Classe non génériques

.h

.cpp

↓ gcc -c

.o

allocation

Classe générique (template)

artificiel

pas d'include

#include ... .hpp

.h

.hpp

implémentation

T ?

on ne peut pas allouer

Piles

Pop ⟶ if (! est_vide ( ))

⋮ ⤳ pop.

T pop ( )

retourne la donnée de tête

else ⟶ pile vide / pop ?

? ⟶ T à ~~renvoyer~~ ?

stop ET erreur

↓

exception

throw ⟶ stop l'exé

⟶ renvoie 1 erreur ↓ ...

throw (...)

basic string

(+) classe

↓

Méthode - - - -

Fonction ex ⤳ **pop**

if ...

else throw (ex...)

⟶ Utilisée ds du code (ex : **main**)

⋮

try

sp. pop ( ) ;

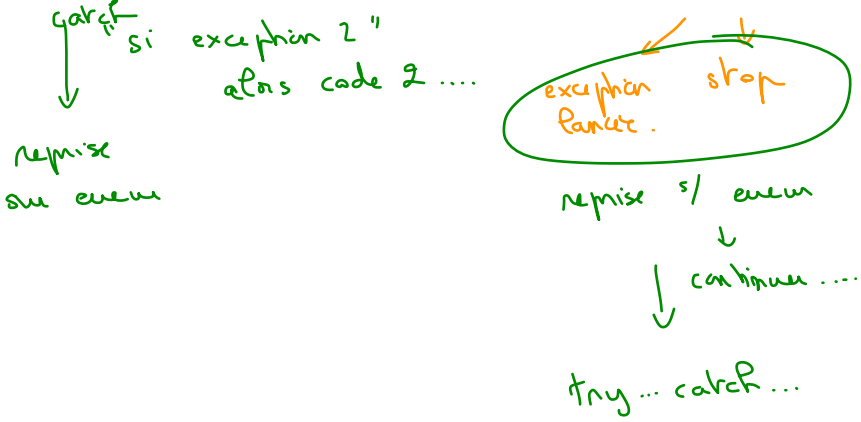
① (~~x~~ ⤳ ~~~~~

② ⤳ catch

↑ " si exception 1 " alors cod 1

① si la fct se déroule bien ↓ continuation instr. suiv.

② ⤳ si pile vide ↓ exception



classe exception.h (base)

```
class MyException {
protected:
    string _mess ;

public:
    MyException(const string &s) : _mess (s) {} ;

    virtual ostream & what (ostream & flot = cerr) const { return flot << _mess << endl ; };
} ;
```

erreur std ~> string.

Dans Vector ~> exception spécifique pour le "débordement"

→ set_item (i), get_item (i), t[i]

Dans ces exception ~> ① fct qui fait l'erreur / ② i / ③ taille_tab.

3 data

string

on ajoute 2 données pour stocker ces infos s/ débordement.

```
class MyExceptionVectorOut : public MyException {
protected:
    int _i, _taille ;
public:
    MyExceptionVectorOut(const string &s, int i, int taille) : MyException (s), _i (i), _taille (taille) {} ;
    ostream & what (ostream &flot = cerr) const {
        flot << _mess << " : accès hors limites à l'indice " << _i << " sur " << _taille << endl ;
        return flot ;
    } ;
} ;
```

Dans chaque fct provoquant ce type d'erreur : throw

```
int MyVector::get_item (const int i)
{
    nb_get++ ;
    if (i < _taille)
        return _data[i] ;
    else
        throw (MyExceptionVectorOut("Dans get_item", i,
_taille)) ;
}
```

e

```
void MyVector::set_item(const int i, const int n)
{
        if (i < _taille)
        {
                _data[i] = n ;
                nb_set++ ;
        }
        else
        {
                throw(MyExceptionVectorOut("Dans set_item", i,
_taille)) ;
        }
}
```

Partie utilisation du cas fct : (ex : main)

Pile erreur(s) top / pop s/ pile vide → on crée 1 classe
throw → MyExceptionEmptyStack

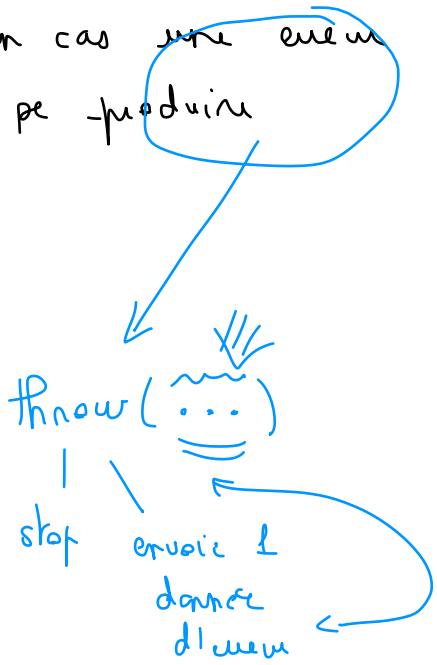
```
try
{   Pile<int> p;
    Vector v(20) ;
    int i ;
    cout << "entrez une case : " ;
    cin >> i ;
    cout << "valeur : " << v.get_item(i) ;
    p.pop();
}
catch (MyExceptionVectorOut &e)
{
    e.what() ;
    // Ce qu'il faut faire pour continuer ...
}
catch(...) // Default
{
    cout << "Erreur non géreée " << endl ;
}

catch (MyExceptionEmptyStack &e)
{   e.what();
    ...
}
```

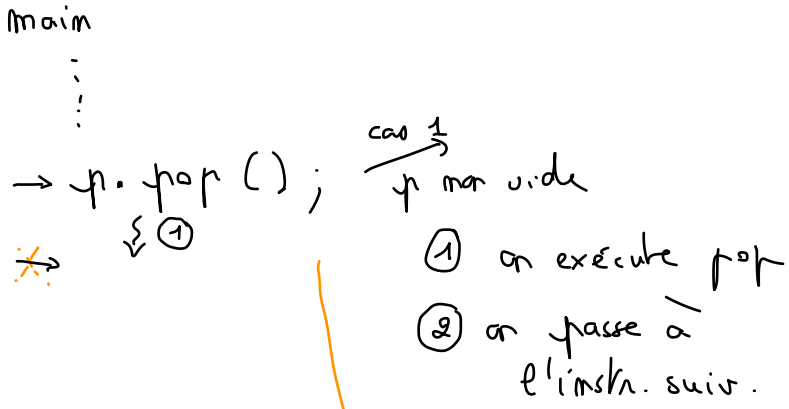
Code (normal) si pas d'erreurs ...

Reprises sur les différentes erreurs possibles.

Coder 1 méthode où dans un cas une erreur peut se produire



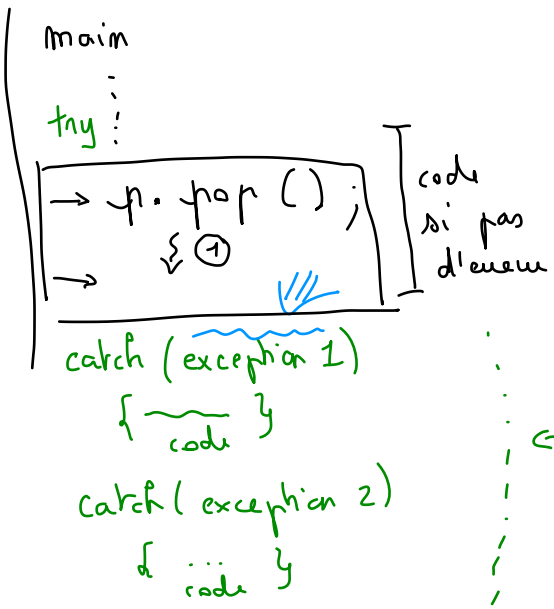
Ex de la main qui utilise / appel de la pile



reprise par erreur ....

```
try
    code "normal"
catch.
    gestion de tous les cas d'erreur
{
catch ( .. )
si erreur 1
{ → alors code 1 }

si erreur 2
    → alors code 2 ....
```



(passage par) référence

classes

Heritage / polymorphisme

classe fille —hérite→ classe mère

fille —enrichit→ mère

vecteur colorés

vecteur
couleur

~fille → exécute par code
auto → exécute ~mère X

Généricité (template)

↳ passer des variables de (T) type (+ var. de classe, de valeurs ....)

classe → dépend d'1 var. de type

≠

ex: pile de (T)
↳ variable de type

↓

T spécifié à l'instanciation (main)

main:

```
Pile<int> p1;
Pile<Vecto> p2;
```

```
template <typename T>
```

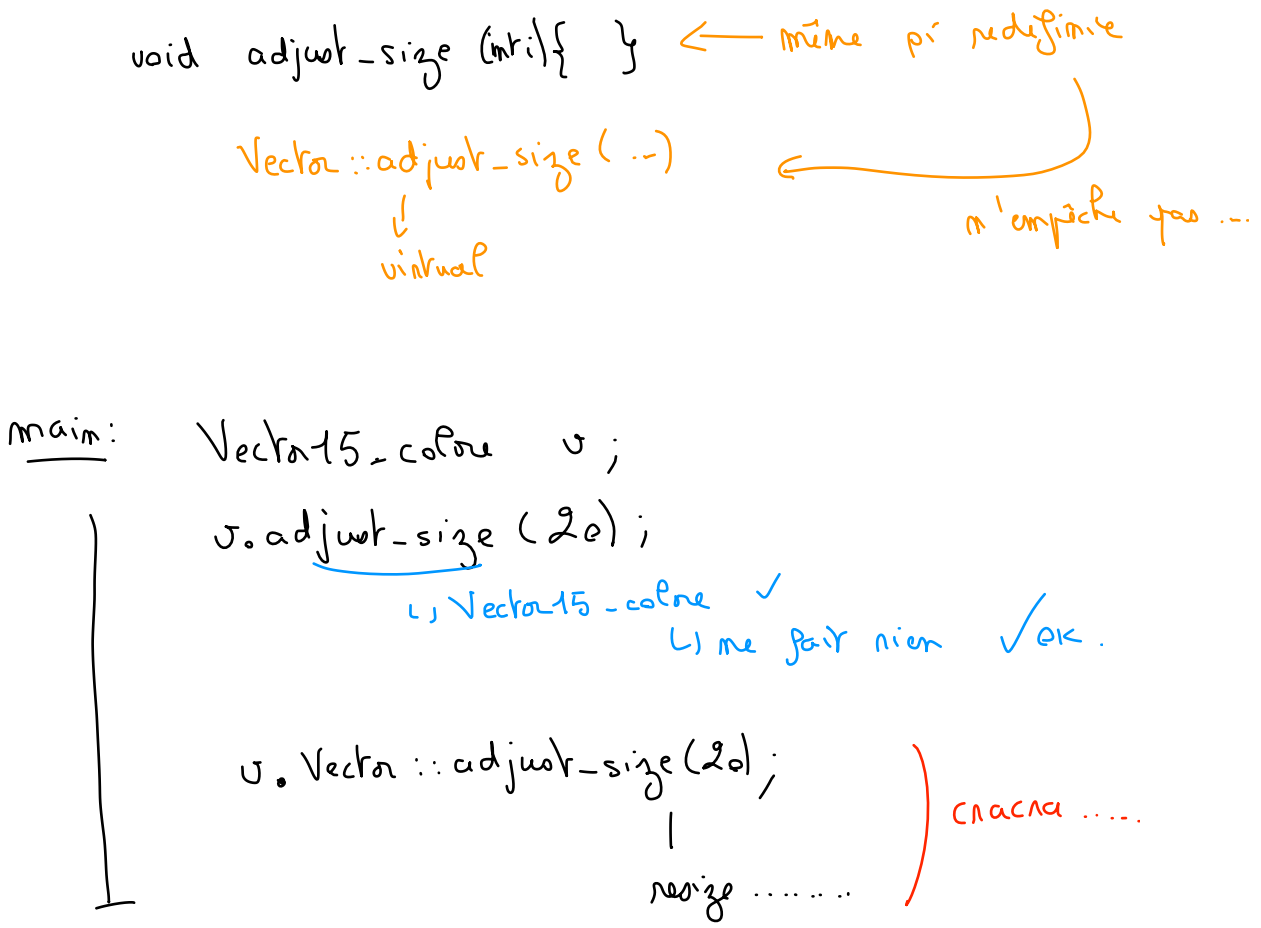
↓

dict. d'1 var. de type T.

```
class Vector15_color : public Vecto
{ private:
    char _color[3] ; //RGB.

  public:
    void afficher () { Vecto::afficher () ;
                       cout << _color[0] << "/" << _color[1] ....
    }
```

afficher de Vecto ....


void adjust_size (int){ } ← même si redéfinie
Vector::adjust_size (..)
virtual
n'empêche pas ...
main:
Vector15_colore v;
v.adjust_size (20);
↳ Vector15_colore ✓
↳ ne fait rien Ok.
v.Vector::adjust_size(20);
resize .....
craca ....