Programmation C++
Le langage impératif

Stéphane Vialette

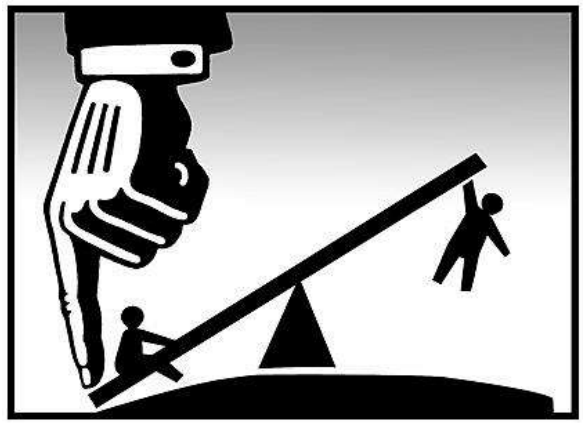
LIGM, Université Paris-Est Marne-la-Vallée

12 novembre 2012

C++ : chronologie

- 1971 : C (Dennis Ritchie)
- 1983 : C ansi
- 1980's : C++ (Bjarne Stroupstrup)
- 1995 : Java
- 2003 : C++ ansi

C++



Avantages

- Extrêmement populaire
- Compatible avec C (presque)
- Orienté objet
- Programmation générique
- Rapidité d'exécution

Inconvénients

- Lourdeurs d'écriture
- Gestion de la mémoire
- Trop de libertés
- Temps de compilation
- Taille des exécutables

Plan d'ensemble du cours

- Le langage impératif
- La couche objet
- Les template
- Programmation générique

Le langage impératif : plan

- Points communs/différences avec C
- Les commentaires
- Les types
- Déclarations de variables et fonctions
- Espaces de nommage
- Bibliothèque standard I :
  - bibliothèques C,
  - `string`,
  - entrées/sorties I
- Opérateurs

Le langage impératif : plan

- Pointeurs sur variables et fonctions
- Allocation dynamique
- Références
- Exceptions I
- Compilation séparée

Le langage C++

99% du langage C est compatible avec C++ :

- forme des instructions ;
- instructions de controle : `if`, `for`, `while`, `switch`, etc. ;
- Les types du C sont des types du C++.

Dennis Ritchie

> 1941-2011
> goodbye, world

Nouveautés du C++

- Commentaire de fin de ligne `//`.
- Les déclarations sont des instructions ;
- Types `bool` et `w_char` ;
- Paramètres par défaut dans les fonctions ;
- Références ;
- Véritables constantes ;
- Fonctions `inline` ;
- Opérateurs `new` et `delete` ;

Nouveautés du C++

- Espaces de noms ;
- Exceptions ;
- Classes, héritage… ;
- Typage dynamique ;
- Surcharge des opérateurs ;
- Templates

Le langage C++ : commentaires

```
                                                                  char
                                                              _3141592654[3141
        ],__3141[3141];_314159[31415],_3141[31415];main(){register char*
     _3_141,*_3_1415, *_3__1415; register int _314,_31415,__31415,*_31,
    _3_14159,__3_1415;*_3141592654=__31415=2,_3141592654[0][_3141592654
   -1]=1[__3141]=5;__3_1415=1;do{_3_14159=_314=0,__31415++;for( _31415
  =0;_31415<(3,14-4)*__31415;_31415++)_31415[_3141]=_314159[_31415]= -
1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
__3_1415    +__3141;for                (_31415 = 3141-
          __3_1415  ;                  _31415;_31415--
          ,_3_141 ++,                  _3_1415++){_314
          +=_314<<2 ;                  _314<<=1;_314+=
         *_3_1415;_31                   =_314159+_314;
         if(!(*_31+1)                   )* _31 =_314 /
         __31415,_314                   [_3141]=_314 %
         __31415 ;* (                   _3__1415=_3_141
        )+= *_3_1415                     = *_31;while(*
        _3__1415 >=                      31415/3141 ) *
        _3__1415+= -                     10,(*--_3__1415
       )++;_314=_314                     [_3141]; if ( !
       _3_14159 && *                     _3_1415)_3_14159
       =1,__3_1415 =                     3141-_31415;}if(
       _314+(__31415                      >>1)>=__31415 )
       while ( ++ *                       _3_141==3141/314
      )*_3_141--=0                        ;}while(_3_14159
      ) ; { char *                        __3_14= "3.1415";
      write((3,1),                        (--*__3_14,__3_14
      ),(_3_14159                          ++,++_3_14159))+
     3.1415926; }                          for ( _31415 = 1;
     _31415<3141-                          1;_31415++)write(
    31415% 314-(                           3,14),_3141592654[
  _31415    ] +                           "0123456789","314"
  [ 3]+1)-_314;                           puts((*_3141592654=0
,_3141592654))                             ;_314= *"3.141592";}
```

2 types de commentaires sont permis :

- Les commentaires de paragraphe : `/* .... */` ;
- Les commentaires de fin de ligne : `// ...`

Le langage C++ : types

`void`	type de retour,`void*`	
`bool`	booléens	`true`, `false`
`char`	caractères	'a'
`wchar_t`	caractères longs	L'a'
`int`	entiers	-3, 0712 , 0xf4d
`long int` ou `long`	entiers "longs"	239078L
`short int` ou `short`	entiers "courts"	124S

Le langage C++ : types

`float`	réels simple précision	2.5e12f
`double`	réels double précision	-4.5E-12
`long double`	réels en “quadruple” précision	4.6L

Le langage C++ : types

- Les types ”entiers” et ”caractères” peuvent être signés (`signed`) ou non signés (`unsigned`).
  – par défaut, les entiers sont signés
  – ce n’est pas le cas des caractères :

  `char`, `signed char` et `unsigned char` sont 3 types $\neq$.
- Taille des types :
  – toujours un multiple de celle de `char` (donné par `sizeof`).
  – `short` $\leq$ `int` $\leq$ `long`;
  – `float` $\leq$ `double` $\leq$ `long double`.

Le langage C++ : types

À partir de ces types de base, on peut construire d'autres types :

- Pointeurs : désignent l'adresse d'une valeur en mémoire
  `int*` : type de l'adresse d'un `int`.
- Références : désigne un objet créé par ailleurs.
  `int&` : type d'une référence sur un `int`.
- Tableaux : tableaux de valeurs d'un même type
  `float x[5]` : `x` tableau de cinq `float`.

Le langage C++ : types

- Unions : permet d'utiliser une variable pour stocker des valeurs de différents types :

```
union exemple_union_t {
  int entier;
  double decimal;
  char lettre;
};
```

Ceci définit un nouveau type `exemple_union_t`

- Une variable `x` de ce type peut stocker un `int`, un `double` ou un `char`. S'il s'agit d'un `double` par exemple, sa valeur sera accessible en lecture et en écriture en écrivant `x.decimal`

Le langage C++ : types

- Enumérations : permet de définir un type acceptant un nombre fini de valeurs :

```
enum exemple_enum_t{
  ROUGE, VERT, BLEU, JAUNE
};
```

Ceci définit un nouveau type `exemple_enum_t`

- Une variable `x` de ce type doit prendre une des valeurs définies ; celles-ci sont assimilées à des entiers, dont on peut fixer explicitement la valeur, mais ne sont pas de type (par exemple `BLEU = 3`).

Le langage C++ : types

- Structures : permet de définir un type pour stocker un nombre fixé de valeurs de types éventuellement différents.

```
struct exemple_struct_t {
  int abs,ord;
  exemple_enum_t couleur;
};
```

Ceci définit un nouveau type `exemple_struct_t`

- On accède aux différents champs (ou attributs) d'une variable `x` de ce type en précisant après un point le nom du type, par exemple `x.abs`.

On verra que `struct` permet de créer des types objets.

Le langage C++ : Déclarations

- Comme en Java, on peut déclarer une variable n'importe où dans un bloc ;
- Tout identifiant doit être déclaré avant d'être utilisé dans un programme.

  C'est en particulier vrai pour les fonctions.

- Lors de la déclaration d'une fonction,

  la liste d'arguments `()` est synonyme de `(void)` .

  On doit déclarer les types des arguments en même temps que la fonction.

Le langage C++ : Déclarations

- On peut initialiser une variable à l'aide de = (`int x=3;`) ou en plaçant la valeur initiale entre parenthèses (`int x(3);`)

Le langage C++ : Paramètres par défaut

C++

On peut préciser lors de la déclaration d'une fonction, avec =, une valeur par défaut pour certains paramètres :

```
int norme(int x,int y=0){
  return x*x+y*y;
}
```

peut être appelée aussi bien par `norme(5)` que `norme(5,3)`.

Le langage C++ : Paramètres par défaut

- Si un paramètre a une valeur par défaut, tous les paramètres suivants doivent en avoir une aussi.
- Lors de l'appel d'une fonction, si on ommet des arguments, il doit s'agir des derniers, et ceux-ci doivent avoir une valeur par défaut spécifiée.
- La valeur des paramètres par défaut de la fonction doit être donnée lors de sa déclaration et non lors de sa définition (si elles sont distinctes).

Le langage C++ : surcharge de fonctions

C++

- On peut définir en C++ différentes fonctions de même nom, à condition que les types de leurs paramètres soient différents.
- Attention, si on déclare une fonction

```
int norme (int x, int y=0);
```

et une fonction

```
int norme(int x);
```

l'appel de `norme` avec un seul paramètre est ambigu et cause une erreur de compilation.

Le langage C++ : fonctions inline

C++

- La déclaration d'une fonction peut être précédée de `inline`, afin de demander au compilateur de remplacer chaque appel de cette fonction par son code.
- Ceci remplace avantageusement la plupart des macros comme

```
#define max(x,y)  (x>y?x:y)
```

- Le compilateur n'est pas obligé de suivre cette directive (selon l'optimisation).

Le langage C++ : fonctions inline

- Attention, comme l'appel de la fonction est remplacé par son code, celui-ci doit être connu au moment de l'appel : pas d'édition de liens.
- De même, une fonction inline ne doit pas être récursive et on ne peut pas faire de pointeurs sur une fonction inline.

Variables constantes

- Si `const` précède une déclaration de variable, qui doit dans ce cas être initialisée, cette variable n'est pas modifiable.
- C++ Cette variable constante sera dans la mesure du possible remplacée par sa valeur lors de la compilation.

```
const int MAX=3;
```

  est préférable à

```
#define MAX 3
```

- C++ Si le champ d'une structure est précédé de `mutable`, ce champ pourra être modifié même si la structure elle-même est constante.

Variables constantes

- `const` est un modificateur de type.
- `const int` et `int` sont deux types différents.
- On peut convertir un `int` en `const int` de manière implicite, mais pas le contraire.

Classes de stockage

`auto`	Classe de stockage par défaut : durée de vie de la variable restreinte au bloc (mention facultative)
`static`	Durée de vie égale à celle du programme. Pour une variable globale, visibilité restreinte au fichier
`register`	Place la variable dans un registre (peu utilisé)
`volatile`	Modifiable par un autre processus, doit être rechargée à chaque appel
`extern`	Déclaration d'une variable globale définie dans un autre fichier

`const`,`mutable`, Déjà vus

Espace de nommage

Afin d'isoler les différents modules d'un programme, on peut définir des *espaces* identifiés par leur nom :

```
namespace toto{
  int mafonction(int x){
    /* ...  */
  }

  struct montype{
    /* ...  */
  };
}
```

Ceci joue le rôle des packages Java :

```
package toto;
```

Les espaces de nommages peuvent s'emboîter.

Espace de nommage

Pour utiliser des identifiant déclarés dans un espace de nommage, il faut

- soit se trouver à l'intérieur du même espace de nommage.
- soit utiliser l'opérateur de *résolution de portée* :

```
toto::mafonction(5);
toto::montype x;
```

Espace de nommage

- soit avoir précédemment spécifié qu'il s'agit de cette fonction :

```
using toto::mafonction;
using toto::montype;
```

  On peut alors directement appeler

```
mafonction(5);
montype x;
```

  Ceci est l'analogue d java `import toto.montype;`.

- soit avoir précédemment spécifié que l'on souhaite avoir accès à ce qui est défini dans cet espace :

```
using namespace toto;
```

  Ceci est l'analogue du java `import toto.*;`

Bibliothèques standards du C

- Pour utiliser proprement des fonctionnalités de la bibliothèque standard C, on inclut les fichiers qui porte le même nom qu'en C, sauf qu'ils sont précédé de la lettre `c` et ne sont pas suivis du suffixe `.h`.
- Par exemple : `cassert cctype cerrno cfloat ciso646 climits clocale cmath csetjmp csignal cstdarg cstddef cstdio cstdlib cstring ctime cwchar cwctype`
- Les fonctions définies se trouve dans l'espace de nommage std.

Chaînes de caractères

- Une chaîne de caractères est un tableau de caractères qui se termine par le caractère `'\0'` de code ascii `0`.
- Par exemple, la chaîne `"bonjour"` est de type `const char[8]`.
- Pour préciser qu'une chaîne de caractères est constituée de caractères longs, on la fait précéder de `L`.
- Par exemple, la chaîne `L"bonjour"` est de type `const wchar_t[8]`.

`string`

- La bibliothèque standard fournit un type `string` qui permet de manipuler facilement les chaînes de caractères. On peut ainsi accéder à la longueur d'un `string s;` par `s.length()`, concaténer deux strings (`s1 + s2`), etc.
- Pour l'utiliser, on doit inclure : `#include<string>`

Entrées/sorties simples

- Au lieu d'utiliser les `printf`, `scanf` du C, on préfère utiliser les opérateurs de flots :
  - `<<` pour les flots sortants, et
  - `>>` pour les flots entrants.
- La bibliothèque de gestion standard des flots est `iostream`,qui est dans l'espace de nom `std`.

```
#include<iostream>
```

(Remarque : on ne met pas `.h`)

- Afficher une chaîne de caractères :

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

`endl` effectue un retour à la ligne et vide le flot.

Entrées/sorties simples

Selon le type de ce qui lui est appliqué, l'opérateur de flot effectue la conversion en chaîne de caractères.

```
std::cout << 123 << std::endl;
```

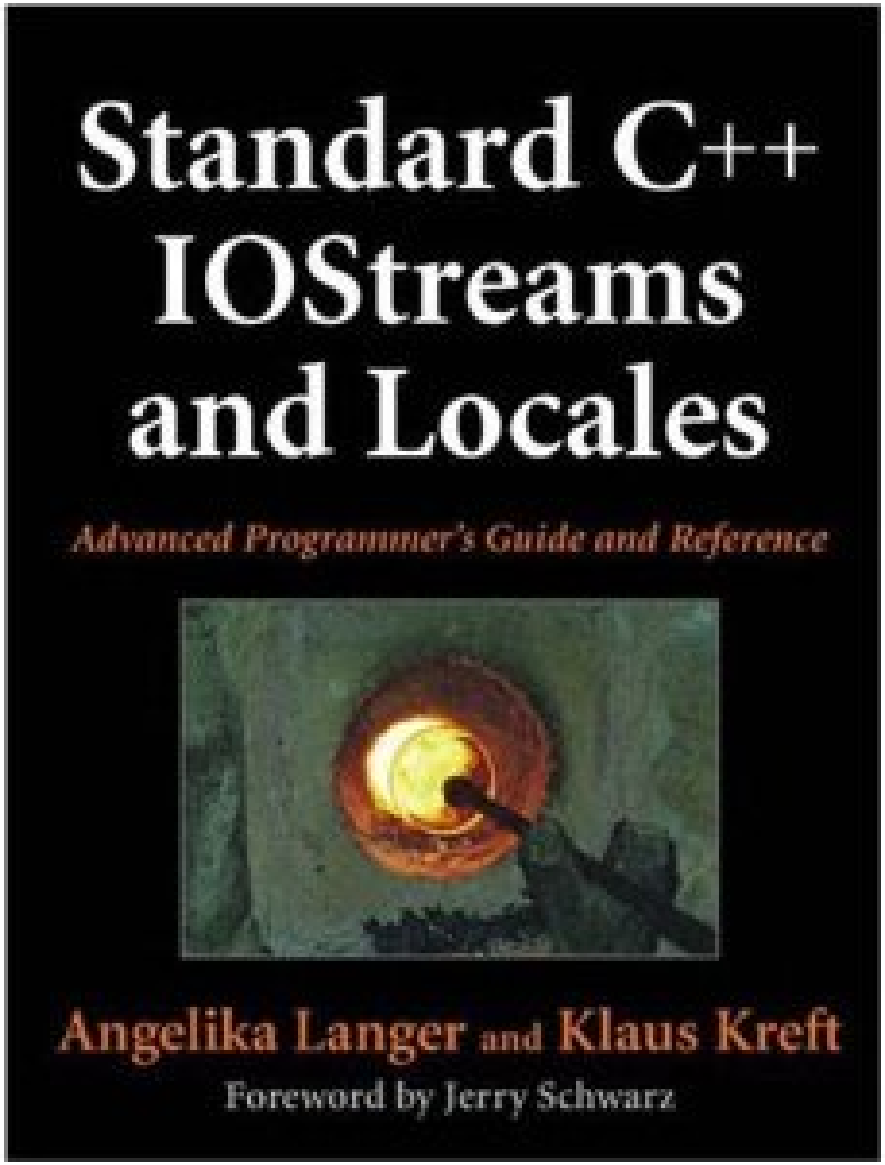
On peut enchaîner les opérateurs de flots :

```
int x;
/* calculs... */
std::cout << "Le resultat est "
<< x << std::endl;
```

- `cout` est le flot correspondant à la sortie standard,
- `cerr` correspond à la sortie standard d'erreur,
- `cin` correspond à l'entrée standard :

```
int x;
std::cin >> x;
```

Entrées/sorties (pas simples !)

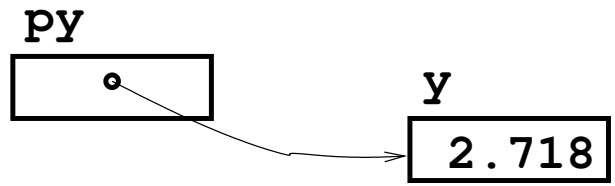


Pointeurs

- Un pointeur représente une adresse en mémoire qui contient une valeur d'un certain type.
- Un pointeur est typé selon l'objet sur lequel il pointe.
  ex :
  `int*` est le type des pointeurs sur `int`.
- On peut obtenir l'adresse d'une variable grâce à l'opérateur d'*indirection* préfixe `&` :
  ex :
  ```
  double y; double *py = &y;
  ```
  Le pointeur `py` a pour valeur initiale l'adresse de la variable `y`.

Pointeurs

- Inversement, on peut obtenir la valeur stockée à une certaine adresse grâce à l'opérateur de *déréférencement* préfixe `*` :



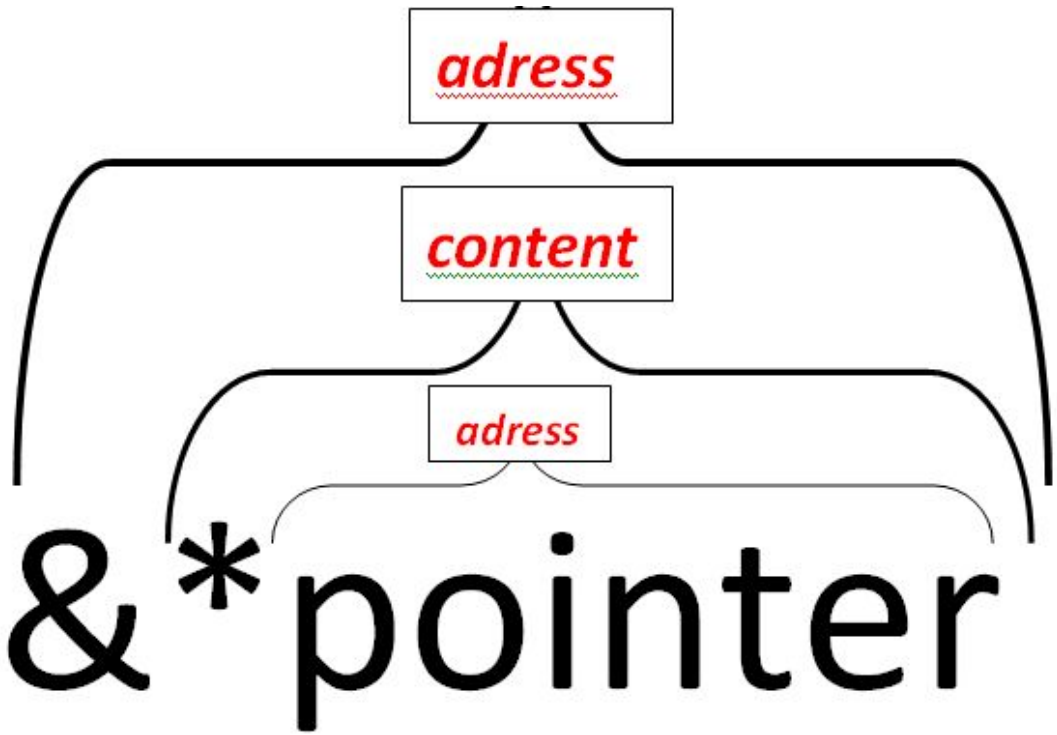

```
double y=2.718;
double *py = &y;
std::cout << py
<< " .... "
<< *py
<< std::endl;
```

affiche par exemple `0xbf980170 .... 2.718`

- Un pointeur déréférencé est une "lvalue", c'est-à-dire qu'on peut lui affecter une valeur s'il ne pointe pas sur un objet constant :

```
*py=3.14;
```

Pointeurs



Pointeur `void*`

- Une adresse non typée est de type `void*`.
- N'importe quel pointeur peut être converti implicitement en `void*` :

```
double y=2.718;
void *py = &y;
```

- On ne peut pas déréférencer un pointeur `void*`.
- On ne peut pas faire d'arithmétique sur un pointeur `void*`.
- On peut convertir un pointeur `void*` en n'importe quel type de pointeur grâce au cast :

```
double y=2.718;
void *py = &y;
int *pi= (int*) py;
```

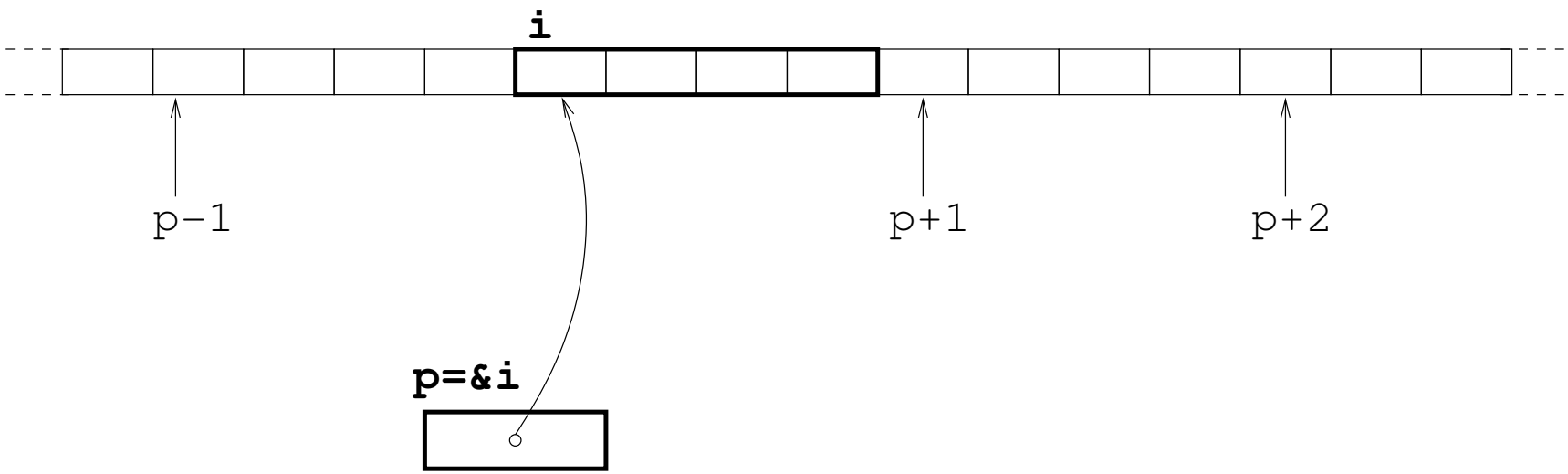
Arithmétique des pointeurs

- À un pointeur typé, on peut ajouter ou retrancher un entier *k*. Le résultat est un pointeur de même type correspondant à un décalage de *k* fois la taille de l'objet pointé.

```
int i;
int *p=(&i)+1;
```

`p` indique l'adresse qui suit immédiatement `i`.

- Le décalage effectué dépend donc du type du pointeur.



Pointeurs et `const`

Attention Il ne faut pas confondre "*pointeur constant*" et "*pointeur sur un objet constant*".

- ```
  const int* p;
  ```

  déclare un pointeur sur un entier constant.
- ```
  int* const p= &x;
  ```

  déclare un pointeur constant sur un entier.
- ```
  const int* const p= &x;
  ```

  déclare un pointeur constant sur un entier constant.
  ```

Pointeurs et `const`

- Un ”pointeur sur un objet constant” peut pointer sur un objet qui n’est pas constant :

```
int i;
const int *p = &i;
```

- Le contraire n’est pas vrai :

```
const int i;
int *p = &i;
```

produit

```
error: invalid conversion from ‘const int*’ to
‘int*’
```

Pointeurs sur fonction

- Si une fonction n'est pas `inline`, on peut accéder à son adresse :

```
int ma_fonction(double t) {
  /* code */
}
...
int (*pf)(double) = &ma_fonction;
int y = (*pf)(3.5);
```

- Noter que dans ce cas, les opérateurs d'indirection et de déréférencement sont facultatifs.

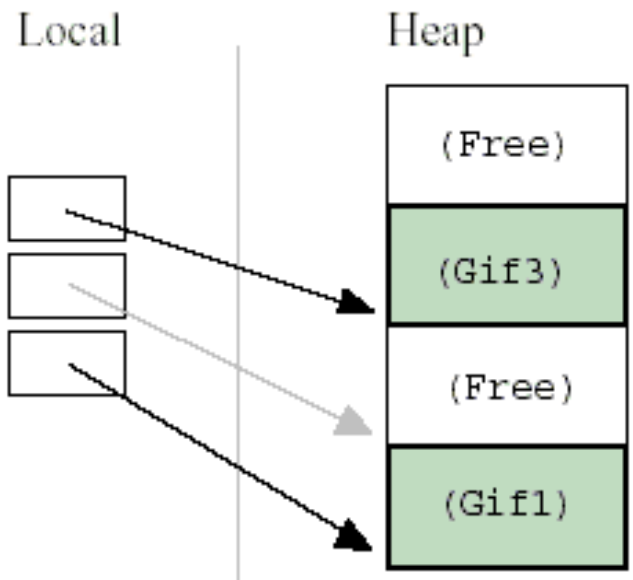
```
int (*pf)(double) = ma_fonction;
int y = pf(3.5);
```

Pointeur sur structure

- Si `p` est un pointeur sur une structure qui contient un champ `ch`, on peut directement accéder à ce champ via l'opérateur `->` :
- `p->ch` est équivalent à `(*p).ch`.

Allocation dynamique

- Si un pointeur pointe sur une variable locale, il devient invalide à la fin de la durée de vie de celle-ci.
- Pour éviter ceci, on peut allouer explicitement de la mémoire pour stocker une variable d'un certain type. Cette mémoire restera allouée jusqu'à ce qu'elle soit explicitement libérée.



Allocation dynamique

- Pour allouer de la mémoire, on utilise l'opérateur `new` suivi du type que l'on veut allouer ; la valeur retournée est un pointeur sur ce type.

```
float *f = new float;
```

- Pour libérer cet espace mémoire, il faut utiliser l'opérateur `delete` :

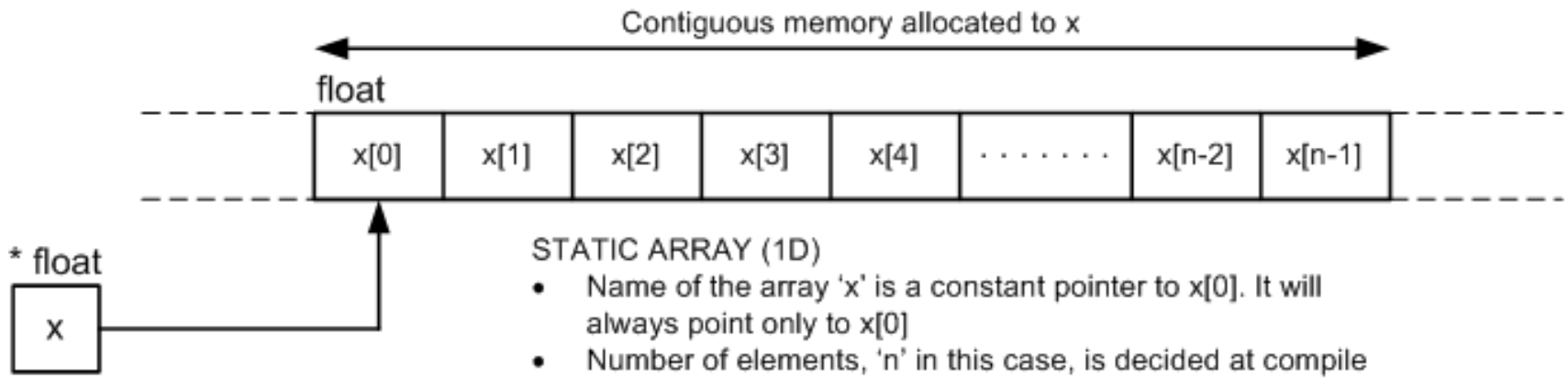
```
delete f;
```

- Attention Ne pas utiliser la fonction `free` de la bibliothèque standard du C pour désallouer de l'espace réservé par `new`.

Tableaux dynamiques

Un tableau en C++ doit avoir une taille connue par le compilateur :

```
double tab[100];
const int i=150;
long tableau[i];
```



STATIC ARRAY (1D)

- Name of the array 'x' is a constant pointer to x[0]. It will always point only to x[0]
- Number of elements, 'n' in this case, is decided at compile time and cannot be changed during run-time
- Memory allocation and deallocation is done automatically, not a responsibility of the programmer

Tableaux dynamiques

- Si on veut utiliser un tableau dont la taille ne sera connue qu'à l'exécution, il faut utiliser un pointeur et allouer explicitement l'espace correspondant, avec l'opérateur `new[]`.

```
int x;
cin >> x;
double *tab = new double[x];
```

- Le pointeur peut alors être utilisé comme un tableau :

```
tab[5] = 3.4;
```

(valide si `tab` est au moins de taille 6).

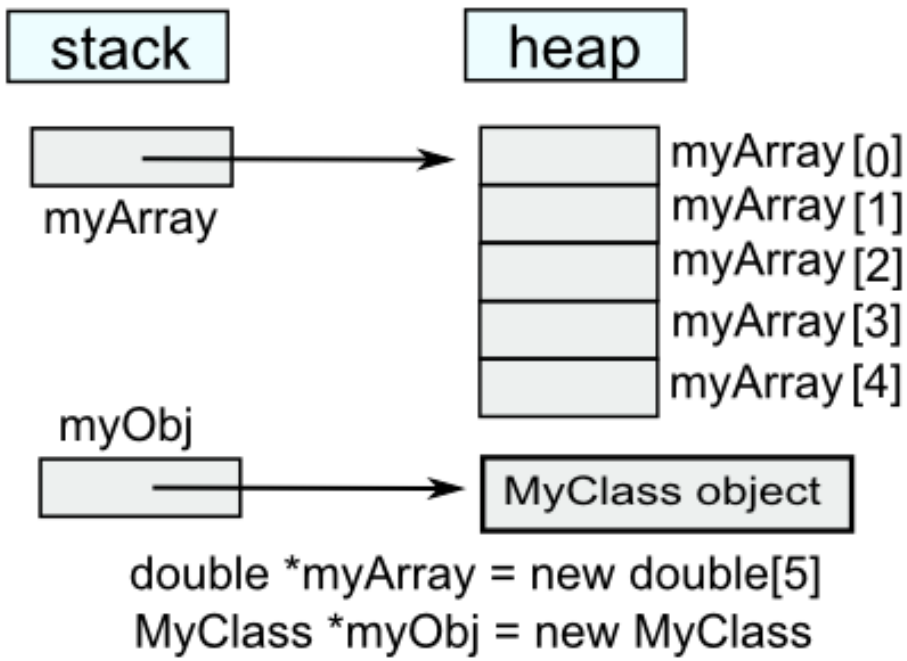
- `tab[5]` est équivalent à `*(tab+5)`

Tableaux dynamiques

- Un tableau dynamique doit être libéré avec l'opérateur `delete[]` :

```
delete[] tab;
```

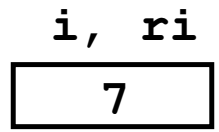
- `delete` et `delete[]` appelés sur des pointeurs nuls ne font rien.



Références

- Une référence est un alias pour une autre variable. Elle doit être initialisée à sa création.

```
int i;
int &ri=i;
```



- Dans cet exemple, `ri` est une référence à la variable `i`.
- Toute modification de l'une entraîne la même modification de l'autre.

```
ri=7;
```

  Après ceci, `i` vaut 7.

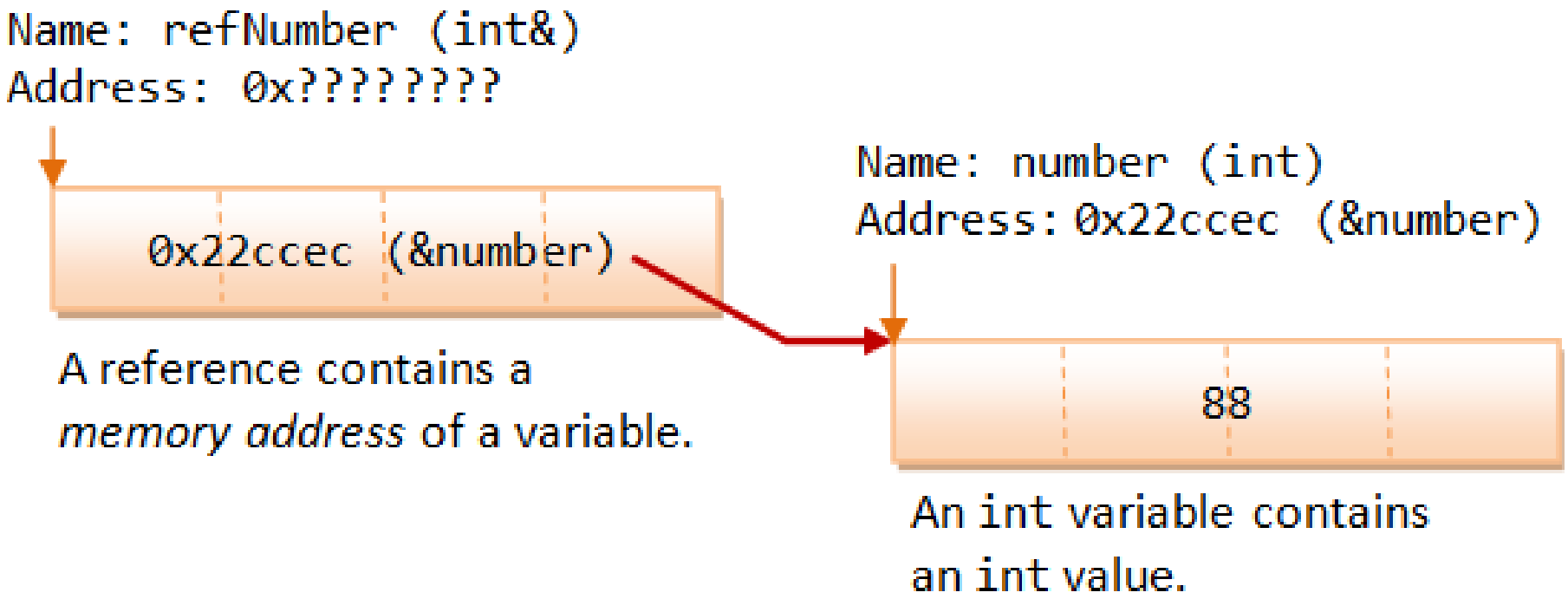
- De même qu'un pointeur, une référence peut être une référence à un objet constant, ou non. Par contre, la référence elle-même est toujours constante, elle se comporte comme un pointeur constant.

Références

Une référence constante peut être une référence sur une constante :

```
const int &ri=7;
```

`ri` est une référence à la constante 7.



Utilisation des références comme argument

- Le mode normal de passage des arguments d'une fonction est par valeur.

```
int ma_fonction(double t) {/*...*/};
```

- Lors de l'appel de cette fonction, la valeur avec laquelle la fonction est appelée est copiée dans la variable locale `t` qui sera désallouée lors du retour de fonction.
- C'est la raison pour laquelle, en C, on passe souvent des pointeurs en argument plutôt que des valeurs.

```
void swap(int *a, int *b) {
  int x=*a;
  *a=*b;   *b=x;
}
// ...
int x,y;
// ...
swap(&x, &y);
```

Utilisation des références comme argument

- En C++, on peut passer les arguments d'une fonction par référence :

```
int ma_fonction(double &t) {
  /*...*/
}
```

- Lors de l'appel de cette fonction, `t` devient une référence sur la variable passée en argument. Celle-ci pourra donc être modifiée :

```
void swap(int &a, int&b) {
  int x=a; a=b; b=x;
}
int x,y;
/* ...  */
swap(x,y);
```

- Une telle fonction ne peut pas être appelée avec une constante pour argument, sauf si la référence est constante. Un tel mécanisme est utilisé pour éviter la copie des objets, par exemple.

Utilisation des références comme type de retour

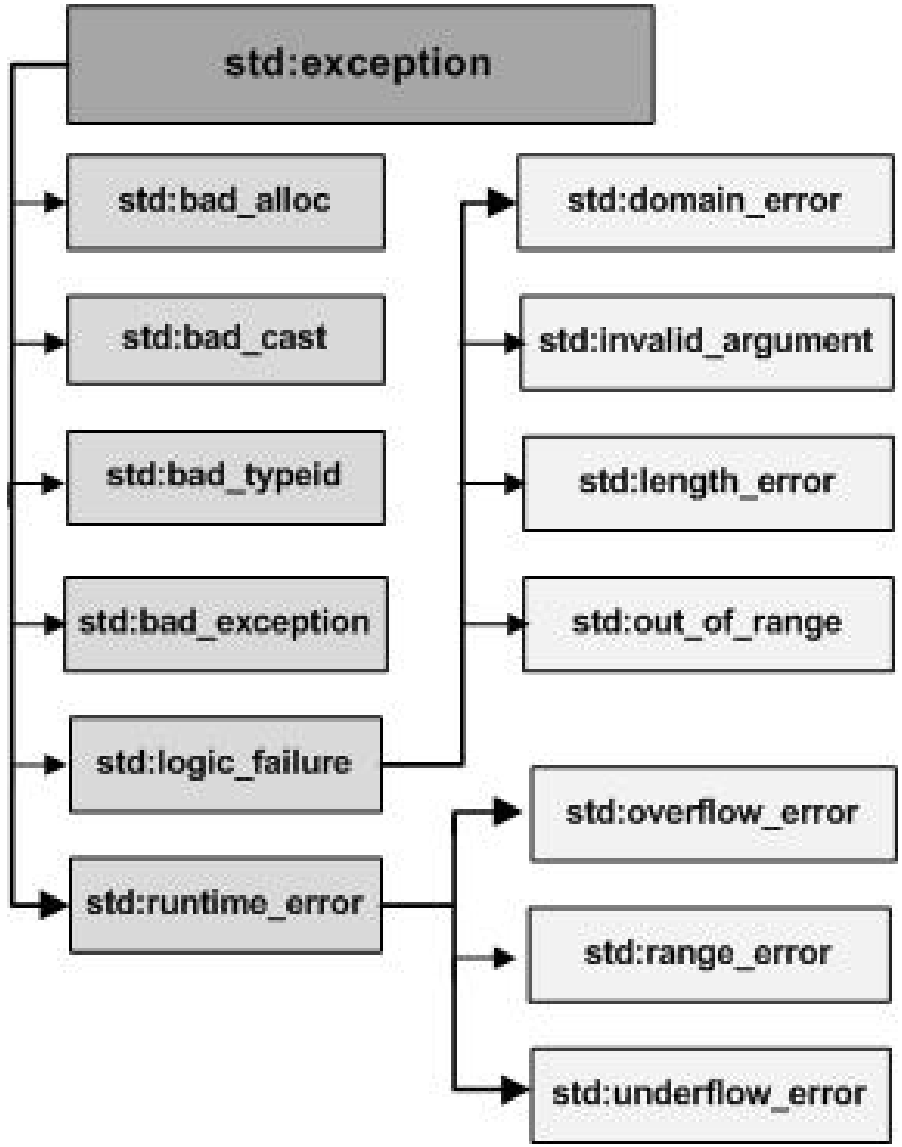
- Lors du retour de fonction, la valeur calculée est habituellement copiée afin de pouvoir être récupérée par la fonction appelante.
- Si on veut que l'objet retourné ne soit pas une copie, mais bien l'objet calculé dans la fonction appelée, on peut effectuer un retour par référence.
- Attention, s'il s'agit d'une référence à une variable locale automatique, celle-ci étant automatiquement désallouée, la référence est invalide.

Utilisation des références comme type de retour

```
const int& ma_fonction(double t){
  static int nb_appels=0;
  /*...*/
  return ++nb_appels;
}
```

On récupère ainsi une référence sur la variable statique `nb_appels` (qu'on ne pourra pas modifier car la référence est constante).

Les exceptions I

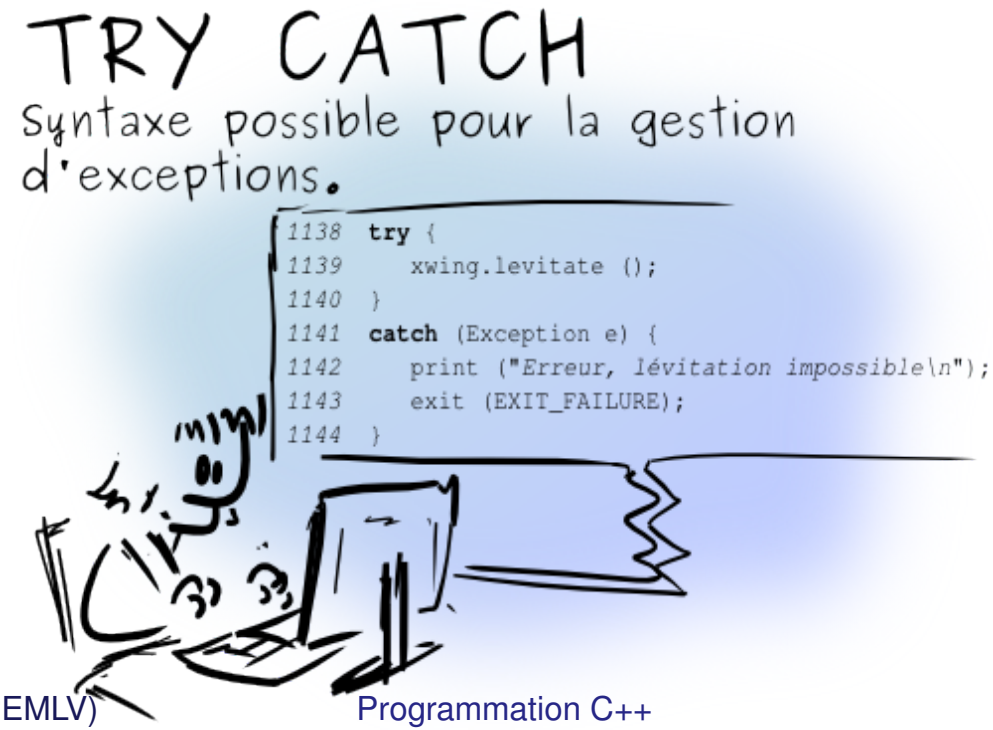


Les exceptions I : `throw`

- Pour lever une exception, on utilise l'opérateur `throw` suivi d'une valeur de n'importe quel type.
- Cette exception, si elle n'est pas attrapée se propage en provoquant l'arrêt successif des méthodes et fonctions appelantes jusqu'à la fonction `main`, puis provoque l'affichage d'un message d'erreur et l'arrêt du programme.

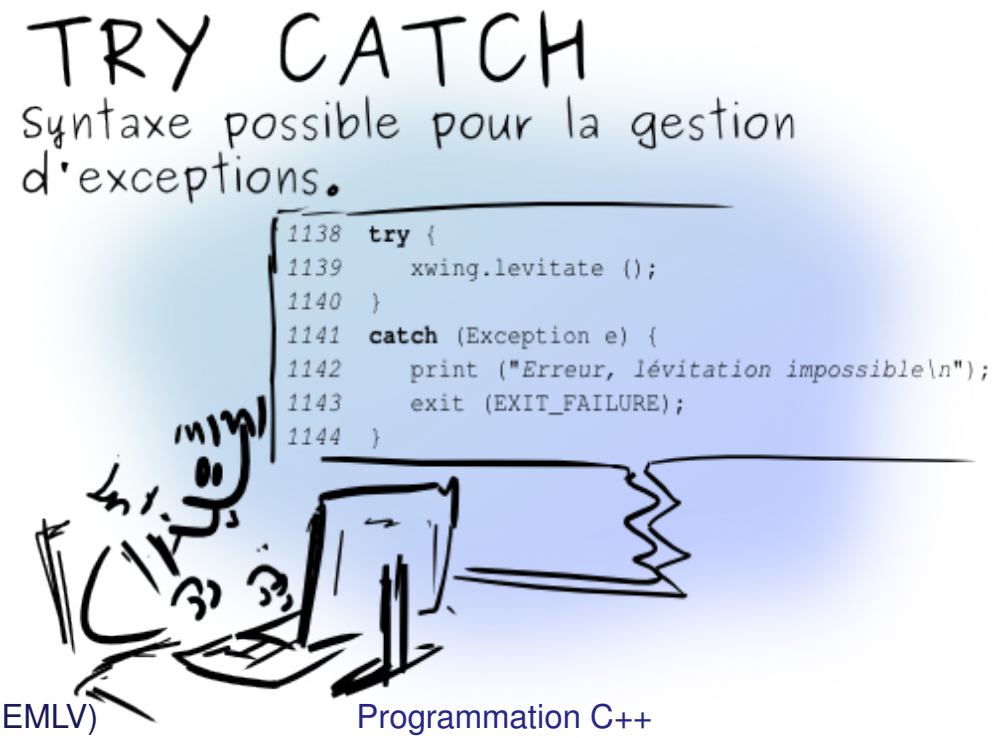
Les exceptions I : `try catch`

- Pour qu'une exception soit interceptée, elle doit survenir dans un bloc introduit par `try` suivi d'au moins un bloc introduit par `catch(type_exc &e)`.
- Si une exception est survient dans le bloc `try`, les types d'exceptions gérés par les blocs `catch` sont examinés successivement. Le premier bloc ayant un argument du type de l'exception ou d'une de ses classes mères est exécuté.



Les exceptions I : `try catch`

- On peut utiliser `catch(...)` pour le dernier bloc `catch` qui sera alors exécuté si aucun autre catch ne correspond à l'exception survenue.
- Pour les autres `catch`, le type spécifié sera généralement une référence afin d'éviter la recopie de l'objet exception.
- Pour relancer dans un bloc `catch` l'exception interceptée, il suffit d'écrire `throw ;`



Les exceptions I : exceptions autorisées

- On n'est pas obligé de spécifier qu'une fonction ou méthode est susceptible de propager une exception, mais c'est possible :

```
int ma_fonction(int x) throw (my_exception);
```

- Si une exception d'un autre type survient dans la fonction/méthode, ceci provoque l'affichage d'un message d'erreur et l'arrêt du programme. (voir gestion plus loin)

Compilation séparée

- Comme en C, on peut compiler des sources en binaires, puis effectuer une *édition de liens* entre ces binaires pour obtenir un exécutable.
- Pour celà, on sépare les déclarations de fonctions et de variables de leurs définitions.
- Les déclarations se trouvent dans des *headers* de suffixe `.hh`. et les définitions (implémentations) dans des fichiers de suffixe `.cc` ou `.cpp`
- Les fichiers contenant des implémentations et destinés à être inclus sont parfois appelés `.hxx`.

Compilation séparée

```
// int_math.hh
#ifndef INT_MATH_HH
#define INT_MATH_HH
namespace persolib {
  unsigned int sqrt(const unsigned int&);
  unsigned int gcd(const unsigned int&,
  const unsigned int&);
  // ...
}
#endif

//main.cc
#include"int_math.hh"
#include<iostream>

int main(){
  std::cout << persolib::sqrt(235) << std::endl;
}
```

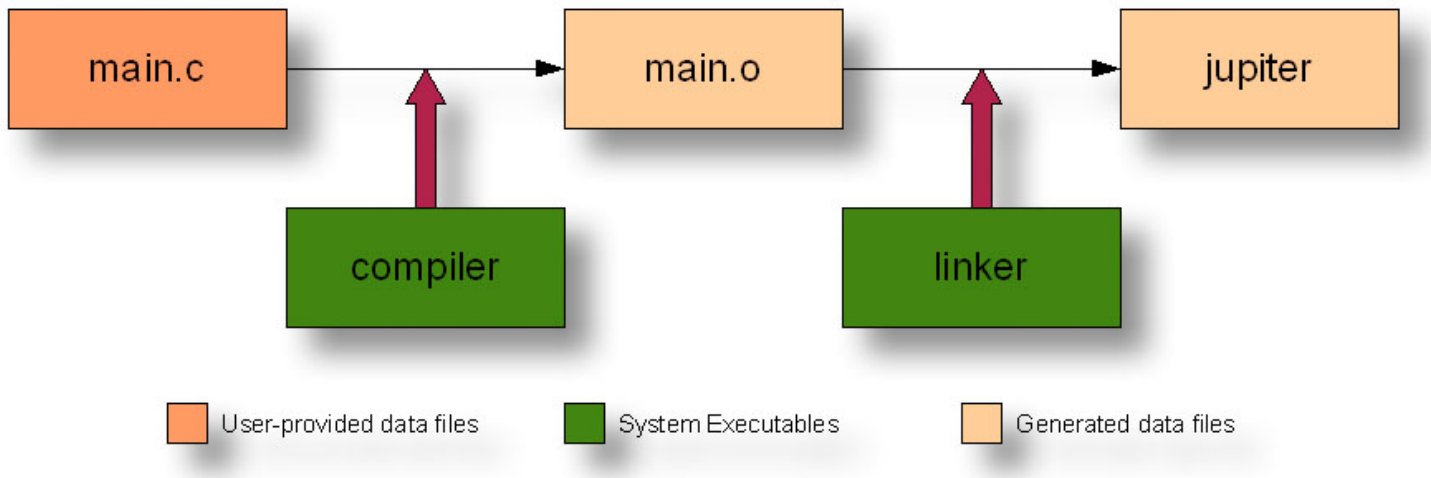
Compilation séparée

```
// int_math.cc
#include"int_math.hh"

namespace persolib{
  unsigned int sqrt(const unsigned int& n){
    unsigned int root=0, rem=0;
    unsigned int shift=8*sizeof(int)-2;
    unsigned int mask= (3u)<<shift;
    while(mask>0){
      rem<<=2; rem|=(n&mask)>>shift;
      shift-=2;  mask>>=2; root<<=1;
      if((root<<1) < rem){
        rem-=((root<<1)|1u); root|=1u;
      }
    }
    return root;
  }
  //...
}//end of namespace persolib
```

Compilation séparée

```
barbalala> g++ -c main.cc
barbalala> g++ -c int_math.cc
barbalala> g++ -o main main.o int_math.o
```



Compilation séparée

- Une variable globale est considérée comme externe (accessible à partir d'un autre fichier), sauf si elle est déclarée `const` ($\neq$ C) ou `static`.
- On peut utiliser des variables définies dans un binaire issu du C :

```
extern "C" int i;
extern "C" void my_function();
extern "C" {
  #include<my_header.h>
}
```

C et C++

On peut écrire des headers à la fois pour C et C++.

```
#ifndef MY_C_CPP_HEADER_H
#define MY_C_CPP_HEADER_H

#ifdef __cplusplus
extern "C" {
  #endif /*__cplusplus*/
  void foo();
  #ifdef __cplusplus
}
#endif /*__cplusplus*/
#endif /*MY_C_CPP_HEADER_H*/
```