cc = GE_Color::object( "bottom" ) ;
GE_QRprovider const* qrp = GE_QRprovider::object( "GE_QRprovider_3" ) ;
PDE_LocalFE::field_id const row = PDE_LocalFE::row ;
PDE_LocalFE::field_id const col = PDE_LocalFE::col ;
LA_PreconditionedSolver* s = LA_PreconditionedSolver::make( 0, ee ) ;
cFE->require_field_calculation( TT, PDE_LocalFE::dN ) ;
FE::add_row_vvgrad_col( ELEMENT_EQ, cFE, aa, 1.0 ) ;
```

Statics: Schizophrenia for C++ Programmers

The **static** keyword is a **C++** construct that takes on multiple meanings.

context	meaning
Applied to a *variable declaration* inside a *function*	**"Global Variable with Local Scope"** "permanent" variable: initialized only once and retains its value from one function call to the next
Applied to a function or variable defined outside the body of any function	**"Local Globals"** *file scope* with a use restricted to its *translation unit*: not accessible through **extern** declarations in other translation units, no conflict with global variables or with statics of other translation units, even with the same name
Applied to *data member* of a *class*	**"Class Data Member"** only one such data for the class, no matter the number of instances that are created
Applied to *function member* of a *class*	**"Class Function Member"** only one copy of the code, available to all objects of the class, can access only static members and do not have a **this** pointer

II.2.9 Function Members that C++ Silently Writes and Calls

If you write this:

```
class Empty{} ;
```

it's the same as if you'd written this:

```
class Empty
{
  public:
    Empty( void ) ;                             // default constructor
    Empty( Empty const& other ) ;               // copy constructor
   ~Empty( void ) ;                             // destructor

    Empty& operator=( Empty const& other ) ; // assigment operator

    Empty* operator&( void ) ;                  // address-of operator
    Empty const* operator&( void ) const ;
} ;
```

The following code will cause each function to be generated:

```
Empty const e1 ;          // default constructor, destructor
Empty e2( e1 ) ;          // copy constructor
e2 = e1 ;                 // assignment operator
Empty* pe2 = &e2 ;        // address-of operator (non-const)
Empty const* pe1 = &e1 ;  // address-of operator (const)
```

II.3 Inheritance: A Presentation

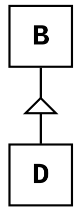
II.3.1 Inheritance

Heir-Parent

A class **D** is an *heir* of a class **B** is **D** incorporates **all members** of **B** **in addition** to its own.

We say that:

- **D** *inherits from* **B**;
- **B** is a *parent* of **D**.



UML notation

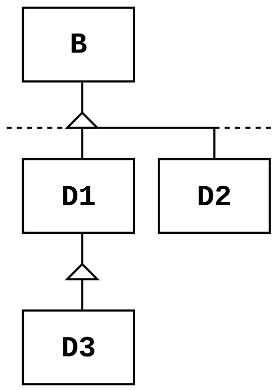
A Meaning of Inheritance: Specialization

If **D** inherits from **B** :

- any instance of **D** may be viewed as an instance of **B**, but not conversely ;
- every operation applicable to instances of **B** is also applicable to instances of **D**.

> Inheritance: **IS-A** relationship (subset inclusion, *ie* relation between two categories).

II.3.2 Inheritance Terminology



Terminology for Classes

- A *descendant* of a class **B** is either **B** itself or, recursively, a descendant of an heir of **B**.
- A *proper descendant* of a class **B** is a descendant other than **B**.
- An *immediate descendant* of a class **B** is a heir **D** of **B**. We say that **D** *inherits directly* from **B**.
- A descendant of a class **D** which is not a heir of **D** is said to *inherits indirectly* from **D**.
- An *ancestor* of a class **D** is a class **B** such that **D** is a descendant of **B**.
- An *proper ancestor* of a class **D** is a class **B** such that **D** is a proper descendant of **B**.

Terminology for Members

- An *inherited* member is a member coming from a proper ancestor.
- An *immediate* member is a member introduced in the class itself.

II.3.3 Abstracting Software Elements

Effective/Deferred Members

- An *effective* member is a member declared with an implementation.
- A *deferred* member is a member which has a specification but no implementation.

Effective/Deferred Class

- A class is *deferred* if it has a deferred member.
- A class is *effective* if it is not deferred.

The more deferred a class, the closer it is to an Abstract Data Type.

Adaptation of Inherited Members

Some properties (apart from the name) of a inherited member might be adapted:

- an implementation may be provided to a *deferred* member;
- some properties of a member inherited as *effective* may be changed;
- the specification of a member inherited as *deferred* may be changed, while leaving that member *deferred*.

II.3.4 Objects and Inheritance

Direct Instance, Generator

- Every object is the *direct instance* of just one class, called its *generator*.
- An instance of a class is a *direct instance* of one of its descendants.

Subobject, Complete Object

- Objects can contain other objects, called *subobjects*.
- A *subobject* may be:
  - a field making up another object ;
  - a parent class part of another object.
- An object that is not the subobject of any other object is called a *complete object*.

Deferred Class No-Instantiation Rule

- There is no such thing as a direct instance of a deferred class.
- The creation type of a creation instruction may not be deferred.

II.3.5 Polymorphism

Polymorphism means the ability to take several forms

- A *polymorphic variable* is any variable that may at run-time become attached to objects of more than one type.
- A *polymorphic attachment* is an assigment in which the type of the source is different from the type of the target.

Variables involved in polymorphic attachments always denote pointer/references

- A *polymorphic call* is a function member call whose target is polymorphic.

II.3.6 Static/Dynamic Type

Objects

- The *dynamic type* of an object is the type with which it has been created.
- The *dynamic type* of an object **will never change** during the object's lifetime.

Pointers/References

- The *dynamic type* of a pointer/reference is the *dynamic type* of the object to which it is currently attached.
- The *dynamic type* of a pointer/reference **may change** as a result of reattachement operations.

Variables

- The *static type* of a variable is the type with which it was declared.
- The *dynamic type* of a variable is the *dynamic type* of the denoted *run-time value* (object or pointer/reference).

Remarks

- Objects and pointers/references only have a *dynamic type* ($\Rightarrow$ *dynamic* omitted).
- *variables* have both a *static type* and a *dynamic type*.

II.3.7 Typing

Type Violation

A *type violation* occurs in the execution of a call $x.f(arg)$, where x is attached to an object O, if either:

- there is no *function member* corresponding to f and applicable to O;
- there is such a *function member*, but arg is not an acceptable argument for it.

Statically Typed Language

An object-oriented language (*eg* **C++**) is statically typed if

- it is equipped with a set of **consistency rules**,
- that are enforceable by compilers **prior to execution**,
- whose observance by the software text guarantees that no execution of the system can cause a type violation.

$\Rightarrow$ reliability, readability, efficiency

II.3.8 Consistency Rules for Statically Typed Languages

Declaration

- Every *variable* must be declared as being of a certain type.
- Every *function* must be declared as being of a certain type and must declare zero or more formal arguments, with a type for each.

Function Member Call Rule

In a *function member call* $x.f(arg)$, where x is an *variable* denoting an object O of class C, the *function member* f must be defined in one of the ancestors of C and must be available to the class in which the call appears.

Limits to Polymorphism

A *variable* declared of a type T may at run-time **only** become attached

- to direct instances of T, **or**
- to direct instances of descendants of T (*ie* to instances of T).

II.3.9 Binding

Binding Problem

Consider the *function member call* $x.f(arg)$

- Since a class can change an inherited function member, there may be two or more operations to execute.
- Binding question: which operation will the call execute ?

Static/Dynamic Binding

Consider the *function member call* $x.f(arg)$

- *dynamic binding* means that **every execution** will select the version of f based on the *dynamic type* of *variable* x (*i.e.* the type of target object denoted by x at run-time).
- *static binding* means that the selection of a version of f to be executed is performed **before execution** on the basis of the *static type* of *variable* x (*i.e.* as declared in the software text).

`C++` Approach to Binding

The programmer is responsible for selecting static or dynamic binding:

- by default, binding is static ;
- to be dynamically bound, a *function member* must be declared **`virtual`**.

II.3.10 Application Framework

Framework

A *framework* is an *object-oriented system* made up of a set of related classes that can be specialized, or *instantiated*, to implement an application.

Plug-Point

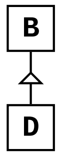
A *plug-point* is a class in a *framework* for which some *function members* must be *overridden* by the framework *client* to build a working application.

II.4 Inheritance: An Application in **C++**

II.4.1 **C++** Heir-Parent Relationship

- In **C++**, **public** inheritance defines a relationship between two classes closely related to the *heir-parent* relationship defined previouly.
- *Definition* of a class **D** that *is derived from* or *inherits from* a class **B**:

```
class B { ... } ;

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



- Because **D** *inherits from* **B**:
  - every member of **B** is also a member of **D**,
  - **except** for the *contructors*, the *assigment operator*, and the *destructor*.
- Derivation chains can be several layers deep:

```
class B { ... } ;
class D : public B { ... } ;
class E : public D { ... } ;
```

- Multiple inheritance is possible, but in the sequel, we will ignore this feature.

II.4.2 Protection Revisited

- **public** members are accessible to any *clients*.
- **private** members of a class are accessible only from within that class.
- The **protected** label gives derived classes access to the **protected** members of their constituent base class objects, but keeps these elements inaccessible to users of the class.
- We recommend that *data members* be **private** with no exception.

```
class A
{
   public:
      // common interface
   protected:
      // implementation members accessible to
      // derived classes
   private:
      // implementation accessible to only
      // the A class
} ;

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

II.4.3 Object of a Class Contruction and Destruction Model

Construction

The **C++** object of a class construction model is:

1. memory allocation

2. memory initialization
   2.1 base-class sub-object initialization (recurcive process)
   2.2 member sub-objects initialization (recurcive process)
   2.3 execution of the *constructor*

Destruction

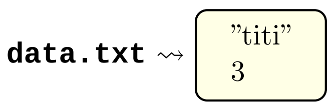
The **C++** object of a class destruction model is:

1. memory deinitialization
   1.1 execution of the *destructor* body
   1.2 member sub-objects deinitialization (recurcive process)
   1.3 base-class sub-object deinitialization (recurcive process)
2. memory deallocation

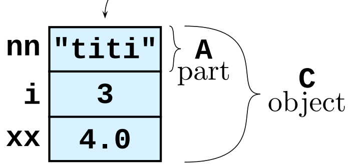
II.4.4 Inheritance and Constructors

- Objects of a derived type are constructed by:
  - allocating space for the **entire** object (base-class members as well as derived members);
  - calling the base-class constructor to initialize the base-class part(s) of the object;
  - Initializing the members of the derived-class as directed by the *constructor initializer*;
  - Executing the body of the derived-class constructor, if any.
- The *constructor initializer* of the derived-class is used to specify the base-class *constructor* that is desired.
- If the *constructor initializer* of the derived-class does not specify which base-class constructor to run, then the base-class *default* constructor is used to build the base-part of the object.

```
class A {
   [public/protected]:
      A( std::istream& is ) ;
   private:
      std::string nn ;
} ;

class C : public A {
   public:
      C( std::istream& is, double x ) ;
   private:
      double xx ;
      int i ;
} ;

A::A( std::istream& is )
: nn()
{ is » nn ; }

C::C( std::istream& is, double x )
: A( is ), xx( x*x ), i( 0 )
{ is » i ; }
```




```
{
ifstream f( "data.txt" );
C cc( f, 2. );
   ...
}
```



II.4.5 Polymorphism

A *reference* or *pointer* to a base-class object may refer or point:

- to a base-class object;
- to an object of a type derived from the base class.

```
class A { ... } ;
void fa( A const& a ) ;

class C : public A { ... } ;
void fc( C const& c ) ;

{
   C cc( ... ) ;
   A& a = cc ;  // ok
   fa( cc ) ;   // ok

   A aa( ... ) ;
   C& c = aa ;  // illegal
   fc( aa ) ;   // illegal
}
```

```
class A { ... } ;
void fa( A const* a ) ;

class C : public A { ... } ;
void fc( C const* c ) ;

{
   C* cc = new C( ... ) ;
   A* a = cc ;  // ok
   fa( cc ) ;   // ok

   A* aa = new A( ... ) ;
   C* c = aa ;   // illegal
   fc( aa ) ;    // illegal
}
```

II.4.6 Static/Dynamic Binding

- The call of a `virtual` function though a *pointer* or *reference* is **dynamically bound** (except in *constructors* and *destructors*).
- **Any** other call is **statically bound**.

```
class A {
   public:
      virtual void foo( void ) ;
} ;
class C : public A { ... } ;

{
   A aa( ... ) ;  C cc( ... ) ;

   aa.foo() ;    // statically bound to A::foo()
   cc.foo() ;    // statically bound to C::foo()

   A& ar = cc ;
   ar.foo() ;   // dynamically bound. C::foo() selected.

   A* ap = &aa ;
   ap->foo() ;  // dynamically bound. A::foo() selected.
   ap = &cc ;
   ap->foo() ;  // dynamically bound. C::foo() selected.
}
```

II.4.7 Deferred Function Members and Classes in **C++**

- A **virtual** function is specified *pure* by using the *pure-specifier* **= 0** in the *function declaration* int the *class definition*.
- A *pure* **virtual** function need be defined only if explicitly called with the *qualified-id* syntax.
- In **C++**, *deferred function members* are represented by *pure* **virtual** functions.
- A *class* is termed *abstract* if it has at least one *pure* **virtual** function.
- An *abstract class* can only be used as a base-class of some other class; no object of an abstract class can be created except as sub-object of a class derived from it.
- In **C++**, *deferred classes* are represented by *abstract classes*.

```
class B
{
   public:
      virtual ... foo( ... ) [const] = 0 ;
} ;
```

II.4.8 Adaptation of Deferred Function Members in **C++**

- A derived-class member *overrides* a **virtual** function with the same name in the base-class if:
  - the two functions have the same number and types of arguments.
  - both or neither are const.

  In that case:
  - the two functions must have the same return type
  - except that, if the base-class function returns a pointer (or reference) to a class, the derived-class function can return a pointer (or reference) to a derived class.
- It is **legal** to *override* a function member that is not **virtual**, but **it is not moral**.

```
class B
{
   virtual B* create_clone( void ) const = 0 ;
   virtual void foo( void ) ;
} ;

class D : public B
{
   virtual D* create_clone( void ) const ;
   virtual void foo( void ) ;
}
```

II.4.9 Virtual Destructors

- If a pointer to a base-class is used to **`delete`** an object that might actually be a derived-class object, then the base-class needs a **`virtual`** destructor.

```
class B {
   ~B( void ) ;
}

class D : public B {
   ~D( void ) ;
}

B* b = new D( ... ) ;
...
delete b ;
// ~B() called
```

```
class B {
   virtual ~B( void ) ;
}

class D : public B {
   virtual ~D( void ) ;
}

B* b = new D( ... ) ;
...
delete b ;
// ~D() called
```

- If the base-class has no other need for a destructor, then the **`virtual`** destructor still must be defined and should be empty.

```
B::~B( void ) {}
```

- The **`virtual`** nature of the *destructor* is inherited by the derived classes.

II.4.10 For A Sensible Use of C++ public Inheritance

Guidelines

- Never *override* an inherited non-**`virtual`** function.
- Never redefine an inherited default parameter value.
- Avoid casts down the inheritance hierarchy.
- Make base class destructors virtual.
- Make non leaf classes abstract.

Uses and Abuses of Inheritance

- If a class relationship can be expressed in more than one way, you should use the **weakest relationship** that's practical.
- Inheritance is nearly the **strongest relationship** you can express in **C++**.
- Always minimize coupling.
- Inheritance is only appropriate when there is no weaker alternative.

> Inheritance is often overused, even by experienced programmers.
> [Herb Sutter]

II.4.11 Say What You Mean; Understand What You're Saying

Understanding what different object-oriented constructs in **C++ mean**
is different from
just knowing the rules of the language.

- **public** inheritance means **IS-A**, *ie* if **D** **public**ly inherit from **B**:
  - every object of type **D** is also an object of type **B**;
  - every operation applicable to objects of type **B** is also applicable to object of type **D**.
- A *pure* **virtual** function means that **only** the function's **interface** is inherited.
- A non-*pure* **virtual** function means that the function's **interface plus a default implementation** is inherited.
- A nonvirtual functions means that the functions **interface plus a mandatory implementation** is inherited.

Some **formalization** would help → Design By Contract

Chapter III

Essential Object-Oriented Development with PELICANS

III.1 Modularization

III.1.1 Modules are Classes

- Every module is a *class*.
- Subprograms do not exist as independent modular units.
- There is no notion of main program: the *main function* is nothing more than the place where execution begins.

III.1.2 Libraries

- Classes are grouped into administrative units (clusters) called libraries.
- The *name* of every class in a library begins with a prefix that is unique for the library.

  (exception: some particular classes of **PELbase** such as **doubleArray2D** which are intended to be considered as C++ fundamental types)

library	**PELbase**	**LinearAlgebra**	**Geometry**	**PDEsolver**	**FrameFE**
prefix	**PEL**	**LA**	**GE**	**PDE**	**FE**

Example

- **PEL_Vector** belongs to the **PELbase** library.
- **GE_Vector** belongs to the **Geometry** library.
- **LA_Vector** belongs to the **LinearAlgebra** library.

III.2 Fundamental Restrictions on **C++** Usage

Some **arbitrary** rules have been followed when developping PELICANS. As much as possible, they should be followed by client authors.

III.2.1 Inheritance

- Never use **`private`** or **`protected`** inheritance.
- Never use multiple inheritance.

III.2.2 Genericity

Do not develop template functions or classes.

III.2.3 Standard Template Library

- The use of the STL is restricted to the implementation tasks.
- None of the STL components should ever be quoted in non secret interfaces.

III.2.4 Controversy

Discussion between a possible client author and a PELICANS developper:

> **Client Author** : But all these C++ functionnalities are tremedous, useful, essential...
> **PELICANS team** : Yes. That's for sure.
> **Client Author** : So why not use them ?
> **PELICANS team** : Because we said so.

End of the discussion.

III.3 Restrictions Imposed to PELICANS Objects

III.3.1 PELICANS Objects

A PELICANS object is an object of a PELICANS class (whose name begins with the prefix of its library).

III.3.2 Storage Duration

- PELICANS objects are compulsorily *dynamic object* (placed on the *heap*).
- PELICANS objects cannot be *static object* nor *automatic objects*.
- The following *compound types* are forbidden for PELICANS objects:
  - built-in arrays of PELICANS objects
  - references to PELICANS objects;
  - classes with PELICANS objects as *data members*.

⇒ PELICANS objects are compulsorily:

- created by **`new`**-expression;
- *referred to*, *accessed* and *manipulated* exclusively **through pointers**.

III.3.3 Lifetime

Ownership

- An owner is assigned to any PELICANS object when it is created.

  The owner can be:
  - the NULL object (represented by the 0 pointer), or
  - another PELICANS object.
- Ownership cannot be transfered: the owner of a PELICANS object cannot be changed (except if it is the NULL object).

⇒ Each PELICANS object

- has one owner determined when it is created;
- owns an evolving collection of other PELICANS objects (its *possessions*).

Semi-Automatic Lifetime Management

- Any PELICANS object whose owner is the NULL object must be destroyed by calling a specially designed *function member*.
- The destruction of any PELICANS object whose owner is not the NULL object is managed by the owner itself and will occur before the termination of that owner.

III.4 Global Inheritance Structure

The principles of the PELICANS object and memory model discussed above have motivated a framework design based on a universal class, namely **`PEL_Object`** whose main features will be introduced now.

III.4.1 Universal Class Rule

- PELICANS classes are organized in a cosmic hierarchy where every type is derived from the cosmic class **`PEL_Object`**.
- Every PELICANS class that does not directly inherit from another PELICANS class publicly inherits from the **`PEL_Object`** class.

III.4.2 `PEL_Object` Class

The **`PEL_Object`** class:

- provides interfaces for operations of universal interest like duplication, comparison, basic output and persistence
- enforces the restrictions on the *storage duration* of PELICANS objects
- implements the semi-automatic lifetime management
- leads to the definition of the universal type **`PEL_Object*`** for referencing any PELICANS object

This latter feature is the implementation foundation of a set of collection classes provided by PELICANS. The objective of these classes is to store and retrieve PELICANS objects and their design is *reference-based*: they maintain pointers to objects of type **`PEL_Object`**. Polymorphism makes it possible to store pointers to PELICANS objects and to retrieve these pointers using a **safe** type cast down the inheritance hierarchy of the pointers retrieved by the PELICANS collection classes.

III.4.3 Remark

In the nineties, the `C++` community decided that the use of cosmic hierarchies was not an effective design approach in `C++`. At this time, it had been observed that such architectures, in which every object type is derived from a root class, result from an attempt to promote as much flexibility as possible or from a renouncement to understand and properly abstract the problem domain. This is a misapprehension of the goal of an architecture: an architecture should be as close to the problem domain as possible while retaining sufficient flexibility to permit reasonable future extensions.

- The role of **`PEL_Object`** is not that exposed by detractors of **`C++`** cosmic hierarchies.
- The mechanism to determine the type of an object is **never** used. Any time an object is manipulated through a pointer to **`PEL_Object`**, we know the expected type of the object and we are not asking for it.
- Inheritance is always applied in conformance with the Liskov Substitution Principle, formalized in terms of Design by Contract.

III.5 Special Functions

III.5.1 Assignment of PELICANS Objects

PELICANS objects are accessed and referred to only through pointers, so operator use becomes unattractive. To avoid any tricky difficulty with the assignment operator, which might be silently written and called by the compiler, it is declared **`private`** and never implemented.

If the assignment operation is semantically meaningful, a specific method, usually called **`set`** or **`copy`**, is implemented.

III.5.2 Constructors and Destructor of PELICANS Objects

Requiring Heap-Based Objects

PELICANS objects are compulsorily dynamic objects. Thus they are exclusively created using **`new`**-expressions. Their introduction by declarations or their implicit creation by the implementation is precluded. In order to prevent the clients and the implementation from creating PELICANS objects other than by calling **`new`**, a specific strategy has been adopted.

The constructors and the destructor should have an access as limited as possible.

- The constructors and the destructor are never **`public`**.
- For concrete leaf classes within the inheritance hierarchy, the constructors and the destructor are **`private`**.
- For non leaf classes within the inheritance hierarchy, the destructor is **`virtual`** with a **`protected`** access and the constructors have a **`protected`** access if they are to be called in derived classes and a **`private`** access otherwise.

Compiler Generated Function Members

Problem:

- The *default constructor*, the *copy constructor* and the *assignment operator* are **never relevant** for PELICANS classes.
- They may be automatically generated by the compiler and made **`public`**.

Solution:

- **declare** the *default constructor*, the *copy constructor* and the *assignment operator* in the **`private`** section ;
- don't **define** them.

Constructors and Destructor in `PEL_Object`

- **`PEL_Object`** objects are organized in a composite structure where every object
  - is owned by another **`PEL_Object`** object (the *owner*)
  - owns a set of **`PEL_Object`** objects (the *possessions*).
- The only constructor defined in **`PEL_Object`** is **`protected`** with a single argument refering to the owner of the created object (thus PELICANS objects cannot be created without specifying a owner).
- The destructor is **`protected`** and can be **`public`**ly accessed *via* the function member **`destroy`** (that commits suicide!).

  Calling **`destroy()`** terminates the **complete** referenced PELICANS object.

```
Class PEL_Object
{
   public:
      void destroy( void ) ;               // pseudo-destructor
   protected:
      virtual ~PEL_Object( void ) = 0 ;    // virtual destructor
      PEL_Object( PEL_Object* a_owner ) ; // only constructor
} ;

void PEL_Object:: destroy( void ) const
{
   PEL_LABEL( "PEL_Object:: destroy" ) ;
   PEL_CHECK_PRE( owner()==0 ) ;
   delete this ;                           // suicide
}
```

- The virtual destructor **`~PEL_Object(void)`** is implemented so that:
  - all possessions of the calling object terminate;
  - the calling object is removed from the possession list of its owner;
  - the calling object terminates.

III.5.3 Factory Methods

- The direct use of **new**-expressions by clients is impossible (constructors have a restricted access).
- The **new**-expressions are **encapsulated** in *factory methods*, with the following caracteristics:
  - **static**
  - name: usually **create** or **make**
  - first argument: pointer to the owner of the object to be created
  - result: a pointer to the created object

Example: Factory Method of PDE_DomainAndFields

```
class PDE_DomainAndFields : public PEL_Object
{
  public:
    static PDE_DomainAndFields* create( PEL_Object* a_owner,
                                        PEL_ModuleExplorer const* exp ) ;
} ;

PDE_DomainAndFields* dom = PDE_DomainAndFields::create( 0, ee ) ;
cout « dom->nb_space_dimensions() « endl ;
dom->destroy() ;
```

The use of such factory methods provide a unified technique of instantiation.

Implementation Example of a Factory Method

```
class Toto : public PEL_Object
{
  public:
    static Toto* create( PEL_Object* a_owner,
                         PEL_ModuleExplorer const* exp ) ;
  private:
    Toto( PEL_Object* a_owner, PEL_ModuleExplorer cont* exp ) ;
} ;

Toto* Toto::create( PEL_Object* a_owner,
                    PEL_ModuleExplorer const* exp )
{
  PEL_LABEL( "Toto:: create" ) ;
  PEL_CHECK_PRE( exp != 0 ) ;

  Toto* result = new Toto( a_owner, exp ) ;

  PEL_CHECK_POST( result != 0 ) ;
  PEL_CHECK_POST( result->owner() == a_owner ) ;
  return( result ) ;
}
```

III.5.4 Duplication

Cloning Method

The cloning operation belongs to creational features.

- A **virtual** member function called **create_clone** is declared in **PEL_Object**.
- Its aim is to create a PELICANS object with the **same value** as the calling object (semantically equivalent to the copy constructor), but with an owner specified in the passing argument.
- Its default implementation does nothing: it has to be overriden in derived classes, if meaningful.

```
Class PEL_Object
{
   public:
      virtual PEL_Object* create_clone( PEL_Object* a_owner ) const ;
} ;

Class Toto : public PEL_Object
{
   public:
      virtual Toto* create_clone( PEL_Object* a_owner ) const ;
} ;
```

Note: the declaration of the **create_clone** member function takes advantage of the fact that if a virtual function's return type is a pointer to a base-class, the derived-class's function may return a pointer to a class derived from that base-class.

Implementation Example of a Cloning Method

```
class Toto : public PEL_Object
{
  public:
    virtual Toto* create_clone( PEL_Object* a_owner ) const ;

  private:
    Toto( PEL_Object* a_owner, Toto const* other ) ;
} ;

Toto* Toto:: create_clone( PEL_Object* a_owner ) const ;
{
  PEL_LABEL( "Toto:: create_clone" ) ;

  Toto* result = new Toto( a_owner, this ) ;

  PEL_CHECK_POST( create_clone_POST( this, a_owner ) ) ;
  return( result ) ;
}
```

III.5.5 Possible Canonical Forms

Leaf Classes:

```
class X : public PEL_Object
{
  public:
    // factory method for creation from a given list of parameters
    static X* create( PEL_Object* a_owner, [...] ) ;

    // duplication operation
    virtual X* create_clone( PEL_Object* a_owner ) const ;

  protected: // empty section

  private:
    ~X( void ) ;

     X( void ) ;                      // declared but not implemented
     X( X const& other ) ;            // declared but not implemented
     X& operator=( X const& other ) ; // declared but not implemented

     X( PEL_Object* a_owner, [...] ) ;          // called by create
     X( PEL_Object* a_owner, X const* other ) ; // called by create_clone
} ;
```

Non Leaf Classes

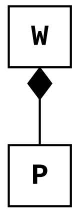
```
class X : public PEL_Object
{
  public:

  protected:
    virtual ~X( void ) ;

    // called by the factory method of derived classes
    X( PEL_Object* a_owner, [flist] ) ;

    // for duplication
    X( PEL_Object* a_owner, X const* other ) ;

 private:
    X( void ) ;                      // declared but not implemented
    X( X const& other ) ;            // declared but not implemented
    X& operator=( X const& other ) ; // declared but not implemented
} ;
```

III.5.6 Implementation of a Containment Relationship

- A *containment relationship* is a whole/part relationship between a class **W** (the whole) and a class **P** (the part) where every object of type **W** contains an object of type **P**.
- usual **C++** implementation: *data member* of type **P** in class **W**.
- if **W** and **P** inherit from **PEL_Object** : make **W** own an object of type **P** and manipulate it through a *pointer* declared as a *data member*

```
class W {
public:
   W( void ) ;
  ~W( void ) ;
private:
   P PART ;
} ;
class P {
public:
   P( void ) ;
} ;

W:: W( void )
   : PART() {}

W::~W( void ) {}
```

```
class W : public PEL_Object {
private:
  ~W( void ) ;
   W( PEL_Object* a_owner ) ;
   P* PART ;
} ;

class P : public PEL_Object {
public:
   static P* create( PEL_Object* a_owner ) ;
} ;

W:: W( PEL_Object* a_owner )
   : PEL_Object( a_owner )
   , PART( P::create( this ) ) {}
W::~W( void ) {}
```



III.5.7 Local Variables of Types Derived from PEL_Object

- No *named automatic object* of a class derived from **PEL_Object** can be created.
- Within a block
  - create a dynamic object with the NULL object as owner;
  - destroy this object before the end of the block.

```
class A : public PEL_Object {
public :
   static A* create( PEL_Object* a_owner ) ;
}

void X::f1( void ) {
   A* a = A::create( 0 ) ; // creation of an unnamed object
   /* ... */
   a->destroy() ;  // destruction of the previouly created object
}

void X::f2( void ) {
   A* a = A::create( this ) ;  // CAUTION : each call creates a new
   /* ... */                   // unnamed object which will last
}                              // until the end of the program
```

- Object created with **this** as owner should be reserved to *containment relationships*.

III.6 PELICANS-based Applications

III.6.1 Framework and Components

Classification in Software Engineering

PELICANS is both:

- an *application framework* ;
- a set of libraries of *components*.

In the PELICANS class index, *plug-points* appear in different fonts or colors. For example, within the **`PELbase`** library, **`PEL_Application`** is a plug-point whereas **`PEL_Root`** is not.

Static Structure of PELICANS-based Applications

In most cases, the software text of a PELICANS-based application

- consists of a set of classes
  - derived from PELICANS *plug-points*,
  - whose implementation uses *off-the-shelf components* provided by PELICANS;
- does not contain the **`main`** function (already defined in **`PELbase`**).

Remark

It is still possible to redefine the **`main`** function, see: **`$PELICANSHOME/SingleApplication`**

Execution

- Execution usually requires a data file.
- That data file stores a *hierarchical data structure* defined **by the application**.
- That data structure quotes a class derived from the *plug-point* **`PEL_Application`**.

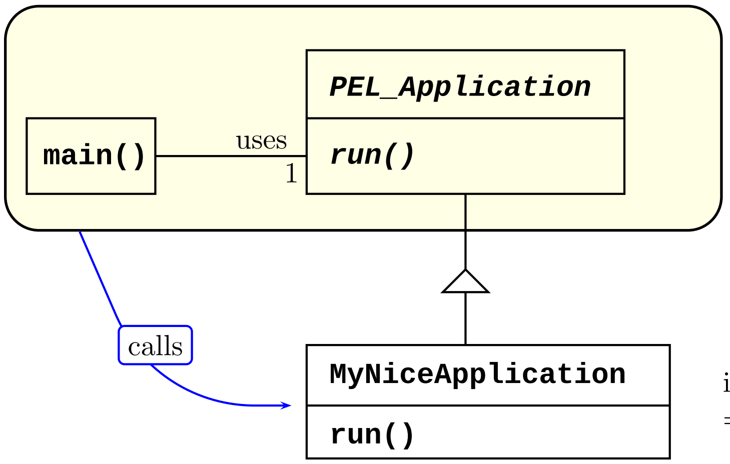
Organization of the execution by the **`main`** function of **`PELbase`**:

1. Initial stage: big-bang.
   ⇒ **irrelevant** to the developper of the current application
2. Reading of the data structure stored in the data deck.
3. Creation of an object whose type is the class derived from **`PEL_Application`** identified in the data deck.
4. Call the *function member* **`run`** on behalf of that object.
   ⇒ program execution proceeds by performing its **specific tasks**
5. Finalisation.
   ⇒ **irrelevant** to the developper of the current application

All that matters to the current developper: write the software text defining the **specific tasks** of its particular application

III.6.2 Pluggable Factories

Necessity



compiled PELICANS library
⇒ cannot be modified

implemented by the current developper
⇒ did not existed when
PELICANS was compiled

Need of a mechanism for adding
- **new** subclasses
- to existing **compiled** class hierarchies

➡ Pluggable Factories

Implementation

Consider an **abstract ancestor** in PELICANS (the *plug-point*) and **concrete subclasses**, (eventually implemented by the user, after the compilation of PELICANS).

Registration of Prototypes by the Abstract Ancestor

- A **`PEL_ObjectRegister`** object, called *register*, is built into the abstract ancestor as a **`static`** data member. It contains {key;data} pairs:
  - key: registration name of a concrete subclass
  - data: polymorphic variable whose *static type* is a pointer to the abstract ancestor and whose *dynamic type* is a pointer to the concrete subclass
- The abstract ancestor defines a **`protected`** constructor called *registration constructor* with one argument of type **`string`**. Executing it adds an entry to the register whose
  - key is the value of the **`string`** argument;
  - data is the pointer **`this`**.
- The concrete subclass defines a **`static`** data member, called *prototype*, whose initialization executes the *registration constructor* with the desired registration name as parameter.

⇒ **Before** the program begins to execute, the static objects are created leading to the registration of each concrete subclass *prototype* in the abstract ancestor's *register*

Selection of a Creational Method by Dynamic Binding

- The abstract ancestor declares the interface of a **`protected`** pure **`virtual`** function member **`create_replica`**, which is implemented in the concrete subclasses.
- The aim of **`create_replica`** is to create an object on basis of its arguments.
- The abstract ancestor defines a **`static`** function member **`make`** whose arguments are devoted to:
  - get the *registration name* of the concrete subclass to instantiate;
  - provide parameters for a call to **`create_replica`**.
- Internally, **`make`** performs a *polymorphic call*

  ```
  pt->create_replica( ... ) ;
  ```

  where **`pt`** is the data associated to the *registration name* deduced from the calling parameters of **`make`**.

  Hence, **`pt`** denotes the *prototype* of the desired concrete subclass. Due to *dynamic binding*, the associated version of **`create_replica`** is executed.

III.6.3 `PEL_Application` Header

```
class PEL_Application : public PEL_Object
{
   public:
      static PEL_Application* make( PEL_Object* a_owner,
                                    PEL_ModuleExplorer const* exp ) ;
      virtual void run( void ) = 0 ;

   protected:
      virtual ~PEL_Application( void ) ;
      PEL_Application( std::string const& name ) ;  // registration

      PEL_Application( PEL_Object* a_owner,
                       PEL_ModuleExplorer const* exp ) ;
      virtual PEL_Application* create_replica(
                            PEL_Object* a_owner,
                            PEL_ModuleExplorer const* exp ) const = 0 ;

      bool create_replica_PRE( PEL_Object* a_owner,
                               PEL_ModuleExplorer const* exp ) const ;
      bool create_replica_POST( PEL_Application const* result,
                                PEL_Object* a_owner,
                                PEL_ModuleExplorer const* exp ) const ;
};
```

III.6.4 **PEL_Application** Descendant

Header (sketch)

```
class X : public PEL_Application
{
   public:
      virtual void run( void ) ;

   private:
     ~X( void ) ;
      X( X const& other ) ;
      X& operator=( X const& other ) ;

      X( PEL_Object* a_owner, PEL_ModuleExplorer const* exp ) ;

      X( void ) ;

      virtual X* create_replica( PEL_Object* a_owner,
                                 PEL_ModuleExplorer const* exp ) const ;

      static X const* PROTOTYPE ;
} ;
```

Implementation (sketch)

data file

```
MODULE PEL_Application
   concrete_name = "MonJoliX"
END MODULE PEL_Application
```

```
X const* X::PROTOTYPE = new X() ;

X::X( void ) : PEL_Application( "MonJoliX" ) {}

X* X::create_replica( PEL_Object* a_owner,
                      PEL_ModuleExplorer const* exp ) const
{
   PEL_LABEL( "X:: create_replica" ) ;
   PEL_CHECK( create_replica_PRE( a_owner, exp ) ) ;
   X* result = new X( a_owner, exp ) ;
   PEL_CHECK( create_replica_POST( result, a_owner, exp ) ) ;
   return( result ) ;
}

X:: X( PEL_Object* a_owner, PEL_ModuleExplorer const* exp )
   : PEL_Application( a_owner, exp ) {}

X::~X( void ) {}

void X::run( void ) { cout « "Hello World" « endl ; }
```

III.7 Design By Contract

III.7.1 Assertions

Built-in Test

Built-in test refers to code added to an application that **checks** the application at *run-time*.

Assertions in PELICANS

The *assertion* is the workhorse of built-in test for object-oriented code.

- An *assertion* is a boolean expression that defines necessary conditions for correct execution.
- Assertions that support built-in test must be *executable*. In PELICANS, an *assertion* has three parts:
  1. a predicate expression (*i.e.* an expression containing conditions that evaluates to **`true`** or **`false`**);
  2. the action of directing a diagnostic message to the **`ostream`** class object **`cerr`** and subsequently terminating the program;
  3. an enable/disable mechanism.

Assertion Usage

The usage cycle of assertions includes several steps.

1. Assertions in PELICANS clients are coded by the client author.

2. Assertions in PELICANS clients are *enabled* or *disabled* at translation time and/or by using command-line switches (depending on the nature of the assertion, see below).

3. Assertions in PELICANS classes are *enabled* or *disabled* by linking with the appropriate PELICANS library.

4. At run-time, when the program reaches an enabled assertion, the *assertion predicate* either evaluates to **`true`** or **`false`**.

- if **`true`**, execution continues;
- if **`false`**, an *assertion violation* occurs. The result is a transfer of control to the associated *assertion action*: a diagnostic message is generated and the execution terminates.

Assertion Violation

A run-time *assertion violation* is the manifestation of a *bug* in the software.

III.7.2 Software Contracting

Client-Supplier

A *client* of a piece of code is somebody (a programmer) or something (a module) that uses that piece of code. This latter is called *supplier*.

The Contract Metaphor

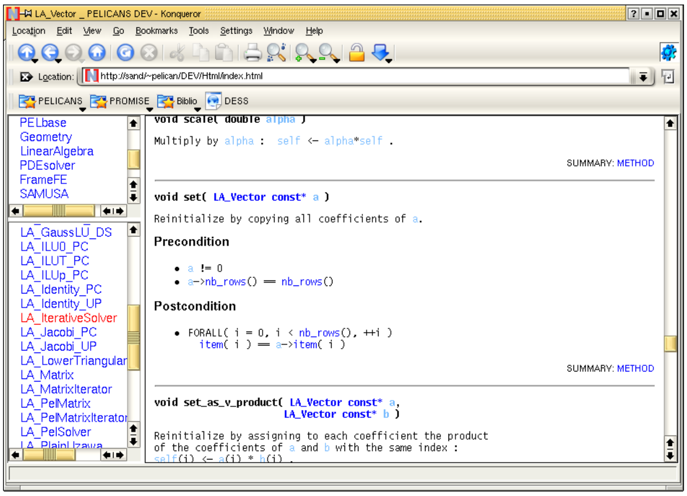
- A software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations – contracts.
- A *class contract* is an **explicit statement** of **rights** and **obligations** between a *client* and a *supplier*. The contract **states what** both parties must do, **independently of how** it is accomplished.

Contract Statement

A class contract:

- governs the interaction of the *supplier* with the rest of the world,
- is enforced using three kinds of assertions: *preconditions*, *postconditions* and *invariants*.

Example : Function Member `set` of Class `LA_Vector`



III.7.3 Preconditions

- A *precondition* is an assertion evaluated **at entry** to a *function member* **before** any of the code in the function body executes.
- A *precondition* expresses *constraints* on:
  - the value of the arguments, and
  - the object state

  required for correct execution of the *function member*.
- Because the clients sets the value of the arguments,
  - meeting a precondition is the **client's responsibility**;
  - a *precondition violation* is the manifestation of a *bug* **in the client**.
- The PELICANS instruction for stating a precondition is **`PEL_CHECK_PRE`**
- Usage:

```
#include <PEL_assertions.hh>

bool PEL_CHECK_PRE( bool expression ) ;
```

- If **`expression`** evaluates to **`false`**, **`PEL_CHECK_PRE`**
  - directs an error message to the **`ostream`** class object **`cerr`**
  - and terminates the program.
- This only happens when the associated *function member* is compiled with the *compilation level* **`opt1`**, **`opt2`** or **`dbg`**

Non-Redundancy Principle

The body of a function must not test for the function's preconditions.

Precondition Design: Tolerant or Demanding?

Let us consider the **semantic** consistency conditions required for a proper functioning of each client-supplier cooperation represented by a function.

- **demanding approach**: the responsibility to enforce these consistency conditions is assigned to the **clients**.

  ⇒ the consistency conditions appear as **preconditions** of the function.
- **tolerant approach**: the responsibility to enforce these consistency conditions is assigned to the **supplier**.

  ⇒ the consistency conditions appear as conditional instructions **in the body** of the function (*e.g.* **`if-then-else`**).
- These two approaches are mutually exclusive.

Precondition Availability Rule

Every member appearing in the precondition of a function must be available to every client to which the function is available.

III.7.4 Postconditions

- A *postcondition* is an assertion evaluated **after** a *function member* finishes executing and before the message result is returned to the client.
- A *postcondition* expresses **properties** on:
  - the outgoing value of the arguments,
  - the outgoing result, and
  - the object state

  guaranteed at the exit of the *function member*, given the message that activated it and the initial object state.
- Because the postcondition verifies that the supplier's promise is met,
  - meeting the postcondition is the **supplier's responsibility**;
  - a *postcondition violation* is the manifestation of a *bug* **in the supplier**.
- The PELICANS instruction for stating a *postcondition* is **`PEL_CHECK_POST`**
- Usage:

```
#include <PEL_assertions.hh>

bool PEL_CHECK_POST( bool expression ) ;
```

- If **`expression`** evaluates to **`false`**, **`PEL_CHECK_POST`**
  - directs an error message to the **`ostream`** class object **`cerr`**
  - and terminates the program.
- This only happens
  - when the associated *function member* is compiled with the *compilation level* **`opt2`** or **`dbg`**
  - when the option **`-Cpost`** or **`-Call`** is specified on the command line.

III.7.5 Class Invariants

- A class invariant is an assertion evaluated
  - after instantiation of any object of a class,
  - just before destruction of any object of a class,
  - upon entry and exit from every method.
- A class invariant specifies properties that must be true for every object of that class.
- If all common requirements are factored out of completely specified preconditions and postconditions, we obtain the class invariant. The class invariant consolidates conditions that would appear in every precondition and in every postconditions.
- The PELICANS instruction for testing a *class invariant* is **PEL_CHECK_INV**
- Usage:

```
#include <PEL_assertions.hh>

bool PEL_CHECK_INV( bool expression ) ;
```

- If **expression** evaluates to **false**, **PEL_CHECK_INV**
  - directs an error message to the **ostream** class object **cerr**
  - and terminates the program.
- This only happens
  - when the associated *function member* is compiled with the *compilation level* **opt2** or **dbg**
  - when the option **-Call** is specified on the command line.

Practically **PEL_CHECK_INV** is called with **expression** equal to the return value of a **protected virtual** function member **invariant()**, declared in **PEL_Object**, that implements the invariant property of the associated class.

```
PEL_CHECK_INV( invariant() ) ;
```

III.7.6 Other Assertions

- The PELICANS instruction for assertions that are always activated is **PEL_ASSERT**
- Usage:

```
#include <PEL_assertions.hh>

bool PEL_ASSERT( bool expression ) ;
```

- If **expression** evaluates to **false**, **PEL_ASSERT**
  - directs an error message to the **ostream** class object **cerr**
  - and terminates the program.
- **PEL_ASSERT** expressions cannot be deactivated, neither at translation time nor at execution time.

- The PELICANS instruction for assertions of the lowest enabling level is **PEL_CHECK**
- Usage:

```
#include <PEL_assertions.hh>

bool PEL_CHECK( bool expression ) ;
```

- If **expression** evaluates to **false**, **PEL_CHECK**
  - directs an error message to the **ostream** class object **cerr**
  - and terminates the program.
- This only happens
  - when the associated *function member* is compiled with the *compilation level* **opt2** or **dbg**
  - when the option **-Call** is specified on the command line.

III.7.7 Assertions Enabling Hierarchy

Compilation

assertion instructions might not be translated at compile time, depending on the *compilation level* **opt0**,**opt1**,**opt2**,**dbg** :

instruction	ignored	compiled
PEL_ASSERT	never	**opt0**,**opt1**,**opt2**,**dbg**
PEL_CHECK_PRE	**opt0**	**opt1**,**opt2**,**dbg**
PEL_CHECK_POST **PEL_CHECK_INV**, **PEL_CHECK**	**opt0**,**opt1**	**opt2**,**dbg**

Execution

Even when they have been compiled, assertion instructions might not be executed, depending of the presence of the **command line option** **-Cpost**, **-Call** :

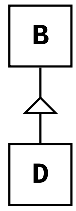
instruction	executed
PEL_ASSERT	always
PEL_CHECK_PRE	always
PEL_CHECK_POST	**-Cpost**,**-Call**
PEL_CHECK_INV, **PEL_CHECK**	**-Call**

III.7.8 Ancestor/Descendant Contracts

Liskov Substitution Principle

The objects of a subclass ought to behave the same as those of the supertype as far as anyone or anyprogram using supertype objects can tell.

Understanding "Substitution"



If **D** inherits from **B**, the LSP states that:

1. every object of type **D** is also an object of type **B**, but not *vice versa*;
2. if you need an object of type **D**, an object of type **B** will not do ;
3. **anywhere** an object of type **B** can be used, an object of type **D** can be used **just as well**.

⇒ Any *overriden* function member in class **D**

- must accept all calls that were acceptable to the original;
- must guarantee at least as much as was guaranted by the original.

It may, but does not have to, accept more cases, or provide stronger guarantees.

Assertion Overriding Rule

An overridden member function may only:

- replace the original precondition by one equal or weaker;
- replace the original postcondition by one equal or stronger.

Most often, the precondition and the postcondition of an overriden function member are identical to the original ones.

"REQUIRE NO MORE and PROMISE NO LESS"

Invariant in Derived Classes

- The invariant of the base class apply to the derived class.
- The invariant property of a derived class is the boolean **and** of the assertions appearing in its specific invariant properties and the invariant property of its base class.

III.7.9 Instantiation of the PELICANS Framework

- Let **`Base`** be a *plug-point* of PELICANS.
- We will consider:
  - the invariant definition in derived classes;
  - the overriding of the function member **`foo`**.
- The *preconditions* and *postconditions* of **`foo`** are made available to *client authors* through two *function members* **`foo_PRE`** and **`foo_POST`**.
- **`foo_PRE`** and **`foo_POST`** are non pure **`virtual`** methods: they will be dynamically linked, they might be overriden and they have a default implementation.

```
class Base [ : public ... ]
{
   [public/protected]:
      virtual [return-type] foo( [flist] ) [const] [=0] ;

   protected:
      virtual bool foo_PRE( [flist] ) const ;
      virtual bool foo_POST( [flist_POST] ) const ;

      virtual bool invariant( void ) const ;
} ;
```

- Let **`Derived`** be a heir of **`Base`**.
- The invariant property of **`Derived`** is the boolean **and** of the assertions appearing in its specific invariant properties and the invariant property of **`Base`** :

```
class Derived : public Base
{
   protected:

      virtual bool invariant( void ) const ;
} ;

bool Derived:: invariant( void ) const
{
   PEL_ASSERT( Base::invariant() ) ;
   PEL_ASSERT( [first Derived specific predicate] ) ;
   ...
   PEL_ASSERT( [last  Derived specific predicate] ) ;
   return( true ) ;
}
```

- Let **`Derived`** be a **non-leaf** heir of **`Base`** (**`Derived`** has heirs).
- The overriden function member **`foo`** may also be overriden in heirs of **`Derived`**.

```
class Derived : public Base
{
   public:
      virtual [return-type] foo( [flist] ) [const] ;

   protected:
      virtual bool foo_PRE( [flist] ) const ;
      virtual bool foo_POST( [flist_POST] ) const ;
} ;

[return-type] Derived::foo( [flist] ) [const]
{
   PEL_LABEL( "Derived::foo" ) ;
   PEL_CHECK_PRE( foo_PRE( [flist] ) ) ;
   PEL_CHECK_INV( invariant() ) ;
   /* method body */
   PEL_CHECK_INV( invariant() ) ;
   PEL_CHECK_POST( foo_POST ( [flist_POST] )  ) ;
   return( result ) ;
}
```

- The overriden precondition may only be **equal** to the original, or **weaker**:

```
bool Derived::foo_PRE( [flist] ) const
{
   PEL_ASSERT( Base::foo_PRE( [flist] )
               || [first Derived specific predicates]
               || ...
               || [last Derived specific predicates] ) ;
   return( true ) ;
}
```

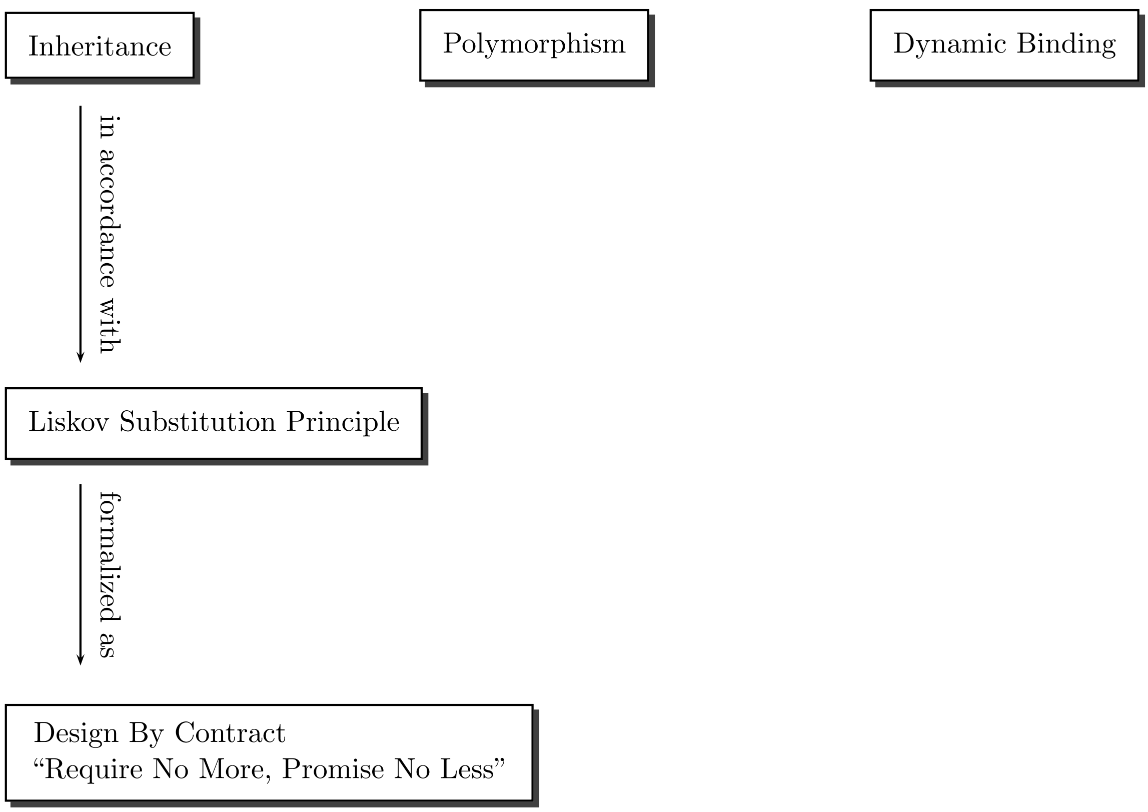
- The overriden postcondition may only be **equal** to the original, or **stronger**:

```
bool Derived::foo_POST( [flist_POST] ) const
{
   PEL_ASSERT( Base::foo_POST( [flist_POST] )  ) ;
   PEL_ASSERT( [first Derived specific predicates] ) ;
   PEL_ASSERT( ... ) ;
   PEL_ASSERT( [last Derived specific predicates] ) ;
   return( true ) ;
}
```

- Let **Derived** be a **leaf** heir of **Base**.
- There is no need to override the function members **foo_PRE** and **foo_POST**.
  The overriden precondition may only be **equal** to the original, or **weaker**.
  The overriden postcondition may only be **equal** to the original, or **stronger**.

```
class Derived : public Base
{
   [public/protected/private]:
      virtual [return-type] foo( [flist] ) [const] ;
} ;

[return-type] Derived::foo( [flist] ) [const]
{
   PEL_LABEL( "Derived::foo" ) ;
   PEL_CHECK_PRE( [Derived specific predicates]
                  || Base::foo_PRE( [flist] ) ) ;
   PEL_CHECK_INV( invariant() ) ;
   /* method body */
   PEL_CHECK_INV( invariant() ) ;
   PEL_CHECK_POST( Base::foo_POST( [flist_POST] ) ) ;
   PEL_CHECK_POST( [Derived specific predicates] ) ;
   return( result ) ;
}
```

III.7.10 Key Concepts of Object-Oriented Programming



III.8 Reference Documentation

The documentation under review here is the *class reference documentation*, not the *user* documentation which is to be found elsewhere.

Danger : if there is any worse situation than having no documentation, it must be having wrong documentation.

III.8.1 Self-Documentation Principle

If there is any worse situation than having no documentation, it must be having wrong documentation.

Goal

Guarantee that the software and its reference documentation remain compatible **when things start changing**.

Strategy

- As much as possible, information about a PELICANS class appears in the class itself rather than externally.
- The text of PELICANS classes is written so that it includes all the elements needed for its documentation, recognizable by a dedicated tool that is made available to extract documentation elements automatically.

III.8.2 Information about Classes for the Clients

Accessible Part of a PELICANS Class

Let **`A`** be a PELICANS class.

- If **`A`** is a *plug-point*: let **`C`** be an user implemented class derived from **`A`**.
  Part of the interface of **`A`** that is *accessible* to **`C`**: **`public`** and **`protected`** sections.
- Let **`C`** be a class that uses **`A`** as a component class.
  Part of the interface of **`A`** that is *accessible* to **`C`**: **`public`** section.

Visible Part of a PELICANS Class

The part of a PELICANS class involved in an application that is visible to another class of that application is defined as:

- the *accessible* part of its interface;
- the preconditions of all *accessible* methods;
- among the postcondition assertions of all *accessible* methods, those involving *accessible* members;
- among the invariant assertions, those involving *accessible* members;
- the *accessible* part of the interface of all its ancestors (recursive definition).

III.8.3 Entries in the Reference Documentation

Learning About a Class

A **client author** can rely on:

1. the header comments of the class itself and the header comments of all its ancestors;
2. the visible part of the interface of the class itself together with the visible part of all its ancestors.

Learning About a Function Member

A **client author** can rely on **five** kind of properties:

1. name;
2. signature;
3. return type (if any);
4. precondition and postcondition (if any);
5. header comments.

III.8.4 Names

Name of Classes

A class name is always either:

- a noun, possibly qualified;
- only for abstract classes describing a structural property: an adjective.

Name of Function Members

The name of methods reflects the **command-query separation principle**.

- *Command* names begin with verbs in the infinitive or in the imperative, possibly followed with complements.
- *Query* names never include imperative or infinitive verbs.
- Non-boolean *query* names are nouns, possibly qualified.
- Boolean *query* names are adjective (possibly starting with **`is`**) or verbs conjugated at the third person, expressing properties of the current object that are either true or false.

III.8.5 Header Comments

Class Header Comments

- first sentence: describes the role of the class, expressed in terms of its instances
- subsequent comments: additional insights on concepts involved in the visible part
- final paragraph in *plug-points*, entitled **`FRAMEWORK INSTANTIATION`**: describes the steps for inheriting from that class and writing the code called by the *framework* itself.

Header Comments of Function Members

Telegram like style is used for function header comments.

- Preconditions and postconditions appear on specifically labelled entries and are never duplicated in the header comment.
- Type information of arguments are not repeated in the header comments.
- The current object is mentioned only when necessary.
- Header comments for commands are imperative in the style of marching orders, and always end with a period.
- Header comments for non-boolean queries never use a verbal form, but simply name what the query returns, typically using a qualified noun. They never end with a period.
- Header comments for boolean queries adopt the form of a question, terminated by a question mark.

III.9 Why Object-Oriented Programming?

III.9.1 Structured Programming

- A programmer should understand:
  1. the concept of programming;
  2. the pragmatics of doing it in one particular language.
- *Structured Programming* (Edsgar Dijkstra) says that all program (independently of the programming language) could be structured in the following four ways:
  - sequences of instructions;
  - loops;
  - conditional statements (branching);
  - modules.

  Additional features make these four structures useful:
  - data and variables;
  - operations;
  - input/output capabilities.

> The first requirement of *object-oriented programming* is
> an appropriate application of the principles of *structured programming*

III.9.2 Designs in a World of Changing Requirements

Volatile Requirements

- Scientific software requirements change throughout the lifetime of the software product as both physical models are refined and numerical methods perfected.
- Challenge: create designs that are stable in the face of change, *i.e.* "good" designs.

Good and Bad Designs

- A *good design* is difficult to define, but everybody should agree with the following criteria defining a *bad design*.
- A piece of code that **fulfills its specifications** and **yet** exhibits any of the following three traits has a *bad design*:

  Rigidity hard to change because every change affects too many other parts of the system

  Fragility when you make a change, unexpected parts of the system break

  Immobility hard to reuse in another application because it cannot be disentangled from the current one

III.9.3 The Cause of Bad Designs

> bad designs are caused by
> **improper dependencies** (nature and strength) between
> *software modules*

Rigidity a single change to heavily interdependent software begins a cascade of changes in dependent modules.

Fragility a single change to heavily interdependent software might yield breakage in areas that have no conceptual relationship with the area that was changed.

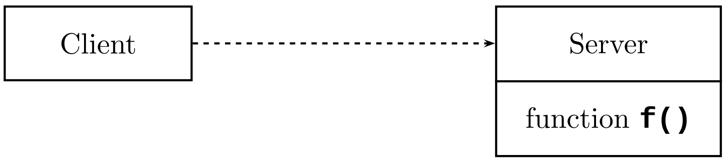
Immobility the desirable parts of the design are highly dependent upon other details that are not desired.

III.9.4 Example of a Rigid and Immobile Design

Split mentally the software system into a *client* module and a *server* module.

The *client* uses the *server*.

```
fclient()
{
   f() ;
}
```

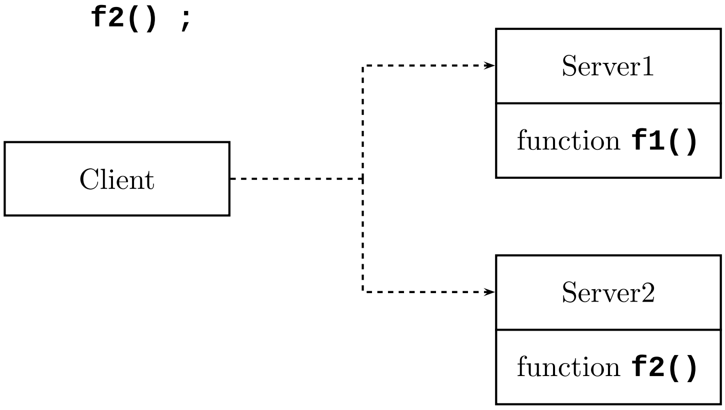


- Changes to the server might propagate to the client (Rigidity).
- If we wish for the client to use a different server, then the client must be changed, at list to name the new server (Immobility).

Adaptation of the *client* to a new context (*ie* a different *server*) implies a modification of the *client*

- to check for the context;
- to name the new *server*.

```
fclient()
{
   if( context1 )
      f1() ;
   else if( context2 )
      f2() ;
}
```



III.9.5 The Route for Good Designs

The Open-Closed Principle

Software modules should be:

- **open for extension**

  the module behavior can be modified and/or extended as the requirements change, or to meet the needs of new applications

- **closed for modification**

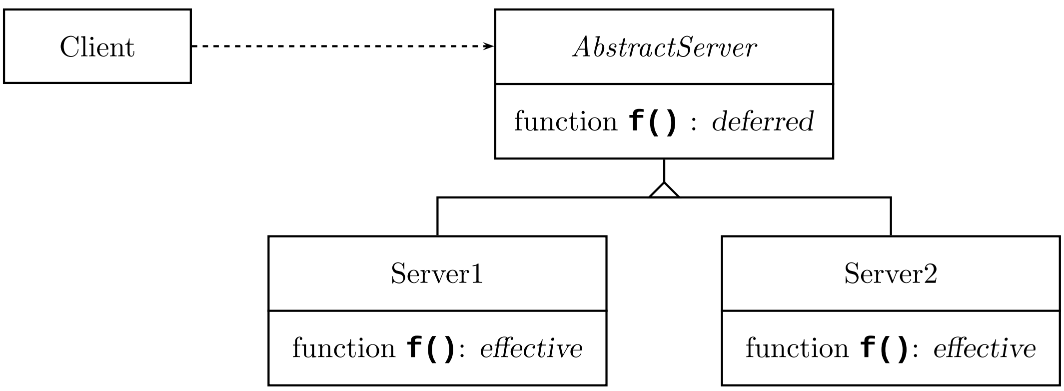
  the module source code is inviolate, no one is allowed to change it

Abstraction is the Key

- OO Languages provide the notion of abstract base classes and derived classes.
- An abstract base class
  - is fixed;
  - and yet represent an unbounded group of possible behaviors.
- All the possible derived classes represent all possible behaviors.
- A module manipulating an abstract base class has:
  - an inviolate source code since it depends upon an abstraction ;
  - an extendible behavior through creation of derivatives of the abstraction.

III.9.6 Example of a Good Design

```
fclient( AbstractServer* s )
{
   // polymorphic call
   // dynamic binding
   s->f() ;
}
```



Inheritance, **Polymorphism**, **Dynamic Binding** in action . . .

III.9.7 Principles of Object Oriented Class Design

The Open-Closed Principle

Software modules should be open for extension and closed for modification.

The Dependency Inversion Principle

The structural implication of using abstractions to fulfill the Open-Closed Principle can be generalized into the Dependency Inversion Principle:

- High level modules should not depend upon low level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

The Liskov Substitution Principle (LSP)

The pragmatic implication of the Dependency Inversion Principle is the Liskov Substitution Principle, formalized in term of Design by Contact:

- Everywhere an object of a base-class can be used, an object of any of the derived-classes can be used just as well.
- Any object of a derived-class should be able to honor any contract that an object of the base-class can honor.
- An overriden function member may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.

III.9.8 Violation of the LSP: a Simple Example

Anytime you find yourself writing code of the form:

- if the object is of type **T1**, then do something,
- but if it's of type **T2**, then do something else,

"slap yourself" [Scott Meyers]. That is not the object oriented way.

Use *dynamic binding* instead (in **C++**, **virtual** functions).

```
class B { ... } ;
class D1 : public B { ... } ;
class D2 : public B { ... } ;

// violation of the Liskov Substitution Principle
void foo( B* b ) {
   D1* d1 = dynamic_cast<D1*>( b ) ;
   D2* d2 = dynamic_cast<D2*>( b ) ;
   if( d1 != 0 ) {
      d1->foo() ;
   else if( d2 != 0 ) {
      d2->foo() ;
   }
}
```

III.9.9 Violation of the LSP: a More Subtle Example

- Consider an abstract base class **Mat** for representing matrices:

```
class Mat {
public:
   virtual size_t nb_cols( void ) const = 0 ;
   virtual size_t nb_rows( void ) const = 0 ;
   virtual double item( size_t i, size_t j ) const = 0 ;
   virtual void set_item( size_t i, size_t j, double x ) = 0 ;
} ;
```

- The concrete class of full matrices **FullMat** is derived from **Mat**.
- A symmetric matrix is a matrix, so we could think that the concrete class of symmetric matrices **SymMat** should be derived from **Mat**.
- Consider the following client of the **Mat** hierarchy:

```
void f( Mat* m ) {
   m->set_item( 1, 2, 3.5 ) ;
   m->set_item( 2, 1, 0.1 ) ;
}
```

The LSP states that “everywhere an object of a base-class can be used, an object of any of the derived-classes can be used just as well”, but what is the meaning of:

```
FullMat* mfull = ... ;          SymMat* msym = ... ;
f( mfull ) ;                    f( msym ) ;
```

- The reasonable contract for **Mat::set_item** is:

precondition $\begin{cases} \texttt{i} < \texttt{nb\_rows()} \\ \texttt{j} < \texttt{nb\_cols()} \end{cases}$ postcondition $\begin{cases} \texttt{item(i,j)==x} \\ \text{other } \texttt{item(k,l)} \text{ unchanged} \end{cases}$

- Algebraically, the following precondition for **SymMat::set_item** makes sense:

$\texttt{i}<\texttt{nb\_rows()} \quad \texttt{j}<\texttt{nb\_cols()} \quad \text{and} \quad \texttt{i}\leq\texttt{j}$

but it is **more restrictive** than the original.

- Keeping the original precondition leads to the following kind of implementation:

```
void SymMat::set_item( size_t i, size_t j, double x ) {
   A[i][j] = x ; // obvious notations for A
   A[j][i] = x ; // to maintain symmetry
}
```

but now, the postcondition is

item(i,j)==x **item(j,i)==x** other **item(k,l)** unchanged

which is inconsistent with the original.

- Doesn’t the ISA relationship hold ?
  - No! A symmetric matrix might be a kind of matrix, but a **SymMat** object is definitely not a **Mat** object.
  - The **behavior** of a **SymMat** object is not consistent with that of a **Mat** object.

Chapter IV

Native Data Structures of PELICANS

IV.1 Hierarchical Data System

IV.1.1 Essentials

PELICANS defines a format of data structures, called *Hierarchical Data System*.

- Data are organized into a tree-like structure (such as directory/file trees). The nodes of the hierarchy are called *modules* while the leaves are the data themselves.
- Every *module* may contain:
  - other *modules*;
  - basic *entries*;
  - definitions of *variables*.

Each *module* defines a *module hierarchy* (a *module* contains other *modules* themselves containing other *modules* and so on). The upper module of a *module hierarchy* is denoted the *root module*.

IV.1.2 Managing Hierarchical Data Structures

The management of *hierarchical data structures* is assigned to a set of dedicated "off-the-shelf" software components of PELICANS.

To interrogate such a hierarchical data structure, PELICANS provides the class of navigators called **`PEL_ModuleExplorer`**.

Every instance of **`PEL_ModuleExplorer`**:

- is attached to a *module*;
- offers a secure access to the *entries* of that *module*;
- provides a delivery mecanism of other **`PEL_ModuleExplorer`** instances attached to the sub-*modules* contained by that *module*.

Hence navigation through a *hierarchical data structure* relies on **`PEL_ModuleExplorer`** instances, one for each *module* that is traversed or interrogated.

IV.1.3 File Representation

Hierarchical data structures can be read from and written into files with the text or binary format.

An example is given below.

```
MODULE my_nice_module
   $DS_SINx = sin( $DS_X )   // a variable definition
   // a contained module
   MODULE inner
      $DS_X = 3.14           // a variable definition
      key = 1                // a basic entry
   END MODULE inner
   door = "closed"           // a basic entry
END MODULE my_nice_module
```

In these files, the `//` characters begin a *comment*, which extends to the end of the line. PELICANS ignores comments, their purpose is to explain the data structure to a human reader.

IV.1.4 Main Occurences of HDS

Users of PELICANS are perpetually confronted with *hierarchical data structures*.

The following two situations are essential.

Delivery Methods

The main mechanism for creating PELICANS objects (§III.3.1) involves *factory methods* (§III.5.3), that are **`static`** member functions usually called **`create`** or **`make`**.

These *factory methods* usually have a *parameter*:

- whose type is: *pointer* to **`PEL_ModuleExplorer`**;
- whose usefulness is to communicate, *via* the attached *hierarchical data structure* (loaded in memory), all necessary informations for the requested delivery.

Let us consider, as an example, a PELICANS-based application whose purpose is the solution of a partial differential equations system. Accessing the PELICANS components devoted to the discretization of such a problem is carried out by way of the *facade* class **`PDE_DomainAndFields`**. An instance of that class can be created by the following statement:

```
// exp of type PEL_ModuleExplorer*
PDE_DomainAndFields const* dom = PDE_DomainAndFields::create( 0, exp ) ;
```

where **`exp`** is attached to a *module* of a *hierarchical data structure* whose organization and contents are defined by the **`PDE_DomainAndFields`** class itself.

Data decks for PELICANS-based applications

A PELICANS-based application may be executed by typing a command of the type:

```
/home/martin/MyAppli/test >  ../bin/exe data.pel
```

or, preferably, using the **`pel run`** utility (§V.4.3):

```
/home/martin/MyAppli/test >  pel run ../bin/exe data.pel resu
```

In these commands, **../bin/exe** denotes an executable file of the considered PELICANS-based application, and **data.pel** denotes a file storing a *hierarchical data structure* that constitutes the data deck of the considered PELICANS-based application.

Obviously, this *hierarchical data structure* depends on the application. A possible general form is:

```
MODULE PEL_Application
   concrete_name = "xxxx"
   ...
END MODULE PEL_Application
```

where **"xxxx"** is the name of the considered PELICANS-based application (§III.6.4).

IV.1.5 Basic Entries

The *entries* of *hierarchical data structures* are { *keyword* ; *data* } pairs, where the *data* have two fondamental properties:

- a *type*;
- a *value*.

In the file representation, the *keyword* and the *data* are separated by the **=** character.

For example:

```
concrete_name = "PDE_StepByStepProgression"
// keyword - concrete_name
// data    - type : PEL_Data::String
//           value: "PDE_StepByStepProgression"

mesh_polyhedron = < "GE_Segment" "GE_Rectangle" >
// keyword - mesh_polyhedron
// data    - type : PEL_Data::StringVector
//           value: vector of 2 items "GE_Segment" and "GE_Rectangle"

vertices_coordinate_0 = regular_vector( 0.0, 10, 0.3 )
// keyword - vertices_coordinate_0
// data    - type : PEL_Data::DoubleVector
//           value: vector of 10 reals evenly spaced between 0. and 0.3

value = vector( 1.+tanh(20.*component($DV_X,0)+10.) )
// keyword - value
// data    - type : PEL_Data::DoubleVector
//         - value: vector of 1 real containing the expression value
```

The enumerated type called **`Type`**, defined within the **`PEL_Data`** class, specifies all the authorized *types* for the *data*:

type of a data	enumerator of **`PEL_Data::Type`**	example of data
boolean	**`Bool`**	**`true`**
real	**`Double`**	**`3.14`**
integer	**`Int`**	**`2`**
string	**`String`**	**`"hello"`**
vector of reals	**`DoubleVector`**	**`< 1.1 1.4 >`**
vector of integers	**`IntVector`**	**`< 1 -3 >`**
vector de booleans	**`BoolVector`**	**`< true true false >`**
vector of strings	**`StringVector`**	**`< "hello" "world" >`**
2D array of reals	**`DoubleArray2D`**	**`array( <0. 1.>, <2. 3.> )`**
3D array of reals	**`DoubleArray3D`**	**`array( array( <0. 1.>, <2. 3.> ) )`**
2D array of integers	**`IntArray2D`**	**`array( <0 -1>, <-2 3> )`**
3D array of integers	**`IntArray3D`**	**`array( array( <0 1>, <2 -3> ) )`**

IV.1.6 Special Data: Expressions

An *expression* is a special *data*

- that accepts *arguments*,
- whose *type* may depend on its actual *arguments*,
- whose *value* is the result of a calculation that is performed only when explicitly asked for (one moment which is called: *evaluation* of the *expression*).

In a *hierarchical data structure* read from a file, the *type* of the *expressions* is known at reading-time, on the other hand, their *value* may be dynamic (*ie* unknown at reading-time) if their arguments contain *variables*).

The following table gives some examples of *expressions*.

expression	type	evaluation result
`"Hello " + "World"`	`String`	`"Hello World"`
`1.0 + 2.1`	`Double`	`3.1`
`1.0 - 2.1`	`Double`	`-1.1`
`5 / 2`	`Int`	`2`
`5.0 / 2.0`	`Double`	`2.5`
`2.1 * 10.0`	`Double`	`21.0`
`component( < 1 2 >, 1 )`	`Int`	`2`
`size( < 1 2 > )`	`Int`	`2`
`( false ? 3.1 : 1.1 )`	`Double`	`1.1`
`( true ? 3 : 1 )`	`Int`	`3`
`1 < 2`	`Bool`	`false`
`1 <= 1`	`Bool`	`true`
`false || true`	`Bool`	`true`
`false && true`	`Bool`	`false`
`!true`	`Bool`	`false`
`sqr( $DS_xx )`	`Double`	square of **$DS_xx**
`sqrt( $DS_xx )`	`Double`	square root of **$DS_xx**
`sin( $DS_xx )`	`Double`	sine of **$DS_xx**
`cos( $DS_xx )`	`Double`	cosine of **$DS_xx**
`exp( $DS_xx )`	`Double`	base-e exponential of **$DS_xx**
`log( $DS_xx )`	`Double`	natural (base-e) logarithm of **$DS_xx**
`pi()`	`Double`	3.141592653589793238
`getenv( "PELICANSHOME" )`	`String`	**PELICANSHOME** environment variable
`dirname("/usr/local/tt")`	`String`	`"/usr/local"` (on UNIX systems)
`basename("/usr/local/tt")`	`String`	`"tt"` (on UNIX systems)
`getcwd()`	`String`	working directory pathname
`join( "r1", "r2", "nn" )`	`String`	`"r1/r2/nn"` (on UNIX systems)
`this_file_dir()`	`String`	pathname of the file containing this expression
`"toto." + to_string( 2 )`	`String`	`"toto.2"`
`unit_sort( 0., 0., 1., 2 )`	`Int`	`0`
`unit_sort( 0.9, 0., 1., 2 )`	`Int`	`1`

It should ne noted that the set of available expression may be freely enriched by the user since the **PEL_Expression** class is a *plug-point* of the PELICANS *framework*.

IV.1.7 Variables and Context

▷ A *variable* is a typed value that has a name.

▷ The name of a *variable* is made of two parts separated by a underscore (_).

- The part on the right of the underscore identifies the variable in its singularity.
- The part on the left of the underscore indicates its type. It has the general form: XY, where X denotes the associated scalar type and Y denotes its rank as an n-dimension tensor (*ie* the number of indices required to identify one of its components).

  The following nomenclature is used:

X	denoted type
D	**PEL_Data::Double**
I	**PEL_Data::Integer**
S	**PEL_Data::String**
B	**PEL_Data::Bool**

Y	denoted rank
S	1 (scalar)
V	2 (vector)
A	3 (array)

▷ In the file representation of a *hierarchical data structure*, the *variable* names appear prefixed with the **$** sign (so that they can be distinguished from other syntactic constructs).

▷ The following table gives some examples.

name	type	example of definition
DS_temperature	**Double**	**$DS_temperature = 315.0**
IS_NbMailles	**Int**	**$IS_NbMailles = 1000**
SS_fichier	**String**	**$SS_fichier = "mon_fichier.txt"**
DV_X	**DoubleVector**	**$DV_X = < 1.0 0.0 >**
DV_value	**DoubleVector**	**$DV_value = vector($DS_temperature)**

▷ A *variable* defined in a *module* can be used in any sub-*module* included in the defining *module*.

▷ The *context* of a *module* is defined by the collection of the variables that can de used in that module: it is the set of variables defined there, concatenated with the context of the possible enclosing *module* (recursive definition).

IV.2 Predefined Expressions

IV.2.1 `abs`

- defining class: **`PEL_MathFunctionExp`**

IV.2.2 `acos`

arc cosine function

- type: **`PEL_Data::Double`**
- 1 argument: **`PEL_Data::Double`**
- evaluation: the return value is the principal value of the arc cosine of the argument in the range $[0, \pi]$
- examples:

expression	evaluation result
`acos( 1.0 )`	`0.0`
`acos( 0.5 )`	$\pi/3$
`acos( cos( -pi()/3. ) )`	$\pi/3$
`acos( cos( pi()/3. + 4.*pi() ) )`	$\pi/3$

- defining class: **`PEL_MathFunctionExp`**

IV.2.3 `acosh`

inverse hyperbolic cosine function

- type: **`PEL_Data::Double`**
- 1 argument: **`PEL_Data::Double`**
- evaluation: the return value is the inverse hyperbolic cosine of the argument
- example:

expression	evaluation result
`acosh( 1.0 )`	`0.0`
`acosh( cosh( 40.0 ) )`	`40.0`
`acosh( cosh( -3.0 ) )`	`3.0`

- defining class: **`PEL_MathFunctionExp`**

IV.2.4 `apply`

vector built applying a given function to the components of a given initial vector

- type: that of the given initial vector
- 3 or 4 arguments: vector type; function to be applied to the initial vector component (same type); **`PEL_Data::PEL_String`** corresponding to the name of the variable (same type than the intial vector components) used in the second argument function (to be substitued by the value of the components of the first argument vector) and an optional 4th argument **`PEL_Data::PEL_String`** corresponding to the name of the variable (**`PEL_Int type`**) used in the second argument function (to be substitued by the index of the components of the first argument vector)

- example:

expression	evaluation result
`apply( < 1. 2. 3. >,` `$DS_x*$DS_x, "DS_x" )`	`< 1. 4. 9. >`
`apply( < true true false >,` `!$BS_x, "BS_x" )`	`< false false true >`

`apply( < "titi" "toto" >,`
`$SS_x + to_string($IS_ic),`
`"SS_x", "SS_ic" )`
	`< "titi0" "toto1" >`

- defining class: **`PEL_VectorExp`**

IV.2.5 `array`

- "2D" case
  - "Int" case
    - type: **`PEL_Data::IntArray2D`**
    - any number of arguments, all of type **`PEL_Data::IntVector`**
  - "Double" case
    - type: **`PEL_Data::DoubleArray2D`**
    - any number of arguments, all of type **`PEL_Data::DoubleVector`**
  - evalution: the i-th line of the return value A is the i-th argument (possibly supplemented with zeros if the size of that argument is strictly "smaller" than the "largest" argument)
    $A(i, j) = j$-th item of the i-th argument if any, zero if none
    $0 \leq i <$ number of arguments $0 \leq j <$ maximum size of the arguments
- "3D" case
  - "Int" case
    - type: **`PEL_Data::IntArray3D`**
    - any number of arguments, all of type **`PEL_Data::IntArray2D`**
  - "Double" case
    - type: **`PEL_Data::DoubleArray3D`**
    - any number of arguments, all of type **`PEL_Data::DoubleArray2D`**
  - evalution: the return value A is a 3D array that may be viewed as a collection of horizontal planes, one for every value of the first index; the i-th plane is the i-th argument (possibly supplemented with zeros if that argument is strictly "smaller" than the "largest" argument)
    $A(i, j, k) = (j, k)$-th item of the i-th argument if any, zero if none
    $0 \leq i <$ number of arguments
    $0 \leq j \leq$ maximum first index value of the arguments
    $0 \leq k \leq$ maximum second index value of the arguments
- examples:

expression	evaluation result
`array( < 0. 1. -1. >, < 2. 3. 4. > )`	$\begin{pmatrix} \texttt{0.} & \texttt{1.0} & \texttt{-1.0} \\ \texttt{2.} & \texttt{3.0} & \texttt{4.0} \end{pmatrix}$
`array( < 0. 1. -1. >, < 2. 3. > )`	$\begin{pmatrix} \texttt{0.} & \texttt{1.} & \texttt{-1.} \\ \texttt{2.} & \texttt{3.} & \texttt{0.} \end{pmatrix}$
`array( < 0 1 -1 >, < 2 3 > )`	$\begin{pmatrix} \texttt{0} & \texttt{1} & \texttt{-1} \\ \texttt{2} & \texttt{3} & \texttt{0} \end{pmatrix}$

- defining class: **PEL_ArrayExp**

IV.2.6 `asin`

- defining class: **PEL_MathFunctionExp**

IV.2.7 `asinh`

- defining class: **PEL_MathFunctionExp**

IV.2.8 `atan`

- defining class: **PEL_MathFunctionExp**

IV.2.9 `atan2`

- defining class: **PEL_MathFunctionExp**

IV.2.10 `atanh`

- defining class: **PEL_MathFunctionExp**

IV.2.11 `basename`

- defining class: **PEL_SystemExp**

IV.2.12 `binary`

- defining class: **PEL_BinStored**

IV.2.13 `ceil`

ceiling function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double**
- evaluation: the return value is the smallest integral value greater than or equal to the argument
- examples:

expression	evaluation result
`ceil( 7.0 )`	`7.0`
`ceil( 7.1 )`	`8.0`
`ceil( 7.9 )`	`8.0`
`ceil( -12.0 )`	`-12.0`
`ceil( -12.2 )`	`-12.0`
`ceil( -12.8 )`	`-12.0`

- defining class: **PEL_MathFunctionExp**

IV.2.14 `component`

- defining class: **`PEL_VectorExp`**

IV.2.15 `conditional_vector`

vector build from a set of scalar value, each of them being retained or not depending on the result of a boolean expression

- type: vector type corresponding to the given scalar values
- 2.N arguments: **`PEL_Data::Bool`** ; scalar type
- examples:

expression	evaluation result
`conditional_vector( true, 1.0, false, 2.0 )`	`< 1.0 >`
`conditional_vector( true, "un", true, "deux" )`	`< "un" "deux" >`
`conditional_vector( true, true )`	`< true >`

- defining class: **`PEL_VectorExp`**

IV.2.16 `cos`

cosine function

- type: **`PEL_Data::Double`**
- 1 argument: **`PEL_Data::Double`**
- evaluation: the return value is the cosine of the argument (measured in radians)
- defining class: **`PEL_MathFunctionExp`**

IV.2.17 `d`

- defining class: **`PEL_DerivativeExp`**

IV.2.18 `data_with_context`

- defining class: **`PEL_DataWithContextExp`**

IV.2.19 `default_roundoff`

rounding off

- type: **`PEL_Data::Double`**
- 3 arguments: **`PEL_Data::Double`** (denoted x) ; **`PEL_Data::Int`** (denoted n) ; **`PEL_Data::Double`** (denoted ε)
- evaluation: the return value is the round off of x computed by keeping n significant digits if $x > \varepsilon$, 0 otherwise.
- examples:

expression	evaluation result
`default_roundoff( 7326.78314912, 5, 1.e-6 )`	`7326.78`
`default_roundoff( 73.2678314912, 5, 1.e-6 )`	`73.2678`
`default_roundoff( 0.732678314912, 5, 1.e-6 )`	`0.73268`
`default_roundoff( 0.732678314912e-3, 5, 1.e-6 )`	`0.73268e-3`
`default_roundoff( 0.732678314912e-6, 5, 1.e-6 )`	`0.0`
`default_roundoff( -7326.78314912, 5, 1.e-6 )`	`-7326.78`
`default_roundoff( -73.2678314912, 5, 1.e-6 )`	`-73.2678`
`default_roundoff( -0.732678314912, 5, 1.e-6 )`	`-0.73268`
`default_roundoff( -0.732678314912e-3, 5, 1.e-6 )`	`-0.73268e-3`
`default_roundoff( -0.732678314912e-6, 5, 1.e-6 )`	`0.0`

- defining class: **GE_RoundoffExp**

IV.2.20 dirname

- defining class: **PEL_SystemExp**

IV.2.21 double

- defining class: **PEL_ConvertTypeExp**

IV.2.22 double_equality

determine if floating quantities are close enough

- type: **PEL_Data::Bool**
- 4 arguments: **PEL_Data::Double** (denoted x), **PEL_Data::Double** (denoted y), **PEL_Data::Double** (denoted $x_{\min}$), **PEL_Data::Double** (denoted ε)
- evalution: the return value is given by **PEL::double_equality**
  (1) $x_{\min}$ represents the lower bound under which x or y are undistinguishable from 0
  (2) if both x and y are undistinguishable from 0, they are close enough
  (3) if both x and y are distinguishable from 0, they are close enough provided that $|x/y - 1| < \varepsilon$ (overflow and underflow when evaluating x/y are handled)
  (4) if one of x or y is undistinguishable from 0, the other is compared to $x_{\min}$ as in (3)
- examples:

expression	evaluation result
`double_equality( 1.0, 1.0, 0.0, 0.0 )`	`true`
`double_equality( 1.e-12, 1.e-14, 1.e-3, 1.e-10 )`	`true`
`double_equality( 1.0, 1.0001, 1.e-3, 1.e-10 )`	`true`
`double_equality( 1.0, 1.01, 1.e-3, 1.e-10 )`	`false`
`double_equality( 0.9999e-10, 1.0001e-10, 1.e-3, 1.e-10 )`	`true`
`double_equality( 1.0e+300, 1.0e-200, 1.e-3, 1.e-300 )`	`false`
`double_equality( 1.0e-200, 1.0e+300, 1.e-3, 1.e-300 )`	`false`

- defining class: **PEL_MathFunctionExp**

IV.2.23 dnum

- defining class: **PEL_DerivativeExp**

IV.2.24 Ei

- defining class: **PEL_MathFunctionExp**

IV.2.25 En

- defining class: **PEL_MathFunctionExp**

IV.2.26 e

- defining class: **PEL_ConstantExp**

IV.2.27 empty

- examples:

expression	evaluation result
`empty( "" )`	`true`
`empty( "toto" )`	`false`

- defining class: **PEL_StringExp**

IV.2.28 erf

- defining class: **PEL_MathFunctionExp**

IV.2.29 erfc

- defining class: **PEL_MathFunctionExp**

IV.2.30 euler

- defining class: **PEL_ConstantExp**

IV.2.31 exp

base-e exponential function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double** (denoted x)
- evaluation: the return value is e^x, the base-e exponential of the argument
- defining class: **PEL_MathFunctionExp**

IV.2.32 extracted_data

- defining class: **PEL_ExtractionExp**

IV.2.33 extracted_module

- defining class: **PEL_ExtractionExp**

IV.2.34 floor

floor function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double**
- evaluation: the return value is the largest integral value greater than of equal to the argument
- examples:

expression	evaluation result
`floor( 7.0 )`	`7.0`
`floor( 7.1 )`	`7.0`
`floor( 7.9 )`	`7.0`
`floor( -12.0 )`	`-12.0`
`floor( -12.2 )`	`-13.0`
`floor( -12.8 )`	`-13.0`

- defining class: **PEL_MathFunctionExp**

IV.2.35 gamma

- defining class: **PEL_MathFunctionExp**

IV.2.36 geometric_sequence

- defining class: **PEL_SigalExp**

IV.2.37 getcwd

working directory pathname

- defining class: **PEL_SystemExp**

IV.2.38 getenv

- defining class: **PEL_SystemExp**

IV.2.39 getpid

- defining class: **PEL_SystemExp**

IV.2.40 `greater`

boolean expression that asks if all the items of a vector are greater than or equal to a given value
expression booléenne testant si tous les éléments d'un tableau sont supérieurs ou égaux à une valeur donnée

- "Int" case
  - type: **`PEL_Data::Bool`**
  - 2 arguments: **`PEL_Data::IntVector`** ; **`PEL_Data::Int`**
- "Double" case
  - type: **`PEL_Data::Bool`**
  - 2 arguments: **`PEL_Data::DoubleVector`** ; **`PEL_Data::Double`**
- evaluation: the return value is **`true`** if all the elememts of the first argument are greater than or equal to the second argument, **`false`** otherwise
- examples:

expression	evaluation result
`greater( < -10 3 >, 1 )`	`false`
`greater( < 1.0 3.0 >, 0.5 )`	`true`
`greater( < 0 1 1 >, 0.5 )`	`true`

- defining class: **`PEL_VectorExp`**

IV.2.41 `has_data`

- defining class: **`PEL_ExtractionExp`**

IV.2.42 `has_module`

- defining class: **`PEL_ExtractionExp`**

IV.2.43 `host_name`

- defining class: **`PEL_SystemExp`**

IV.2.44 `in_box`

- examples:

expression	evaluation result
`in_box( <0. 0.>, <-1. -1.>,<1. 1.>)`	`true`
`in_box( < -3.255 1.3 >, < -3.256 -5.0 >, < 0.0 1.32 > )`	`true`
`in_box( < -3.257 1.3 >, < -3.256 -5.0 >, < 0.0 1.32 > )`	`false`
`$DV_X = < 15.3 -3. 1.3 >` `in_box( $DV_X, < 12.5 -3.2 -5.0 >, < 28. 0. 1.32 > )`	`true`

- defining class: **`PEL_MembershipExp`**

IV.2.45 `incomplete_gamma`

- defining class: **`PEL_MathFunctionExp`**

IV.2.46 `increasing`

boolean expression that asks whether all the items of a vector all sorted by increasing values

- ”Int” case
  - type: **`PEL_Data::Bool`**
  - 1 argument: **`PEL_Data::IntVector`**
- cas ”Double”
  - type: **`PEL_Data::Bool`**
  - 1 argument: **`PEL_Data::DoubleVector`**
- evalution: the return value is **`true`** if each item of the argument is greater than or equal to the immediately preceeding one, **`false`** otherwise
- examples:

expression	evaluation result
`increasing( < 0.0 1.0 1.0 2.0 > )`	`true`
`increasing( < 0 1 2 1 > )`	`false`

- defining class: **`PEL_VectorExp`**

IV.2.47 `in_range`

- examples:

expression	evaluation result
`in_range( 1.0, < 3.0 3000.0 > )`	`false`
`in_range( 1.0, < 1.0 2.0 > )`	`true`
`in_range( 3, < 1 2 > )`	`false`
`in_range( 1, < 1 2 > )`	`true`

- defining class: **`PEL_MembershipExp`**

IV.2.48 `int`

- defining class: **`PEL_ConvertTypeExp`**

IV.2.49 `interpol`

- defining class: **`PEL_InterpolExp`**

IV.2.50 `is_defined`

- defining class: **`PEL_VariableExp`**

IV.2.51 **join**

- defining class: **PEL_SystemExp**

IV.2.52 **j0**

- defining class: **PEL_MathFunctionExp**

IV.2.53 **j1**

- defining class: **PEL_MathFunctionExp**

IV.2.54 **jn**

- defining class: **PEL_MathFunctionExp**

IV.2.55 **lgamma**

- defining class: **PEL_MathFunctionExp**

IV.2.56 **log**

natural (base-e) logarithm function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double**
- evaluation: the return value is the natural logarithm of the argument
- defining class: **PEL_MathFunctionExp**

IV.2.57 **log10**

base-10 logarithm function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double**
- evaluation: the return value is the base-10 logarithm of the argument
- defining class: **PEL_MathFunctionExp**

IV.2.58 **max**

- defining class: **PEL_ComparisonExp**

IV.2.59 **middle_point**

- type: **PEL_Data::Double**
- 2 arguments: **PEL_Data::Double** (denoted x) ; **PEL_Data::DoubleVector** (denoted $\mathbf{v}$)
- evalution: the return value is the middle point of the interval defined by two successive elements of $\mathbf{v}$ containing x.

- examples:

expression	evaluation result
`middle_point( 1.2, < 1. 2. 6. > )`	`1.5`
`middle_point( 1.9, < 1. 2. 6. > )`	`1.5`
`middle_point( 5.9, < 1. 2. 6. > )`	`4.`

- defining class: **`PEL_CutPointsExp`**

IV.2.60 `middle_points`

- type: **`PEL_Data::DoubleVector`**
- 1 or 2 argument(s): optional **`PEL_Data::DoubleVector`** (denoted **x**) ; **`PEL_Data::DoubleVector`** (denoted **v**)
- evalution: the return value is the table of all the middle points of the intervals defined by two successive elements of **v** if **x** is not specified, or only the middle points of the intervals containing the elements of **x**.
- examples:

expression	evaluation result
`middle_points( < 1. 2. 6. > )`	`< 1.5 4. >`
`middle_points( < 1.1 1.9 2. >, < 1. 2. 6. > )`	`<1.5 1.5 4. >`

- defining class: **`PEL_CutPointsExp`**

IV.2.61 `min`

- defining class: **`PEL_ComparisonExp`**

IV.2.62 `modulo`

remainder of the euclidian division

- type: **`PEL_Data::Int`**
- 2 arguments: **`PEL_Data::Int`** ; **`PEL_Data::Int`**
- examples:

expression	evaluation result
`modulo( 3, 2 )`	`1`
`modulo( 2, 3 )`	`0`

- defining class: **`PEL_ArithmeticExp`**

IV.2.63 `nvector`

vector built on basis of its dimension and an initializing scalar value

- type: vector type corresponding to the given scalar value
- 2 arguments: **`PEL_Data::Int`** ; scalar type
- examples:

expression	evaluation result
`nvector( 3, 1.0 )`	`< 1.0 1.0 1.0 >`
`nvector( 1, 2 )`	`< 2 >`
`nvector( 2, true )`	`< true true >`
`nvector( 2, "to" )`	`< "to" "to" >`

- defining class: **`PEL_VectorExp`**

IV.2.64 `nb_ranks`

- defining class: **`PEL_CommunicatorExp`**

IV.2.65 `path_name_separator`

- type: **`PEL_Data::String`**
- no argument
- evalution:

expression	evaluation result
`path_name_separator()`	UNIX systems : `"/"` windows systems : `"\"`

- defining class: **`PEL_SystemExp`**

IV.2.66 `perturbated_coordinates`

- type: **`PEL_Data::DoubleVector`**
- 4 arguments: **`PEL_Data::Double`** (denoted ε) ; **`PEL_Data::DoubleVector`** (denoted $\mathbf{r}$) ; **`PEL_Data::Double`** (denoted δ_h) ; **`PEL_Data::Bool`** (denoted b)
- evalution: the return value is a vector $\widetilde{\mathbf{r}}$ such that:
  - $\widetilde{\mathbf{r}}$ has the same size as $\mathbf{r}$
  - if b is **`true`**: $\widetilde{\mathbf{r}} = \mathbf{r}$
  - if b is **`false`**: $\widetilde{\mathbf{r}}$ is obtained by adding to $\mathbf{r}$ a displacement of length $\varepsilon \cdot \delta_h$ in a pseudo-random direction (which is computed in a sequence of pseudo-random numbers)
- defining class: **`GE_PerturbatedMeshingExp`**

IV.2.67 `pi`

- defining class: **`PEL_ConstantExp`**

IV.2.68 `pow`

- defining class: **`PEL_MathFunctionExp`**

IV.2.69 **rand**

- type: **PEL_Data::Int**
- no argument
- evaluation: the return value is a positive random int value
- examples:

expression	evaluation result
rand()	**1071501796**
rand()	**622778817**
rand()	**1226502026**

- defining class: **PEL_MathFunctionExp**

IV.2.70 **random_double**

- type: **PEL_Data::Double**
- no argument
- evaluation: the return value is a random double value greater than 0. and lower than 1.
- examples:

expression	evaluation result
random_double()	**6.212538e-01**
random_double()	**6.596979e-01**
random_double()	**4.799387e-01**

- defining class: **PEL_MathFunctionExp**

IV.2.71 **rank**

- defining class: **PEL_CommunicatorExp**

IV.2.72 **regular_vector**

vector with a constant difference between two successive items

- ”Int” case
  - type: **PEL_Data::IntVector**
  - 3 arguments: **PEL_Data::Int** (denoted x); **PEL_Data::Int** (denoted n); **PEL_Data::Int** (denoted y)
- ”Double” case
  - type: **PEL_Data::DoubleVector**
  - 3 arguments: **PEL_Data::Double** (denoted x); **PEL_Data::Int** (denoted n); **PEL_Data::Double** (denoted y)
- evalution: the return value is a vector whose first (resp. last) item is x (resp. y), and such that n represents the number of intervals between x and y, each of these intervals having the same length (hence the number of elements of the returned vector is $n + 1$)
- examples:

expression	evaluation result
`regular_vector( 1.0, 4, 5.0 )`	`< 1.0 2.0 3.0 4.0 5.0 >`
`regular_vector( 16.0, 4, 22.0 )`	`< 16.0 17.5 19.0 20.5 22.0 >`
`regular_vector( 1, 4, 5 )`	`< 1 2 3 4 5 >`
`regular_vector( -100, 5, -80 )`	`< -100 -96 -92 -88 -84 -80 >`

- defining class: **PEL_SigalExp**

IV.2.73 reverse

- defining class: **PEL_VectorExp**

IV.2.74 segm_sort

- defining class: **PEL_GroupExp**

IV.2.75 segm2D_sort

- defining class: **PEL_GroupExp**

IV.2.76 segm3D_sort

- defining class: **PEL_GroupExp**

IV.2.77 sin

sine function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double**
- evaluation: the return value is the sine of the argument (measured in radians)
- defining class: **PEL_MathFunctionExp**

IV.2.78 sinh

- defining class: **PEL_MathFunctionExp**

IV.2.79 size

- defining class: **PEL_VectorExp**

IV.2.80 sort

- defining class: **PEL_SortExp**

IV.2.81 **sqr**

square function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double**
- evaluation: the return value is the square of the argument
- defining class: **PEL_MathFunctionExp**

IV.2.82 **sqrt**

square root function

- type: **PEL_Data::Double**
- 1 argument: **PEL_Data::Double**
- evaluation: the return value is the non-negative square root of the argument
- defining class: **PEL_MathFunctionExp**

IV.2.83 **stretched_vector**

- example:

expression	evaluation result
stretched_vector(1.0, 1.0, 8.0, 16.0)	**< 1.0 2.0 4.0 8.0 16.0 >**

- defining class: **PEL_SigalExp**

IV.2.84 **sum**

- defining class: **PEL_VectorExp**

IV.2.85 **tan**

- defining class: **PEL_MathFunctionExp**

IV.2.86 **tanh**

- defining class: **PEL_MathFunctionExp**

IV.2.87 **this_file_dir**

pathname of the file containing this expression

- defining class:

IV.2.88 **to_string**

- defining class: **PEL_StringExp**

IV.2.89 `uname`

- defining class: **`PEL_SystemExp`**

IV.2.90 `unit_sort`

- defining class: **`PEL_GroupExp`**

IV.2.91 `value`

- defining class: **`PEL_VariableExp`**

IV.2.92 `vector`

- defining class: **`PEL_VectorExp`**

IV.2.93 `x_cut_points`

- type: **`PEL_Data::DoubleArray2D`**
- 2 arguments: **`PEL_Data::DoubleVector`** (denoted $\mathbf{x}$) ; **`PEL_Data::Double`** (denoted y) and an optional 3rd argument: **`PEL_Data::Double`** (denoted z)
- evalution: the return value is a vector of 2-items vectors (if 2 arguments) or 3-items vectors (if 3 arguments) where the first items of these vectors are the successive midpoints of the elements of $\mathbf{x}$ and where the last items of these vectors are all y (if 2 arguments) or y, z (if 3 arguments).
- examples:

expression	evaluation result
`x_cut_points( < 1. 2. 6. >, -1. )`	`array( < 1.5 -1. > ,` `       < 4.  -1. > )`
`x_cut_points( < 1. 2. 6. >, -1., 0. )`	`array( < 1.5 -1. 0. > ,` `       < 4.  -1. 0. > )`

- defining class: **`PEL_CutPointsExp`**

IV.2.94 `y_cut_points`

- type: **`PEL_Data::DoubleArray2D`**
- 2 arguments: **`PEL_Data::Double`** (denoted x) ; **`PEL_Data::DoubleVector`** (denoted $\mathbf{y}$) and an optional 3rd argument: **`PEL_Data::Double`** (denoted z)
- evalution: the return value is a vector of 2-items vectors (if 2 arguments) or 3-items vectors (if 3 arguments) where the first (resp. third when 3 arguments) items of these vectors are all x (resp. z when 3 arguments) and where the second items of these vectors are the successive midpoints of the elements of $\mathbf{y}$.
- examples:

expression	evaluation result
`y_cut_points( -1., < 1. 2. 6. > )`	`array( < -1. 1.5 > ,` `< -1. 4. > )`
`y_cut_points( -1., < 1. 2. 6. >, 0. )`	`array( < -1. 1.5 0. > ,` `< -1. 4. 0. > )`

- defining class: **PEL_CutPointsExp**

IV.2.95 y0

- defining class: **PEL_MathFunctionExp**

IV.2.96 y1

- defining class: **PEL_MathFunctionExp**

IV.2.97 yn

- defining class: **PEL_MathFunctionExp**

IV.2.98 z_cut_points

- type: **PEL_Data::DoubleArray2D**
- 3 arguments: **PEL_Data::Double** (denoted x) ; **PEL_Data::Double** (denoted y) ; **PEL_Data::DoubleVector** (denoted $\mathbf{z}$)
- evalution: the return value is a vector of 3-items vectors where the first two items of these vectors are all x, y and where the last items of these vectors are the successive midpoints of the elements of $\mathbf{z}$.
- example:

expression	evaluation result
`z_cut_points( -1., 0., < 1. 2. 6. > )`	`array( < -1. 0. 1.5 > ,` `< -1. 0. 4. > )`

- defining class: **PEL_CutPointsExp**

IV.2.99 «

- defining class:

IV.2.100 (?:)

- defining class: **PEL_ConditionalExp**

IV.2.101 < <= > >= =

- defining class: **PEL_RelationalExp**

IV.2.102 `+`

- defining class: **`PEL_ArithmeticExp`**

IV.2.103 `-`

- defining class: **`PEL_ArithmeticExp`**

IV.2.104 `/`

- defining class: **`PEL_ArithmeticExp`**

IV.2.105 `*`

- defining class: **`PEL_ArithmeticExp`**

IV.2.106 `&&`

- defining class: **`PEL_BooleanExp`**

IV.2.107 `||`

- defining class: **`PEL_BooleanExp`**

IV.2.108 `!`

- defining class: **`PEL_BooleanExp`**

Chapter V

Administration of PELICANS-based Applications

A PELICANS-based application is *final user* software that has been implemented using the functionalities provided by PELICANS (§III.6).

A **user** of PELICANS is a **software developper** that **implements** its software using PELICANS as a toolkit. For this reasons, he is compelled with the standard administration work including the common tasks of compilation (through *ad-hoc* makefiles), documentation, recording of non regression tests *etc...*

Without being unecessarily prescriptive, PELICANS provides a set of tools commonly called **`pel`**, accessible through the command line, that facilitate the accomplishment of these tasks. A particular attention has been drawn toward contexts such that:

- sources are spread into various directories;
- these directories are stored on disks shared by multiple computers with possibly different hardware and operating system;
- multiple compilers are used on each of these computers.

Invocation of the **`pel`** commands may be wrapped into an *administration makefile*.

V.1 Setting Up the Environment

The very first point is to discern which *installation of PELICANS* will be used.

An *installation of PELICANS* is simply identified by the name of the *installation directory*, for instance:

`/usr/local/pelicans.08_04_2007`

(we will use that generic name hereafter). A necessary condition for the validity of that installation is the occurence of two files with basename **`libpel0`** and **`libpel1`** in one or more subdirectories of **`/usr/local/pelicans.08_04_2007/lib`**. These files are *dynamic libraries* of PELICANS. Their extension depends on the *compiler architecture*, it is typically **`.so`** for LINUX and **`.dylib`** on MacOS.

If such is not the case, refer to the PELICANS installation guide [1].

Access to the **`pel`** commands requires a preliminary setup of environment variables (essentially **`PATH`** and **`PELICANSHOME`**). This has to be done manually, in a way that depends on the current shell. This latter can be determined by running the command:

```
echo $SHELL
```

If it says "**csh**" of "**tcsh**" in it, the current shell is the C shell. If it is "**bash**", "**zsh**", "**ksh**", "**sh**" or something similar, the current shell is likely a variant of the Bourne Shell.

C Shell – `init.csh`

- The **init.csh** file is devoted to the C Shell.
- To use the installation of PELICANS located in the **/usr/local/pelicans.08_04_2007** directory, enter the command:

  ```
  source /usr/local/pelicans.08_04_2007/bin/init.csh
  ```
- The above line may be added to a startup file like like **.cshrc** or **.tcshrc** in the home directory.

Bourne shell – `init.sh`

- The **init.sh** file is devoted to the Bourne Shell.
- To use the installation of PELICANS located in the **/usr/local/pelicans.08_04_2007** directory, enter the command:

  ```
  . /usr/local/pelicans.08_04_2007/bin/init.sh
  ```
- The above line may be added to a startup file like **.profile** or **.bash_profile** in the home directory.

The files **init.csh** and **init.sh** set environment variables like **PATH** and **MANPATH**. Hence, they are not supposed to be run like normal commands. To have a lasting effect on the shell, they must be processed with the **source** command for C Shells, or with the **.** command for Bourne Shells.

Note that the modification of startup files is not recommended since it may lead to seemingly disturbing behaviors. Indeed, a shell begins by executing in sequence commands from several system files, among which is the startup file that may have been (unjudiciously) modified for PELICANS usage. But the other system files that get run after this latter may on turn overwrite setups previously done by the PELICANS **init** commands. Another difficulty is that login and non-login shells begin differently.

But the most important reason to preclude the sourcing of PELICANS **init** commands within a shell startup file is that one should be perfectly aware of which version of PELICANS is used. Indeed, the development process of PELICANS strongly relies on *continuous delivery* with *incremental modifications*, so several versions of PELICANS are likely to be installed side by side progressively as they are delivered. Older version are kept close to the most recent one for some time, and are discarded only when it is completely secure from the point of view of the PELICANS-based applications. Hence, it is essential to know what version of PELICANS is used and to be able to switch one with the other. The recommended attitude is then to set up the environment for PELICANS explicitly, voluntarily and only when necessary.

V.2 Compilation Level and Architecture

V.2.1 Customization of the Compilation Process

PELICANS-based applications are written in human readable programming languages, **C++98** for the PELICANS building block. This form is called the *source code*. Before a computer can actually run them, they must be transformed into low level machine code instructions (unreadable by most humans), a form that is called *machine code*. That process is called *compiling*, or, more generally, *building* because it involves more steps than just compiling (§I.2.3).

A *compiler* is a *program* that translates *source code* into *machine code*.

The **C++98** compiler that has been used when installing PELICANS [1] will in turn be used for the compilation of PELICANS-based applications (at least for their **C++** part which interests us here).

To some extent, the behavior of the *compiler* can be customized. PELICANS defines the *compilation levels* to provide an easy accessibility to the balance between built-in testing and computational efficiency, two antonymous requirements.

The role of *compilation levels* is double.

1. Enabling/disabling of built-in assertions.

   The object oriented methodology for software development with PELICANS strongly relies on built-in assertions: *preconditions*, *postconditions*, *invariants* and *checks* which are implemented using respectively the four macros:
   PEL_CHECK_PRE, **PEL_CHECK_POST**, **PEL_CHECK_INV** and **PEL_CHECK** .
   These assertions can be disabled/enabled at compilation time (§III.7).

2. Select compiler options.

   A *compiler* is a program, and as such, takes some parameters as input that govern the way the processing is done. Compiler options are parameters passed to the compiler (in a way, input to the compiler) to advise/instruct the compiler as to how it should compile the PELICANS-based application. There are various compiler options, all of them being dependent upon:
   - a particular *machine* (*ie* hardware environment together with its operating system);
   - a particular *compiler*.

   These two specificities determine the *compiler architecture* [1]. For each *compiler architecture* defined in an *installation of PELICANS*, sets of compiler options have been attached to the various *compilation levels* (within specific files called *architecture-makefiles* [1]).

Hence, compilation of PELICANS-based applications must be understood in the context of a given *compiler architecture* defined in an *installation of PELICANS*.

Moreover, for each compilation process, a particular customization of the *compiler* behavior **must** be choosen. This choice is performed by selecting a *compilation level* with the conscious aim of promoting a particular characteristic of the generated *machine code* (*eg* run-time efficient, ready to run in debugging or profiling mode).

V.2.2 Compiler Architectures

Within an *installation of PELICANS*, *compilers* are identified by a name. The symbol **<compiler>** will be used hereafter as a placeholder for that name (the default being always **gcc**).

Compiler architectures facilitate working in environments that rely on disk sharing between multiple computers with possibly different hardware and operating systems (*ie* multiple *machines*). They are widely described in [1], but there is no need for users of PELICANS to understand the details of that notion. Any *installation of PELICANS* handles a given set of *compilers* [1] and provides the associated *compiler architectures* which define all relevant option settings for their specific *compiler* and *machine*.

The *compiler architecture* associated to a given *compiler* on the current *machine* can be identified by running the command:

```
pel arch <compiler> ; echo
```

It should produce a sequence of non-blank characters (*eg* **Linux-gcc**, **SunOS-CC**, **Darwin-gcc**) that will subsequently be used by the **pel** utilities to isolate compilation and testing results in particularized directories.

V.2.3 The Compilation Level of the PELICANS libraries

A compiled version of the PELICANS platform presents itself in the form of two *dynamic libraries* stored in two files with basename **libpel0** and **libpel1** (and simply called **libpel0** and **libpel1**), each of them associated with a particular *compilation level*.

- **libpel1** : The *preconditions* (in **PEL_CHECK_PRE** macros) appearing within the classes of PELICANS are executed. Oppositely, the *postconditions* (in **PEL_CHECK_POST** macros), *invariants* (in **PEL_CHECK_INV** macros) and *checks* (in **PEL_CHECK** macros) are all ignored.
- **libpel0** : The *preconditions* (in **PEL_CHECK_PRE** macros), *postconditions* (in **PEL_CHECK_POST** macros), *invariants* (in **PEL_CHECK_INV** macros) and *checks* (in **PEL_CHECK** macros) appearing within the classes of PELICANS are all ignored.

The compiler options used to produce **libpel1** and **libpel0** are identical and designed to promote an optimal run-time efficiency on the current machine.

The **libpel1** library is adapted for the **construction** and **testing** phase of PELICANS-based applications whereas the **libpel0** library should be reserved to the **production** phase of PELICANS-based applications.

V.2.4 Compilation Level for PELICANS-based Application

The *compilation level* used to compile the source files of a PELICANS-based application is **distinct** from that of the PELICANS library used by the linker.

There are 4 essential *compilation levels*, denoted respectively: **opt0**, **opt1**, **opt2**, **dbg**.

- **opt0**
  - The compiler options are designed to promote an optimal run-time efficiency on the current machine.
  - The *preconditions* (in **PEL_CHECK_PRE** macros), *postconditions* (in **PEL_CHECK_POST** macros), *invariants* (in **PEL_CHECK_INV** macros) and *checks* (in **PEL_CHECK** macros) are all ignored.
- **opt1**
  - The compiler options are the same as for **opt0**.
  - The *preconditions* (in **PEL_CHECK_PRE** macros) are executed.
  - Oppositely, the *postconditions* (in **PEL_CHECK_POST** macros), *invariants* (in **PEL_CHECK_INV** macros) and *checks* (in **PEL_CHECK** macros) are all ignored.
- **opt2**
  - The compiler options are the same as for **opt0**.
  - The *preconditions* (in **PEL_CHECK_PRE** macros) are executed.
  - Moreover, the *postconditions* (in **PEL_CHECK_POST** macros) may be executed provided that the command line options **-Cpost** or **-Call** are set.
  - Similarly, the *invariants* (in **PEL_CHECK_INV** macros) and *checks* (in **PEL_CHECK** macros) may be executed, provided that the command line option **-Call** is set.
- **dbg**
  - The compiler options are such that running in debug mode is possible.
  - The conditions for executing *preconditions*, *postconditions*, *invariants* and *checks* are the same as for **opt2**.

As can be seen, the library **libpel0** (resp. **libpel1**) has been produced with the **opt0** (resp. **opt1**) *compilation level*.

Note that the assertion enabling hierarchy associated to the *compilation levels* had already been presented during the design by contract exposition (§III.7.7).

V.3 Sensible Directory Layout and Administration Makefile

A PELICANS-based application is a numerical simulation software composed not only of a set of source and header files (§II.2.2) but also of administration tools, executable files, online documentation, non regression tests *etc...*

Developing then administrating such an application generally starts from an already existing application which is progressively modified (even if only a negligible part of the original source may be reused, the overall organization and the administration tools stay mainly relevant).

V.3.1 A First PELICANS-based Applications

For any application appearing in the "Examples of Applications" page of the documentation, it is possible to get a complete working environment that may serve as a starting point for future developments. Let us consider for example the application called **`Tutorial`**. The link "Get Source" gives a simple way to download a standard compressed unix archive file called **`Tutorial.tgz`**.

The following command extracts the archive in the directory where it was put (say **`/home/martin`**):

```
/home/martin 1>  tar -zxvf Tutorial.tgz
```

That creates a directory with the same name as the archive, organized as follows:

```
/home/martin/Tutorial
                  ├──── src
                  ├──── RegressionTests
                  └.......................... Makefile
```

The two directories **`src`**, **`RegressionTests`** and the file **`Makefile`** are described below.

`src`

- This directory contains the source and header files.

`Makefile`

- This file specifies all the necessary instructions for the administration tasks of a PELICANS-based application. It is called an *administration makefile* and requires GNU **`make`** (version 3.77 or newer) to be executed. It defines various targets devoted to:
  - building of the executables;
  - execution of non regression tests;
  - automatic generation of the online documentation;
  - cleaning files that may be regenerated.

  Most of the commands that are wrapped into this makefile are based on the **`pel`** utility and could be run straight from the command line.
- Using the *administration makefile* requires that the environment be set up previously (§V.1).
- Once the environment has been set up, the following command, where **`make`** denotes GNU make (sometimes called **`gmake`**):

  ```
  /home/martin/Tutorial 2>  make
  ```

  provides the list of the available targets with a short description of the associated actions. For example, the command:

  ```
  /home/martin/Tutorial 3>  make exe2
  ```

compiles all the source files stored in subdirectories of **/home/martin/Tutorial** (in the present case, all of them are located in **src**) with the **opt2** *compilation level* and builds the executable **exe2** by linking with **libpel1** (itself with the **opt1** *compilation level*). The **exe2** file is located in the directory (newly created):

/home/martin/Tutorial/lib/<arch>/opt2

where **<arch>** is a placeholder for the name of the current *compiler architecture* (*ie* the result of the **pel arch** command).

- Similarly, the command:

  /home/martin/Tutorial 4> make exe0

  creates, in the same directory, the executable **exe0** compiled with the **opt0** *compilation level* and linked with **libpel0**.

- The reference documentation, with the HTML format, is automatically generated on basis of informations extracted from the source and header files, by running the command:

  /home/martin/Tutorial 5> make doc

  On completion, the index file of that documentation, called:

  /home/martin/Tutorial/doc/Html/index.html

  may be opened with any standard HTML browser.

- The non regression tests are executed by the command:

  /home/martin/Tutorial 6> make test

  On completion, report files are produced in the directory (newly created):

  /home/martin/Tutorial/lib/tests/<arch>

  where **<arch>** is a placeholder for the name of the current *compiler architecture* (*ie* the result of the **pel arch** command). These tests are related to the **RegressionTests** directory described below. Note that the **test** target also generates the reference documentation (*via* the **doc** target, see above).

- Finally, the command:

  /home/martin/Tutorial 7> make clean

  deletes all the files and directories created during the compilation stage, the documentation stage and the testing stage (in fact, everything that is generated when calling a target of the administration makefile).

RegressionTests

- This directory contains three subdirectories: **fem**, **fvm** and **greeting**, each of them representing a reference run of the considered application.
- As example, let us describe briefly the **fem** subdirectory.

data.pel	file storing the *hierarchical data structure* of the data deck
save.pel	result file with the PELICANS *hierarchical data structure* format
save.gene	result file with the TIC format
visu	directory containing the TIC command file **t.tic** for a postprocessing of **save.gene** with the TIC utility, by running the command: **tic < visu/t.tic** in the **fem** directory.
config.pel	file storing instructions for the **pel test** utility which is called when the command **make test** is executed.

V.3.2 Guidelines

The file and directory organisation of the **`Tutorial`** application can be reused and adapted for real life applications.

Administration Makefile

The *administration makefile* of the **`Tutorial`** application can be reused as such for real life applications. It is a straight copy of the following file:

`$PELICANSHOME/etc/makefile_for_appli`

where **`$PELICANSHOME`** denotes the installation directory for the current version of PELICANS.

To stay durably up-to-date with the evolutions of PELICANS, it is preferable to invoke directly **`$PELICANSHOME/etc/makefile_for_appli`** instead of one of its copy bound to become rapidly outdated.

The simple script **`pmake`**, downloadable from the *utils* package next to the *pelicans* distribution package, simplifies this direct invocation. Its main features are depicted below:

```
1  #!/bin/sh
2  application_name=PROMISE
3  if [ -z $PELICANSHOME ]
4  then
5     echo "Undefined variable PELICANSHOME"
6     exit 1
7  fi
8  makefile=$PELICANSHOME/etc/Makefile_for_appli
9  cd ‘dirname $0‘
10 make -f $makefile APPLINAME=$application_name CALLER=$0 $*
```

The invocation of **`makefile_for_appli`** occurs at line 10. The word PROMISE should be replaced at line 2 (without introducing any blank character around the **`=`** sign) by the name of the current application (this name will appear in the generated documentation).

As soon as the file **`pmake`** is placed in the root directory of the current PELICANS-based application, invoking the administration makefile is performed by replacing **`make`** with **`pmake`** in the commands of §V.3.1.

Setting Up the Environment

Of course, the *administration makefile* can be invoked only after the PELICANS environment has been set up (hence the lines 3-7 in the above depiction of **`pmake`**).

Setting up the evironment can be performed as explained in §V.1. Nevertheless, the processing of **`init.sh`**, or **`init.csh`**, can fruitfully be encapsulated into the command files **`my_env.sh`**, or **`my_env.csh`**, downloadable from the *utils* package next to the *pelicans* distribution package. The main features of these two files are depicted below:

`my_env.sh`

```
pel_install_dir=/usr/local/pelicans.08_04_2007
local_arch_dir=/usr/local/pelicans/local_arch
. $pel_install_dir/bin/init.sh
export PELARCHDIR=$local_arch_dir
export EXE0=‘pwd‘/lib/‘pel arch‘/opt0/exe
export EXE2=‘pwd‘/lib/‘pel arch‘/opt2/exe
export EXEG=‘pwd‘/lib/‘pel arch‘/optg/exe
```

my_env.csh

```
1 set pel_install_dir = /usr/local/pelicans.08_04_2007
2 set local_arch_dir = /usr/local/pelicans/local_arch
3 source $pel_install_dir/bin/init.csh
4 setenv PELARCHDIR $local_arch_dir
5 setenv EXE0  ‘pwd‘/lib/‘pel arch‘/opt0/exe
6 setenv EXE2  ‘pwd‘$CW_DIR/lib/‘pel arch‘/opt2/exe
7 setenv EXEG  ‘pwd‘$CW_DIR/lib/‘pel arch‘/optg/exe
```

The desired PELICANS *installation directory* is notified at line 1.

Compiler architectures may have been created during the installation of PELICANS to adapt it to particular *machines* or *compilers* [1]. The related files are stored in a directory whose name should be assigned to the **PELARCHDIR** environment variable in order to be taken into account when running **pel arch**. This justifies lines 2 and 3.

The *compiler architectures* are used by the *administration makefile* to isolate compilation results in particularized directories whose name are the result of the **pel arch** command. To facilitate the access to the executables, their full path is assigned to environment variables at lines 4-7.

As soon as the customized files **make.sh** and **make.csh** are placed in the root directory of the current PELICANS-based application, they can be used to set up the environment by running the Bourne shell command:

```
. my_env.sh
```

or the C Shell command:

```
source my_env.csh
```

Source and Header Files

Starting from the application root directory, the source files of the **Tutorial** application were all located in the **./src** subdirectory.

In real life applications, it may be relevant to organize the source and header files in thematic packages, with one directory per package, and to separate, in these directories, header files, stored in subdirectory called **include**, from source files, stored in a subdirectory called **src**.

From the point of view of the *administration makefile* invoked by **pmake**, any header file or source file found in a subdirectory of the application root directory is consider to be part of the application (header files are those having a **.h** or **.hh** extension whereas source files are those having a **.cpp**, **.cc**, **.c**, **.F** or **.f** extension).

Regression Testing

Regression testing seeks to uncover *regression faults*.

A *regression fault* occurs whenever a functionality that previously worked as desired stops working or no longer works the same way that was previously planned.

The PELICANS method for *regression testing* is to extensively, repeatedly and automatically re-run previously running tests. These previously running tests are organized as a sequence of directories such that each directory

- corresponds to a single test;
- contains everything necessary to run the test (*eg* data files);
- contains the desired results (in specialized files).

That sequence of directories is stored by the **Tutorial** application in the **RegressionTest** directory.

Keeping this organization makes it possible to use the **`test`** target of the *administration makefile* in order to perform the *regression testing*. The command of this target delegates its processing to the very important **`pel test`** utility (§V.4.4). Note that this latter is often used directly, for instance during the implementation phase, to re-run only one, or a few regression tests that may be affected by a particular development.

V.4 Command Line Interface

The **`pel`** commands are a collection of utilities devoted to the management of PELICANS-based applications.

The "Command Line Interface" page of the PELICANS documentation describes in details the **`pel`** commands. The same information is directly available on the command line by running the command:

`pel -man`

or, for one of the **`pel`** commands, *eg* **`run`**:

`pel run -man`

The foregoing only provides a partial description. The focus is on the commands actually used in the *administration makefile* of §V.3.1, namely:

`pel depend`	generation of makefiles
`pel build`	generation of executables, objects and libraries
`pel run`	execution of a PELICANS-based application
`pel test`	comparison between runs (for regression testing)
`pel predoc`	possible preparation before using **`peldoc`**

V.4.1 `pel depend`

Description

The task of compiling a PELICANS-based application is highly simplified by using the GNU **`make`** utility. The aim of **`pel depend`** is to write a suitable makefile that describes the relationships among files in the considered application and provides commands for compiling each file and linking with the appropriate PELICANS library. Once this makefile exists, **`pel build`** can be used to build the desired executable file.

`pel depend` has been specially designed to handle sources located in multiple directories, and to handle multiple *compilers* on the same file system. Moreover the generated makefile includes special targets and commands to determine or re-determine the dependencies between the files if necessary.

`pel depend` together with **`pel run`** have some options and arguments specially devoted to handling *compilation levels*.

Synopsis

`pel depend -man`

`pel depend` *[options...] arguments...*

`pel depend` *[* **`-l`** *lib|dir] opt bindir sources*

Arguments

- *opt*
  Decide the compilation level that will be set in the generated makefile. This option influences on the one hand the optimization used by the *compiler* when generating the binary code and on the other hand the assertions that will be evaluated. Allowed values for *opt* are: **dbg**, **opt2**, **opt1**, **opt0**, **optpg** or **optcov**.
  - **dbg** The commands of the generated makefile will ask the compiling system to generate a binary code prepared for debugging.
  - **opt2**,**opt1**,**opt0** The commands of the generated makefile will ask the compiling system to use an extensive set of optimization techniques when generating the binary code.
  - **dbg**,**opt2** In the generated makefile, the commands invoking the *compiler* will define the preprocessor name **LEVEL** as **2**. Thus, when running the application, the preconditions will be evaluated, and the postconditions,invariants and checks will be possibly evaluated (depending on command-line switches, see **pel run**).
  - **opt1** In the generated makefile, the commands invoking the *compiler* will define the preprocessor name **LEVEL** as **1**. Thus the preconditions will be evaluated when running the application, but the statement associated to postconditions, invariants and checks will be removed during preprocessing stage.
  - **opt0** In the generated makefile, the commands invoking the *compiler* will define the preprocessor name **LEVEL** as **0**. Thus any statement associated to a precondition, a postcondition, an invariant or a check will be removed during the preprocessing stage.
  - **optpg** Same as **opt0** combined with **-profile**. Sets the more agressive optimisation level and sets the profiling option.
  - **optcov** Same as **-coverage**. Neither optimisation options nor debug information are generated by the compiling system.
- *bindir*
  Any file produced (objects, libraries, executables, dependency files ...) will be located in *bindir*. By default, the generated makefile, called **Makefile**, is created in the directory *bindir*.
- *sources*
  A list of directories and source files. **pel depend** will add the given source files and will add the source files found in the given directories. Any header file or source file found in a subdirectory of *sources* is considered to be part of the application. Header files are those having a **.h** or **.hh** extension whereas source files are those having a **.cpp**, **.cc**, **.c**, **.F** or **.f** extension.

Options

- **-man**
  Print the manual page and exit.
- **-verbose**
  Execute verbosely.
- **-D** *name*
  In the generated makefile, predefine name as a macro, with definition 1. This option may be used any number of times.
- **-D** *name=value*
  In the generated makefile, predefine *name* as a macro, with definition *value*. This option may be used any number of times.

- **`-compiler`** *comp*
  In the generated makefile, use the the *compiler* denoted by *comp* for all tasks of preprocessing, compilation, assembly and linking. Default is: **`gcc`**.
- **`-makefile`** *makefile*
  Set the name of generated makefile used for all tasks of compilation, assembly and linking. Default is: *bindir*/**`Makefile`**.
- **`-l`** *lib*|*dir*
  Add archive file *lib* to the list of files to link. The commands of the generated makefile will invoke the linker with "**`-l`***lib*" on the command line.
  When the argument is a directory, add all objects files found in the *dir* directory to the list of files to link.
  This option may be used any number of times.
- **`-path`** *searchdir*
  Add directory *searchdir* to the list of paths that the linker will search for archive libraries when invoked by the commands of the generated makefile. This option may be used any number of times.
- **`-I`** *searchdir*
  Add directory *searchdir* to the list of paths that the *compiler* will search for header files when invoked by the commands of the generated makefile. Note that any subdirectory of *srcdirs* that contains a header file is automatically added to that list of paths. This option may be used any number of times.
- **`-profile`**
  In the generated makefile, activate the profiling option when invoking the *compiler* and *linker*. This allows the profiling analysis of the application with tools such as **`gprof`**.
- **`-coverage`**
  In the generated makefile, activate the basic block coverage analysis option when invoking the *compiler* and *linker*. This allows the structural analysis of the application with tools such as **`tcov`**.

Examples

`pel depend -l pel1 dbg lib .`

The current application is made of all the header and source files located in any subdirectory of the working directory. The generated **`Makefile`** will be created in the directory **`lib`**. Further compilations will be performed with the **`dbg`** *compilation level*, and linking will be performed with the library **`libpel1`**.

`pel depend -l pel0 -I hea -compiler gcc opt1 bin src`

The current application is made of all the header and source files located in any subdirectory of **`src`**. The generated **`Makefile`** will be created in the directory **`bin`**. Further compilations will be performed by **`gcc`** with the **`opt1`** *compilation level*, and linking will be performed with the library **`libpel0`**. The current application probably uses header files that are not in the directory **`src`** and that are not PELICANS header files since the options **`-I hea`** is used.

Environment

It is possible to store arguments and options, overwritable by the command line arguments, in the environment variable **`PELDEPEND`**.

V.4.2 `pel build`

Description

The generation of executables, objects and libraries associated to a particular PELICANS-based application can be simply performed by using the GNU C<make> utility. The first step consists in building a suitable makefile with the **`pel depend`** utility. The second step consists in running GNU **`make`** with suitable arguments, a task whose responsiblity is assigned to **`pel build`**.

Essentially, GNU **`make`** is invoked with the following instructions:

1. read the makefile called **`Makefile`**, located in the directory *bindir* (unless otherwise specified with the **`-makefile`** options);
2. update the target determined by the mutually exclusive instructions **`-exe`**,**`-object`**,**`-archive`**;
3. use a specific set of make options that are determined by the calling options of **`pel build`**.

Synopsis

`pel build -man`

`pel build` *[options...]* **`-exe`** *bindir*

`pel build` *[options...]* **`-object`** *filename.o bindir*

`pel build` *[options...]* **`-archive`** *archname.so bindir*

Argument

- *bindir*
  Directory containing the makefile generated by **`pel depend`**, called **`Makefile`**. Any file produced by the execution of **`pel build`** (that is: when updating a target of **`Makefile`**) will be created in that directory (objects, libraries, executable, dependency files ...). The executable will be called **`exe`**.

Options

- **`-man`**
  Print the manual page and exit.
- **`-verbose`**
  Execute verbosely.
- **`-exe`**
  Ask **`make`** to update the target defined by the executable of the PELICANS-based application.
- **`-object`** *filename.o*
  Ask **`make`** to update the target defined by the module object *filename.o* (typically, *filename* is the name of one of the classes making up the considered application).
- **`-archive`** *archname.ext*
  Ask **`make`** to update the target defined by the archive *archname.ext* (the associated command depends on the extension *.ext*).
- **`-link_mode`** *link_mode*
  Set the linker:

link_mode=**`cc`**	:	use the **`C++`** linker (default)
link_mode=**`c`**	:	use the **`C`** linker
link_mode=**`f`**	:	use the **`fortran`** linker

- **`-make`** *makename*
  Use the command of name *makename* as the make command (default: **`make`**). This option is typically used for systems on which the GNU **`make`** utility is called **`gmake`**.
- **`-makefile`** *makefile*
  Set the makefile used for all tasks of compilation, assembly and linking.
  Default is: *bindir*/**`Makefile`**.
- **`-with`** *EXTlist*
  Notify that the current application **does** require the packages of *EXTlist* so that the archives associated to those packages should be added to the list of files to link. *EXTlist* is a comma separated list denoting external APIs that might be used by some components of PELICANS. This option is meaningful only for the targets of the options **`-exe`** and **`-archive`**. Note that **`pel depend`** defines some defaults in the generated makefile for all possible external libraries.
- **`-without`** *EXTlist*
  Notify that the current application **does not** require the packages of *EXTlist* so that the archives associated to those packages should **not** be added to the list of files to link. *EXTlist* is a comma separated list denoting external APIs that might be used by some components of PELICANS. This option is meaningful only for the targets of the options **`-exe`** and **`-archive`**. Note that **`pel depend`** defines some defaults in the generated makefile for all possible external libraries.

Examples

`pel depend -l pel1 dbg bin/dbg .`
`pel build -exe bin/dbg`

The current application is made of all the header and source files located in any subdirectory of the working directory. The generated **`Makefile`**, the executable **`exe`** and all the files created during the compiling process will be located in the subdirectory **`bin/dbg`** of the working directory.

`pel build -without petsc,opengl -exe lib`

All the files generated for and during the compilation process are located in the subdirectory lib of the working directory (and possibly in the working directory itself). Linking will be performed without the archives associated to the PETSc and OpenGL libraries.

`pel build -object myclass.o /usr/smith/appli/bin`

Update the target **`myclass.o`** of the makefile **`Makefile`** located in the directory **`/usr/smith/appli/bin`**, that is: build object file **`myclass.o`** in that directory.

Environment

It is possible to store arguments and options, overwritable by the command line arguments, in the environment variable **`PELBUILD`**.

V.4.3 `pel run`

Description

`pel run` executes a given PELICANS-based application with a given data file, and copy all the output messages in a given file (or set of files in parallel mode).

Execution is performed in the current directory unless the option **`-R`** is specified. Other options are devoted to the customization of the execution.

Synopsis

`pel run -man`

`pel run` *[options...] exe data resu*

`pel run -build_pattern` *filename* **`-R`** *dir exe data resu*

`pel run -check_pattern` *filename exe data resu*

Arguments

- *exe*
  Name of the executable of the PELICANS-based application to run (this executable has usually been built with a **`pel build`** command).
  If the environment has been set up as recommended in §V.3.2, the full path of the executable has been assigned to an environment variable.
- *data*
  Name of the file storing the PELICANS Hierarchical Data System used as a data file for the PELICANS-based application to run.
- *resu*
  In sequential mode : name of a file into which any message directed to the standard error or to the standard output will be copied (this file will be created or truncated).

  In parallel mode : basename of the files into which any message directed to the streams **`PEL::out()`** and **`PEL::err()`** will be copied (these files will be created or truncated). The extension of these files is the rank of their associated process.

Options

- **`-man`**
  Print the manual page and exit.
- **`-noverb`**
  Deactivate the verbose option of the PELICANS-based application (which is activated by default).
- **`-R`** *dir*
  Instead of running in the current directory, run recursively in all the subdirectories of *dir* (including *dir* itself) that contain a file of name *data*.
- **`-Cpost`**
  Activate the evaluation of the postconditions (only effective for sources compiled with the **`opt2`** or **`dbg`** *compilation level*, see §III.7.7, §V.2).
- **`-Call`**
  Activate the evaluation of the postconditions, invariants and checks (only effective for sources compiled with the **`opt2`** or **`dbg`** *compilation level*, see §III.7.7, §V.2).

- **`-Xpetsc`** *option*
  Transfer *option* to the PETSc external API.
- **`-X`** *option*
  Transfer *option* to external APIs.

To execute a PELICANS-based application in parallel on multiple processors, **`pel run`** forwards its request to **`mpirun`**.

- **`-np`** *n*
  Specify the number of processors to run on (call **`mpirun`** with **`-np`** *n* as an option).
- **`-machinefile`** *file*
  Take the list of possible machines to run on from the file *file* (call **`mpirun`** with **`-machinefile`** *file* as an option).
- **`-nolocal`**
  Call **`mpirun`** with **`-nolocal`** as an option.

During the final stage of the execution of a PELICANS-based application, the object to which **`PEL_Root::object()`** refers is destroyed (internally in the PELICANS framework), leading to the destruction of all remaining instances of subclasses of **`PEL_Object`** whose owner is not the NULL object. But it is the reponsibility of the developer of the PELICANS-based application to call the **`destroy()`** method on behalf of any objects that he may have created with NULL as owner. If such is not the case, some dynamically allocated memory might not be released when the execution terminates, and an *ad-hoc* message is printed by PELICANS.

The following two options help identifying the functions in which non-destroyed objects have been created. They should be used during two successive runs: the first run, with the **`-Cobject`** option identifies a non-destroyed object whereas the second run, with the **`-catch`** option locates the function in which it had been created.

- **`-Cobjects`**
  Assign an identification number to each object that has not been destroyed when the execution terminates, and display all available information about them (the identification number is an integer appearing between brackets on top of each printed block).
- **`-catch`** *nb*
  Display a warning message at runtime when the object whose identification number is *nb* is created (necessarily with the NULL owner).

A PELICANS-based application that can be executed by **`pel run`** requires a data file (whose name is the argument *data*). That file stores a Hierarchical Data System whose structure is implicitely defined by the application itself (through calls to the various member functions of **`PEL_ModuleExplorer`**). PELICANS offers the possibility to "learn" that structure during an execution and to record this infered knowledge in a called "pattern" file (option **`-build_pattern`**). This pattern file can be used in turn to check the conformity of other data file with this structure (option **`-check_pattern`**).

These two options are mutually exclusive. The activation of one of them inhibits the autocheck feature (see below).

- **`-build_pattern`** *filename*
  During the run, extract the pattern of the hierarchical data structure associated to *data*, and add it to the file called *filename* (which is created if it did not existed before, or extended otherwise).
- **`-check_pattern`** *filename*
  Prior to execution, check the conformity of *data* with the pattern file *filename*.

Let us consider an execution associated to a given data file *data*. If there is no available pattern file against which the conformity of *data* can be checked, **`pel run`** offers an "autocheck" mode which splits the execution in two steps: a run is first performed and a pattern is extracted from *data* (in a temporary file), then, after completion of that run, *data* is checked against the extracted pattern. This mode allows finding unread parts in the data file which may be caused by typing errors.

- **-autocheck**
  Perform the run in the "autocheck" mode. This option is activated by default, unless the options **-build_pattern** or **check_pattern** are present.
- **-noautocheck**
  Deactivate the "autocheck" mode.

Examples

pel run ../bin/exe data.pel resu

Run executable **exe** located in the directory **../bin/** with the data file **data.pel**, store all outputs in the file **resu** and perform an autocheck on **data.pel**.

pel run -Cpost ../bin/exe data.pel resu

Same as before, with in addition the evaluation of all postconditions of the member functions compiled with the **dbg** or **opt2** compilation level.

pel run -np 3 -machinefile ms $EXE0 data.pel resu

Launch the executable whose name is stored in the **EXEO** environment variable on 3 processors, with the data file **data.pel**. The first process will be on the current machine, whereas the other ones will be on the machines defined in the file **ms**). Outputs directed to **PEL::out()** and **PEL::err()** will be copied in the files **resu.0**, **resu.1** and **resu.2** (first, second and third process).

pel run -np 3 -machinefile ms -nolocal $EXE0 data.pel resu

Same as before, but the 3 processes will be launched on the machines defined in the file **ms**.

pel run -np 3 -machinefile ms -nolocal -Xpetsc -trace $EXE0 data.pel resu

Same as before, with a transfer of the **-trace** option to PETSc.

pel run -R Test bin/exe data.pel resu

If EXE denotes the file **exe** located in the subdirectory **bin** of the working directory, this command is equivalent as executing: **pel run** EXE **data.pel resu** in all the subdirectories of **Test** containing a file **data.pel**.

pel run -R -build_pattern etc/pattern.pel bin/exe data.pel resu

Same as before, with in addition the learning and storage of the requested structure of the data files in **etc/pattern.pel**. No autocheck is performed.

pel run -check_pattern ../etc/pattern.pel ../bin/exe data.pel resu

Check the conformance of **data.pel** with **../etc/pattern.pel** and, if successful, run subsequently **../bin/exe** with data file **data.pel** and store all outputs in the file **resu**. No autocheck is performed.

V.4.4 **pel test**

Description

Comparing in details two runs of an application is important for the sake of non regression or installation testing.

Given a hierarchy of directories containing reference runs of a given application, **pel test** will perform the following actions:

- create, in the working directory, a subdirectory in which that hierarchy is duplicated;
- in all subdirectories of the duplicate hierarchy, run the application with the associated reference data file (called **data.pel**);
- compare the results of this run with the reference results.

On completion, the conclusions of all these comparisons are recorded in a report file located in the current directory.

The essentials of **pel test** tasks are forwarded to the PELICANS-based application **peltest** (contained in the executable specified by the **-peltest_exe** option if any, or by the argument *exe*).

Further details are given below.

- If a test failure is pronounced, the character sequence "**Test failed**" will appear in the report file.
- Runs are performed with commands like:

  sequential : *exe* **data.pel -v** *[opts...]* **> resu**
  parallel : **mpirun** *[mpi_opts...]* *exe* **data.pel -v -o resu** *[opts...]*

  where the options *[opts...]* and *[mpi_opts...]* are determined for all runs by the calling options of **pel test** (see below) and for one specific run by a possible file **config.pel** in the reference directory of that run.

  A file **config.pel** containing:

  ```
  MODULE test_config
     run_options = vector( "...", "..." )
  END MODULE test_config
  ```

  leads to the addition in *[opts...]* of all items in the StringVector (only for the run associated to the directory containing the considered file **config.pel**).

  A file **config.pel** containing:

  ```
  MODULE test_config
     mpi_options = vector( "...", "..." )
  END MODULE test_config
  ```

  switches to parallel execution and leads to the addition in *[mpi_opts...]* of all items in the StringVector (only for the run associated to the directory containing the considered file **config.pel**). An additional optional StringVector data of keyword **mpi_machinefile** can be used to specify the list of possible machines to run on (this data is written on a temporary machine file transmitted to **mpirun** *via* the option **-machinefile**).
- The exit code is tested. The test failure is pronounced if it is non zero, unless it exists a file **config.pel**, stored in the reference directory, containing:

  ```
  MODULE test_config
     failure_expected = true
  END MODULE test_config
  ```

  in which case success might be pronounced if exit code is non zero and one or more produced files called **expected.err*** are identical (same name and same content) to those present in the reference directory.

- The test failure is pronounced if the **resu** file has not been produced.
- All files that have been produced during the run (other than **resu**) are compared (as described below) with the reference ones (that must exist). This comparison might be avoided from some particular file of a particular run if the associated reference directory contains a file **config.pel** containing:

  ```
  MODULE test_config
     files_to_ignore = vector( "...", "..." )
  END MODULE test_config
  ```

  The items of the StringVector correspond to files produced during the run for which no comparison will be performed.
- The comparison method between the reference and produced files depends on the format of these files.

  There are three ”native” formats understood by PELICANS: the format called GENE, for **TIC** postprocessing; the format called **PEL**, for Hierarchical Data Structures with the PELICANS format; the format called **CSV**, for comma separated values.

  The files with format **GENE**, **PEL** or **CSV** are compared to the reference ones with the PELICANS-based application **pelcmp** (contained in the executable specified by the **-peltest_exe** option if any, or by the argument *exe*). If they are not identical, the comparison results are recorded in the report file (the test failure is not pronounced since differences may be acceptable, depending on the use case).

  The other files are compared line by line with the reference ones. If they are not the same, the test failure is pronounced.
- The format of a file, say **save.zzz**, is determined as follows. It can be specified via a configuration file **config.pel** stored in the reference directory:

  ```
  MODULE test_config
    MODULE PEL_Comparator
      MODULE xxx               // xxx is a non significant name
        filename = "save.zzz"
        format = "CSV"         // either "GENE", "PEL" of "CSV"
      END MODULE xxx
    END MODULE PEL_Comparator
  END MODULE test_config
  ```

  If such a specification is absent, the format is identified on the basis of a motif appearing in the file name: **.gene** gives the **GENE** format, **.pel** gives the PEL format, **.csv** gives the **CSV** format.
- The files with format **GENE** or **PEL** contain data identified by keywords. The comparison might ignore some of these data if the associated reference directory stores a file **config.pel** containing:

  ```
  MODULE test_config
    MODULE PEL_Comparator
      MODULE xxx               // xxx is a non significant name
        filename = "save.zzz"  // file with format GENE or PEL
        ignore_data = vector( "...", "..." )
      END MODULE xxx
    END MODULE PEL_Comparator
  END MODULE test_config
  ```

  The items of the StringVector correspond to keywords of data that should be ignored during the comparison.
- The floating point values contained in the reference and produced files (with format **GENE**, **PEL** or **CSV**) are compared with **PEL::double_equality**. The last two arguments of this member function are respectively called **a_dbl_eps** (a kind of tolerance on relative errors) and **a_dbl_min** (a lower bound under which values are undistinguishable from zero).

By default, **a_dbl_eps** and **a_dbl_min** are equal to zero (which means that comparisons without any tolerance are performed). They can be given other values either globally (for all runs) using the **-dbl_eps** and **-dbl_min** options, or for a specific run via a file **config.pel** in the reference directory of that run.

For instance, a file **config.pel** containing:

```
MODULE test_config
   MODULE PEL_Comparator
      MODULE xxx                       // xxx is a non significant name
         filename = "save.csv"
         MODULE double_comparison
            dbl_min = 1.e-10
            dbl_eps = 1.e-8
         END MODULE double_comparison
      END MODULE xxx
   END MODULE PEL_Comparator
END MODULE test_config
```

will set **a_dbl_min**=10^{-10} and **a_dbl_eps**=10^{-8} for comparisons between the floating point values of the files **save.csv**.

Note that the command line options **-dbl_eps** and **-dbl_min** always overread the options stated in the files **config.pel**. Moreover the line option **-exact** can be used to ignore any setting of **a_dbl_min** and **a_dbl_max** in the **config.pel** files.

When the **-verify_pattern** option is activated, the behavior of **pel test** is slightly different: the only test performed is the conformance of the reference data file **data.pel** with the given pattern file.

Synopsis

pel test -man

pel test *[options...] exe dirs*

pel test -build_pattern *filename exe dirs*

pel test -verify_pattern *filename exe dirs*

pel test -build_then_verify_pattern *filename exe dirs*

Arguments

- *exe*
  Name of the executable of the PELICANS-based application to run.
  If the environment has been set up as recommended in §V.3.2, the full path of the executable has been assigned to an environment variable.

- *dirs*
  List of the directories defining the reference runs. Any subdirectory of an item of *dirs* containing a file **data.pel** is considered by **pel test** as a definition of a reference run whose data file is **data.pel**. This subdirectory must contain the reference version of all the files produced when calling *exe* with that data file. It might also contain (see above) a file called **config.pel**, and, more rarely, files called F<expected.err*>.

Options

- **-man**
  Print the manual page and exit.
- **-verbose**
  Execute verbosely.
- **-Cpost**
  Call **pel run** with this option for all runs.
- **-Call**
  Call **pel run** with this option for all runs.
- **-build_pattern** *filename*
  Call **pel run** with this option for all runs.
- **-verify_pattern** *filename*
  Do not perform the runs, but instead use the PELICANS-based application **check** (contained in the argument *exe*) to check the conformity of all reference data file **data.pel** with the pattern file *filename*.
- **-build_then_verify_pattern** *filename*
  Call **pel run** with the option **-build_pattern** *filename* for all runs and then check the conformity of all reference data file **data.pel** with the created pattern file *filename* (equivalent to two calls of **pel test** with successively the **-build_pattern** and the **-verify_pattern** options).
- **-test_directory** *dirname*
  Duplicate the hierarchy of directories containing the reference runs in the subdirectory *dirname* of the working directory, and run the application in the subdirectories of *dirname* for further result comparison with the reference runs (default: **PELICANS_TEST**).
- **-peltest_exe** *texe*
  Specify the executable containing the **peltest** and **pelcmp** applications. Default is the argument *exe* itself.
- **-dbl_eps** *eps*
  Specify the **a_dbl_eps** argument in calls to **PEL::double_equality** when comparing floating point values. This option is only significant for files with format **PEL**, **CSV** and **GENE**.
- **-dbl_min** *min*
  Specify the **a_dbl_min** argument in calls to **PEL::double_equality** when comparing floating point values. This option is only significant for files with format **PEL**, **CSV** and **GENE**.
- **-exact**
  Always perform comparisons between floating point values without any tolerance, whatever settings of **a_dbl_eps** and **a_dbl_eps** in files **config.pel**.

Examples

pel test ../bin/exe ../RegressionTests

Run executable **exe** located in the directory **../bin** with all data files **data.pel** contained in the subdirectories of **../RegressionTests** and compare the results with the reference ones. Create a report file in the current directory recording the conclusions of all comparisons.

pel test -build_pattern pat.pel ../bin/exe ../Tests

Same as before, with in addition the learning and storage of the requested structure of the data files in **pat.pel**.

pel test -verify_pattern pat.pel ../bin/exe ../Appli

Check the conformance with **pat.pel** of all files **data.pel** contained in a subdirectory of **../Appli**, and record the conclusions in a report file in the current directory.

V.4.5 pel predoc

Description

pel predoc will traverse a list of directory hierarchies from which it will infer a particular packaging and create a data file, both preparing a subsequent use of the PELICANS-based application **peldoc**.

Synopsis

pel predoc -man

pel predoc *[options...] descfile datfile* **appli** *dirs*

Arguments

- *descfile*
  Name of the description file to be produced.
- *datfile*
  Name of the **peldoc** data file to be produced.
- **appli**
  Name given by **peldoc** for the application to be documented.
- *dirs*
  List of the directories from which the packaging of the classes will be inferred.

Options

- **-man**
  Print the manual page and exit.
- **-verbose**
  Execute verbosely.
- **-Wno_unresolved**
  Add options to the data file of **peldoc** so that it will inhibit warning messages for assertions expressed with functions that are implemented outside the current application.

Examples

pel predoc doc/description.txt doc/data.pel beauty .

A description file **description.txt** and a **peldoc** data file **data.pel** will be produced in the subdirectory **doc** of the current directory, for an application that will be called "**beauty**", by recursively scanning all subdirectories of the current directory.

Bibliography

[1] Installation and administration of the PELICANS platform. Reference Documentation of PELICANS.

License for PELICANS

CeCILL-C FREE SOFTWARE LICENSE AGREEMENT

Notice

This Agreement is a Free Software license agreement that is the result of discussions between its authors in order to ensure compliance with the two main principles guiding its drafting:

- firstly, compliance with the principles governing the distribution of Free Software: access to source code, broad rights granted to users,
- secondly, the election of a governing law, French law, with which it is conformant, both as regards the law of torts and intellectual property law, and the protection that it offers to both authors and holders of the economic rights over software.

The authors of the CeCILL-C (for Ce[a] C[nrs] I[nria] L[ogiciel] L[ibre]) license are:

Commissariat à l'Energie Atomique - CEA, a public scientific, technical and industrial research establishment, having its principal place of business at 25 rue Leblanc, immeuble Le Ponant D, 75015 Paris, France.

Centre National de la Recherche Scientifique - CNRS, a public scientific and technological establishment, having its principal place of business at 3 rue Michel-Ange, 75794 Paris cedex 16, France.

Institut National de Recherche en Informatique et en Automatique - INRIA, a public scientific and technological establishment, having its principal place of business at Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay cedex, France.

Preamble

The purpose of this Free Software license agreement is to grant users the right to modify and re-use the software governed by this license.

The exercising of this right is conditional upon the obligation to make available to the community the modifications made to the source code of the software so as to contribute to its evolution.

In consideration of access to the source code and the rights to copy, modify and redistribute granted by the license, users are provided only with a limited warranty and the software's author, the holder of the economic rights, and the successive licensors only have limited liability.

In this respect, the risks associated with loading, using, modifying and/or developing or reproducing the software by the user are brought to the user's attention, given its Free Software status, which may make it complicated to use, with the result that its use is reserved for developers and experienced professionals having in-depth computer knowledge. Users are therefore encouraged to load and test the suitability of the software as regards their requirements in conditions enabling the security of their systems and/or data to be ensured and, more generally, to use and operate it in the same conditions of security. This Agreement may be freely reproduced and published, provided it is not altered, and that no provisions are either added or removed herefrom.

This Agreement may apply to any or all software for which the holder of the economic rights decides to submit the use thereof to its provisions.

Article 1 - DEFINITIONS

For the purpose of this Agreement, when the following expressions commence with a capital letter, they shall have the following meaning:

Agreement: means this license agreement, and its possible subsequent versions and annexes.

Software: means the software in its Object Code and/or Source Code form and, where applicable, its documentation, "as is" when the Licensee accepts the Agreement.

Initial Software: means the Software in its Source Code and possibly its Object Code form and, where applicable, its documentation, "as is" when it is first distributed under the terms and conditions of the Agreement.

Modified Software: means the Software modified by at least one Integrated Contribution.

Source Code: means all the Software's instructions and program lines to which access is required so as to modify the Software.

Object Code: means the binary files originating from the compilation of the Source Code.

Holder: means the holder(s) of the economic rights over the Initial Software.

Licensee: means the Software user(s) having accepted the Agreement.

Contributor: means a Licensee having made at least one Integrated Contribution.

Licensor: means the Holder, or any other individual or legal entity, who distributes the Software under the Agreement.

Integrated Contribution: means any or all modifications, corrections, translations, adaptations and/or new functions integrated into the Source Code by any or all Contributors.

Related Module: means a set of sources files including their documentation that, without modification to the Source Code, enables supplementary functions or services in addition to those offered by the Software.

Derivative Software: means any combination of the Software, modified or not, and of a Related Module.

Parties: mean both the Licensee and the Licensor.

These expressions may be used both in singular and plural form.

Article 2 - PURPOSE

The purpose of the Agreement is the grant by the Licensor to the Licensee of a non-exclusive, transferable and worldwide license for the Software as set forth in Article 5 hereinafter for the whole term of the protection granted by the rights over said Software.

Article 3 - ACCEPTANCE

3.1 – The Licensee shall be deemed as having accepted the terms and conditions of this Agreement upon the occurrence of the first of the following events:

(i) loading the Software by any or all means, notably, by downloading from a remote server, or by loading from a physical medium;

(ii) the first time the Licensee exercises any of the rights granted hereunder.

3.2 – One copy of the Agreement, containing a notice relating to the characteristics of the Software, to the limited warranty, and to the fact that its use is restricted to experienced users has been provided to the Licensee prior to its acceptance as set forth in Article 3.1 hereinabove, and the Licensee hereby acknowledges that it has read and understood it.

Article 4 - EFFECTIVE DATE AND TERM

4.1 EFFECTIVE DATE – The Agreement shall become effective on the date when it is accepted by the Licensee as set forth in Article 3.1.

4.2 TERM – The Agreement shall remain in force for the entire legal term of protection of the economic rights over the Software.

Article 5 - SCOPE OF RIGHTS GRANTED

The Licensor hereby grants to the Licensee, who accepts, the following rights over the Software for any or all use, and for the term of the Agreement, on the basis of the terms and conditions set forth hereinafter.

Besides, if the Licensor owns or comes to own one or more patents protecting all or part of the functions of the Software or of its components, the Licensor undertakes not to enforce the rights granted by these patents against successive Licensees using, exploiting or modifying the Software. If these patents are transferred, the Licensor undertakes to have the transferees subscribe to the obligations set forth in this paragraph.

5.1 RIGHT OF USE – The Licensee is authorized to use the Software, without any limitation as to its fields of application, with it being hereinafter specified that this comprises:

1. permanent or temporary reproduction of all or part of the Software by any or all means and in any or all form.
2. loading, displaying, running, or storing the Software on any or all medium.
3. entitlement to observe, study or test its operation so as to determine the ideas and principles behind any or all constituent elements of said Software. This shall apply when the Licensee carries out any or all loading, displaying, running, transmission or storage operation as regards the Software, that it is entitled to carry out hereunder.

5.2 RIGHT OF MODIFICATION – The right of modification includes the right to translate, adapt, arrange, or make any or all modifications to the Software, and the right to reproduce the resulting software. It includes, in particular, the right to create a Derivative Software.

The Licensee is authorized to make any or all modification to the Software provided that it includes an explicit notice that it is the author of said modification and indicates the date of the creation thereof.

5.3 RIGHT OF DISTRIBUTION – In particular, the right of distribution includes the right to publish, transmit and communicate the Software to the general public on any or all medium, and by any or all means, and the right to market, either in consideration of a fee, or free of charge, one or more copies of the Software by any means.

The Licensee is further authorized to distribute copies of the modified or unmodified Software to third parties according to the terms and conditions set forth hereinafter.

5.3.1 DISTRIBUTION OF SOFTWARE WITHOUT MODIFICATION – The Licensee is authorized to distribute true copies of the Software in Source Code or Object Code form, provided that said distribution complies with all the provisions of the Agreement and is accompanied by:

1. a copy of the Agreement,
2. a notice relating to the limitation of both the Licensor's warranty and liability as set forth in Articles 8 and 9,

and that, in the event that only the Object Code of the Software is redistributed, the Licensee allows effective access to the full Source Code of the Software at a minimum during the entire period of its distribution of the Software, it being understood that the additional cost of acquiring the Source Code shall not exceed the cost of transferring the data.

5.3.2 DISTRIBUTION OF MODIFIED SOFTWARE – When the Licensee makes an Integrated Contribution to the Software, the terms and conditions for the distribution of the resulting Modified Software become subject to all the provisions of this Agreement.

The Licensee is authorized to distribute the Modified Software, in source code or object code form, provided that said distribution complies with all the provisions of the Agreement and is accompanied by:

1. a copy of the Agreement,
2. a notice relating to the limitation of both the Licensor's warranty and liability as set forth in Articles 8 and 9,

and that, in the event that only the object code of the Modified Software is redistributed, the Licensee allows effective access to the full source code of the Modified Software at a minimum during the entire period of its distribution of the Modified Software, it being understood that the additional cost of acquiring the source code shall not exceed the cost of transferring the data.

5.3.3 DISTRIBUTION OF DERIVATIVE SOFTWARE – When the Licensee creates Derivative Software, this Derivative Software may be distributed under a license agreement other than this Agreement, subject to compliance with the requirement to include a notice concerning the rights over the Software as defined in Article 6.4. In the event the creation of the Derivative Software required modification of the Source Code, the Licensee undertakes that:

1. the resulting Modified Software will be governed by this Agreement,
2. the Integrated Contributions in the resulting Modified Software will be clearly identified and documented,
3. the Licensee will allow effective access to the source code of the Modified Software, at a minimum during the entire period of distribution of the Derivative Software, such that such modifications may be carried over in a subsequent version of the Software; it being understood that the additional cost of purchasing the source code of the Modified Software shall not exceed the cost of transferring the data.

5.3.4 COMPATIBILITY WITH THE CeCILL LICENSE – When a Modified Software contains an Integrated Contribution subject to the CeCILL license agreement, or when a Derivative Software contains a Related Module subject to the CeCILL license agreement, the provisions set forth in the third item of Article 6.4 are optional.

Article 6 - INTELLECTUAL PROPERTY

6.1 OVER THE INITIAL SOFTWARE – The Holder owns the economic rights over the Initial Software. Any or all use of the Initial Software is subject to compliance with the terms and conditions under which the Holder has elected to distribute its work and no one shall be entitled to modify the terms and conditions for the distribution of said Initial Software.

The Holder undertakes that the Initial Software will remain ruled at least by this Agreement, for the duration set forth in Article 4.2.

6.2 OVER THE INTEGRATED CONTRIBUTIONS – The Licensee who develops an Integrated Contribution is the owner of the intellectual property rights over this Contribution as defined by applicable law.

6.3 OVER THE RELATED MODULES – The Licensee who develops a Related Module is the owner of the intellectual property rights over this Related Module as defined by applicable law and is free to choose the type of agreement that shall govern its distribution under the conditions defined in Article 5.3.3.

6.4 NOTICE OF RIGHTS – The Licensee expressly undertakes:

1. not to remove, or modify, in any manner, the intellectual property notices attached to the Software;
2. to reproduce said notices, in an identical manner, in the copies of the Software modified or not;
3. to ensure that use of the Software, its intellectual property notices and the fact that it is governed by the Agreement is indicated in a text that is easily accessible, specifically from the interface of any Derivative Software.

The Licensee undertakes not to directly or indirectly infringe the intellectual property rights of the Holder and/or Contributors on the Software and to take, where applicable, vis-à-vis its staff, any and all measures required to ensure respect of said intellectual property rights of the Holder and/or Contributors.

Article 7 - RELATED SERVICES

7.1 – Under no circumstances shall the Agreement oblige the Licensor to provide technical assistance or maintenance services for the Software.

However, the Licensor is entitled to offer this type of services. The terms and conditions of such technical assistance, and/or such maintenance, shall be set forth in a separate instrument. Only the Licensor offering said maintenance and/or technical assistance services shall incur liability therefor.

7.2 – Similarly, any Licensor is entitled to offer to its licensees, under its sole responsibility, a warranty, that shall only be binding upon itself, for the redistribution of the Software and/or the Modified Software, under terms and conditions that it is free to decide. Said warranty, and the financial terms and conditions of its application, shall be subject of a separate instrument executed between the Licensor and the Licensee.

Article 8 - LIABILITY

8.1 – Subject to the provisions of Article 8.2, the Licensee shall be entitled to claim compensation for any direct loss it may have suffered from the Software as a result of a fault on the part of the relevant Licensor, subject to providing evidence thereof.

8.2 – The Licensor's liability is limited to the commitments made under this Agreement and shall not be incurred as a result of in particular: (i) loss due the Licensee's total or partial failure to fulfill its obligations, (ii) direct or consequential loss that is suffered by the Licensee due to the use or performance of the Software, and (iii) more generally, any consequential loss. In particular the Parties expressly agree that any or all pecuniary or business loss (i.e. loss of data, loss of profits, operating loss, loss of customers or orders, opportunity cost, any disturbance to business activities) or any or all legal proceedings instituted against the Licensee by a third party, shall constitute consequential loss and shall not provide entitlement to any or all compensation from the Licensor.

Article 9 - WARRANTY

9.1 – The Licensee acknowledges that the scientific and technical state-of-the-art when the Software was distributed did not enable all possible uses to be tested and verified, nor for the presence of possible defects to be detected. In this respect, the Licensee's attention has been drawn to the risks associated with loading, using, modifying and/or developing and reproducing the Software which are reserved for experienced users.

The Licensee shall be responsible for verifying, by any or all means, the suitability of the product for its requirements, its good working order, and for ensuring that it shall not cause damage to either persons or properties.

9.2 – The Licensor hereby represents, in good faith, that it is entitled to grant all the rights over the Software (including in particular the rights set forth in Article 5).

9.3 – The Licensee acknowledges that the Software is supplied "as is" by the Licensor without any other express or tacit warranty, other than that provided for in Article 9.2 and, in particular, without any warranty as to its commercial value, its secured, safe, innovative or relevant nature.

Specifically, the Licensor does not warrant that the Software is free from any error, that it will operate without interruption, that it will be compatible with the Licensee's own equipment and software configuration, nor that it will meet the Licensee's requirements.

9.4 – The Licensor does not either expressly or tacitly warrant that the Software does not infringe any third party intellectual property right relating to a patent, software or any other property right. Therefore, the Licensor disclaims any and all liability towards the Licensee arising out of any or all proceedings for infringement that may be instituted in respect of the use, modification and redistribution of the Software. Nevertheless, should such proceedings be instituted against the Licensee, the Licensor shall provide it with technical and legal assistance for its defense. Such technical and legal assistance shall be decided on a case-by-case basis between the relevant Licensor and the Licensee pursuant to a memorandum of understanding. The Licensor disclaims any and all liability as regards the Licensee's use of the name of the Software. No warranty is given as regards the existence of prior rights over the name of the Software or as regards the existence of a trademark.

Article 10 - TERMINATION

10.1 – In the event of a breach by the Licensee of its obligations hereunder, the Licensor may automatically terminate this Agreement thirty (30) days after notice has been sent to the Licensee and has remained ineffective.

10.2 – A Licensee whose Agreement is terminated shall no longer be authorized to use, modify or distribute the Software. However, any licenses that it may have granted prior to termination of the Agreement shall remain valid subject to their having been granted in compliance with the terms and conditions hereof.

Article 11 - MISCELLANEOUS

11.1 EXCUSABLE EVENTS – Neither Party shall be liable for any or all delay, or failure to perform the Agreement, that may be attributable to an event of force majeure, an act of God or an outside cause, such as defective functioning or interruptions of the electricity or telecommunications networks, network paralysis following a virus attack, intervention by government authorities, natural disasters, water damage, earthquakes, fire, explosions, strikes and labor unrest, war, etc.

11.2 – Any failure by either Party, on one or more occasions, to invoke one or more of the provisions hereof, shall under no circumstances be interpreted as being a waiver by the interested Party of its right to invoke said provision(s) subsequently.

11.3 – The Agreement cancels and replaces any or all previous agreements, whether written or oral, between the Parties and having the same purpose, and constitutes the entirety of the agreement between said Parties concerning said purpose. No supplement or modification to the terms and conditions hereof shall be effective as between the Parties unless it is made in writing and signed by their duly authorized representatives.

11.4 – In the event that one or more of the provisions hereof were to conflict with a current or future applicable act or legislative text, said act or legislative text shall prevail, and the Parties shall make the necessary amendments so as to comply with said act or legislative text. All other provisions shall remain effective. Similarly, invalidity of a provision of the Agreement, for any reason whatsoever, shall not cause the Agreement as a whole to be invalid.

11.5 LANGUAGE – The Agreement is drafted in both French and English and both versions are deemed authentic.

Article 12 - NEW VERSIONS OF THE AGREEMENT

12.1 – Any person is authorized to duplicate and distribute copies of this Agreement.

12.2 – So as to ensure coherence, the wording of this Agreement is protected and may only be modified by the authors of the License, who reserve the right to periodically publish updates or new versions of the Agreement, each with a separate number. These subsequent versions may address new issues encountered by Free Software.

12.3 – Any Software distributed under a given version of the Agreement may only be subsequently distributed under the same version of the Agreement or a subsequent version.

Article 13 - GOVERNING LAW AND JURISDICTION

13.1 – The Agreement is governed by French law. The Parties agree to endeavor to seek an amicable solution to any disagreements or disputes that may arise during the performance of the Agreement.

13.2 – Failing an amicable solution within two (2) months as from their occurrence, and unless emergency proceedings are necessary, the disagreements or disputes shall be referred to the Paris Courts having jurisdiction, by the more diligent Party.

Version 1.0 dated 2006-09-05.

Index

Contents