

# Erreur dans un constructeur C++

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



*Préservez l'environnement,  
n'imprimez pas ce document*

## TABLE DES MATIERES

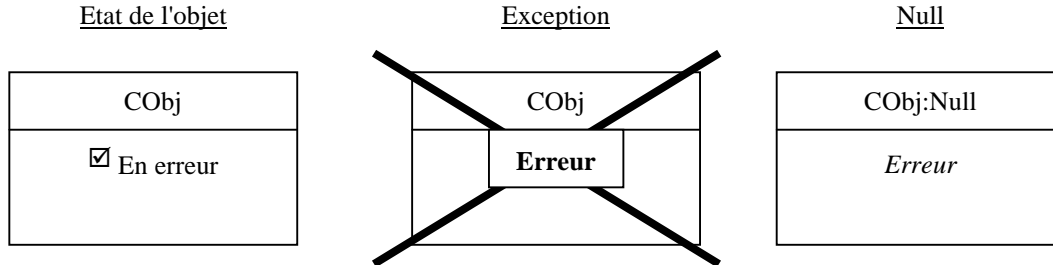
1.	Les erreurs dans les constructeur .....	3
1.1	État dans l'objet.....	3
1.2	Contexte de l'objet.....	3
1.3	Exceptions.....	3
1.4	Instance NULL.....	4
1.5	Encapsulation dans une classe dérivée.....	5
2.	Traits de caractères .....	5
3.	Exception dans le constructeur de copie .....	7
4.	Exception dans un template.....	8

## *Avant propos*

*Ce document explique comment gérer les erreurs dans un constructeur C++.*

### 1. LES ERREURS DANS LES CONSTRUCTEURS

Le constructeur est la seule méthode ne retournant pas de valeur. Pendant son invocation, il peut y avoir des erreurs. Comment les signaler à l'appelant ? Plusieurs approches sont possibles.



#### 1.1 État dans l'objet

Une première technique consiste à ajouter un état à l'objet indiquant si celui-ci est correctement construit.

```
class CObj
{
public:
    CObj() : _status(false)
    { // ...
      _status=true;
    }
    bool isOk() const
    { return _status; }
};
```

#### 1.2 Contexte de l'objet

Une autre approche consiste à utiliser les éléments déjà présents dans l'objet.

```
class CFile
{
public:
    CFile(const char* name)
    : _st(NULL)
    { _st=fopen(name, "w"); }
    ~CFile()
    { fclose(_st); }
    int isOk() const
    { return (_st!=NULL); }
};
```

Le problème principal de cette approche, est que chaque méthode doit s'assurer que l'objet est correct avant d'en utiliser les éléments. Cela entraîne énormément de redondances à l'exécution.

```
void CFile::methode()
{ if (!isOk()) return;
  // ...
}
```

Il ne faut pas oublier qu'un objet peut être créé de façon temporaire par le compilateur pour y appeler une méthode ou un opérateur. La méthode doit s'assurer que l'objet est correct avant d'être exécutée. Il peut être prudent d'ajouter en tête de celle-ci une ligne `assert(isOk());` pour ne pas pénaliser le programme en mode exploitation.

#### 1.3 Exceptions

La solution la plus pratique est d'utiliser les exceptions. Une erreur lors d'un constructeur générera une exception.

```
class CObj
{
public:
    CObj()
    { // ...
      if (erreur) throw xError();
    }
};
```

## 1.4 Instance NULL

Pour détecter un objet en erreur, il est souvent utile d'avoir une instance particulière correspondant à une version de l'objet en *erreur*. Le C utilise cette technique pour indiquer qu'un pointeur est erroné. La valeur `NULL` indique qu'un pointeur n'est pas valide. Les allocateurs mémoire garantissent qu'aucune allocation ne sera faite à cette adresse particulière. `EOF` indique également un caractère invalide. Une fonction peut retourner cette valeur pour informer de son comportement anormal. Cela permet d'éviter de retourner deux variables : un caractère (ou un pointeur) et un code d'erreur. L'objet retourné, pointeur ou caractère, informe en lui-même de l'erreur possible.

Avec une classe, il peut être utile d'avoir ce même type de comportement. Une instance particulière peut informer de l'erreur d'une fonction. Par exemple, prenons une classe `CDate`. Nous désirons avoir une date particulière pour signaler une date erronée.

```
class CDate
{ int _jour, _mois, _annee;
public:
    CDate(const char* str);
    CDate(int jour, int mois, int annee)
    : _jour(jour), _mois(mois), _annee(annee)
    { assert((jour>0) && (mois>0) && (annee>1900)); }
    int operator ==(const CDate& x) const
    { return ((_jour==x._jour) &&
              (_mois==x._mois) &&
              (_annee==x._annee)); }
}
static const CDate Null;
};
```

Nous désirons créer l'instance statique `CDate::Null` avec une valeur que ne peut pas fournir l'utilisateur de la classe. Pour cela, il faut ajouter un constructeur particulier en protégé, afin de pouvoir construire cette instance sans le test de post condition.

```
class CDate
{ int _jour, _mois, _annee;
protected:
    enum TError { error };
    CDate(TError)
    : _jour(0), _mois(0), _annee(0)
    { }
public:
    static const CDate Null;
    //...
};
```

Il est alors possible de construire l'instance `CDate::Null` à l'aide de ce constructeur. Le type de paramètre permet de sélectionner le constructeur désiré.

```
const CDate CDate::Null=CDate::error;
```

Maintenant, il est possible de retourner cette valeur lors d'une erreur d'une fonction.

```
CDate f()
{ //...
  if (erreur) return CDate::Null;
  return date;
}

void main()
{ if (f()==CDate::Null) cout << "Erreur dans f()" << endl;
}
```

Le test s'effectue sur cette valeur à l'aide de l'opérateur `==()` de `CDate`. Étant donné que la seule instance pouvant avoir cette valeur est `CDate::Null`, il n'y a pas de confusion possible.

L'objet `CDate::Null` est constant car il ne faut pas laisser la possibilité à l'utilisateur de modifier cet objet. Il faut interdire une écriture comme

```
CDate::Null=CDate();
```

En déclarant cette instance constante, il n'est pas nécessaire d'ajouter une précondition dans les services pour interdire sa modification. Le compilateur est là à cet effet.

Il est parfois utile qu'il ne soit pas possible d'avoir plusieurs objets ayant simultanément la valeur `Null`. Il faut dans ce cas interdire de copier l'instance `Null` dans un autre objet.

```

class CDate
{ //...
    CDate(const CDate& x)
    { assert(&x!=&CDate::Null);
    }
    CDate& operator =(const CDate& x)
    { assert(&x!=&CDate::Null);
    //...
    }
    bool operator ==(const CDate& x) const
    { if ((&x==&CDate::Null) && (this==&CDate::Null)) return true;
    // ...
    }
};

```

Il faut ajouter des pré-conditions pour interdire la copie de `CDate::Null`. Il est alors possible d'optimiser l'opérateur de comparaison en vérifiant l'adresse particulière de `CDate::Null`.

### 1.5 Encapsulation dans une classe dérivée

S'il n'existe pas de combinaison de valeur pouvant être associée à la valeur `NULL`, il faut écrire une classe dérivée de `CDate`. Par exemple, si toutes les combinaisons de dates sont valides, l'objet `CDate` ne peut pas, en lui-même, indiquer une date invalide.

```

class CTstDate : public CDate
{ bool _erreur;
public:
    enum TError { error };
public:
    CTstDate(const CDate& x)
    : CDate(x), _erreur(false)
    { }
    CTstDate(TError)
    : CDate(1,1,1990), _erreur(true)
    { }
    operator bool() const
    { return _erreur; }
    bool operator !() const
    { return !_erreur; }
};

```

Une fonction ou une méthode désirant retourner une date ou une erreur, doit retourner un objet du type `CTstDate`.

```

CTstDate f()
{ //...
    if (erreur) return CTstDate(CTstDate::error);
    return CTstDate(date);
}
void main()
{ if (f()) cout << "Erreur dans f()" << endl;
}

```

C'est la même approche qu'utilise la fonction `C fgetc()` qui retourne un entier et non un caractère pour pouvoir retourner le code `EOF`. L'entier étant plus grand qu'un caractère, il peut contenir toutes les valeurs possibles d'un caractère. En quelque sorte, nous avons :

```

// Ce n'est pas du C++ !
class int : public char
{ ... };

```

Toutes les valeurs d'un caractère sont possibles. Il faut alors un type plus grand pour pouvoir contenir la valeur particulière `EOF`.

## 2. TRAITS DE CARACTERES

Pour pouvoir écrire un `template` avec une méthode retournant la valeur d'erreur d'un objet, il faut connaître son type. Une instance en erreur n'a pas forcément la même taille qu'une instance sans erreur. Par exemple, la valeur `EOF` qui indique un caractère incorrect, est stockée dans un entier et non dans un `char`. En effet, toutes les valeurs possibles d'un `char` sont valides. Il a fallu trouver une valeur supplémentaire à mettre dans un type de taille supérieure à `char`. La fonction `fgetc()` retourne un entier et non un `char` pour pouvoir vérifier la valeur `EOF`. Un `template` voulant retourner une valeur d'erreur, doit connaître les deux types possibles à manipuler : le type de base, et le type permettant de retourner une erreur. Imaginons que nous voulons écrire un `template` du type « pile » dont la méthode `pop()` retourne l'objet paramètre ou une valeur d'erreur lorsque la pile est vide. Une pile de caractère retournera le caractère présent dans celle-ci ou `EOF` si la pile est vide.

```

template <class T>
class CStack
{ T _tab[10];
  int _nb;
public:
    CStack() : _nb(0)
    { }
};

```

```

void push(const T& data)
{ _tab[_nb++]=data;
}
<Type?> pop()
{ return (_nb) ? _tab[--_nb] : <Valeur?>;
};

```

Il faut connaître le type capable de recevoir la valeur d'erreur, et il faut connaître cette valeur d'erreur. Voici plusieurs combinaisons acceptables pour ce `template` :

Type de base	Type d'erreur	Valeur d'erreur
<code>char</code>	<code>int</code>	<code>EOF</code>
<code>wchart_t</code>	<code>wint_t</code>	<code>WEOF</code>
<code>int</code>	<code>int</code>	<code>INT_MAX</code>
<code>void*</code>	<code>void*</code>	<code>NULL</code>
<code>CDate</code>	<code>CDate</code>	<code>CDate::Null</code>
<code>CDate</code>	<code>CTstDate</code>	<code>CTstDate(error)</code>

Nous allons déclarer un `template` qui devra contenir les informations manquantes.

```

template <class T>
struct TError { };

```

Ce `template` est volontairement vide. Il permet simplement de signaler sa signature. Pour chaque type d'objet, il faudra rédiger une version spécifique de ce `template`.

```

struct TError<char>
{ typedef char type_base;
  typedef int type_error;
  static inline type_error error()
  { return EOF; }
};

```

```

struct TError<int>
{ typedef int type_base;
  typedef int type_error;
  static inline type_error error()
  { return INT_MAX; }
};

```

```

struct TError<CDate>
{ typedef CDate type_base;
  typedef CTstDate type_error;
  static inline type_error error()
  { return type_error(error); }
};

```

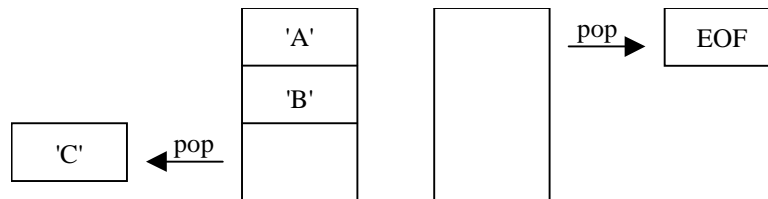
Le `template` de la pile, utilisera `TError` pour générer le code correspondant au type utilisé.

```

template <class T>
class CStack
{ T _stack[10];
  int _sp;
  typedef TError<T>::type_error type_error;
public:
  CStack() : _sp(0)
  { }
  void push(const T& data)
  { _stack[_sp++]=data;
  }
  type_error pop()
  { return (_nb) ? _stack[--_sp] : TError<T>::error();
  }
};

```

Une instance de `CStack<char>::pop()` retournera un entier.



Une instance de `CStack<CDate>::pop()` retournera un objet `CTstDate`. Cette technique a été utilisée par le comité ANSI/ISO C++ pour rédiger les flux paramétrés.

### 3. EXCEPTION DANS LE CONSTRUCTEUR DE COPIE

Certaines situations sont difficiles à gérer. Une exception peut être appelée lors d'un constructeur de copie. Un constructeur de copie peut être appelé lors du retour d'une fonction. Il est difficile dans ce cas de maîtriser l'exception. Il n'est pas possible de la capturer.

```
class CObj
{ /*...*/
public:
    CObj();
    CObj(const CObj& x);
    ~CObj();
};

class CStackObj
{ int _sp;
  int _max;
  CObj* _tab;
  // ...
public:
    CStackObj();
    ~CStackObj();
    void push(const CObj& x);
    CObj pop();
};

CStackObj::CStackObj() : _sp(0), _max(0)
{ _tab=(CObj*)new char[sizeof(CObj)*10];
}

CStackObj::~CStackObj()
{ for (int i=_max-1;i>=0;--i)
  { _tab[i].CObj::~CObj();
  }
  delete [] (char*)_tab;
}

void CStackObj::push(const CObj& x)
{ assert(_sp<10);
  _tab[_sp++].CObj::CObj(x);
  // ou new (&_tab[_sp++]) CObj(x);
  if (_sp>_max) _max=_sp;
  assert(_sp<10);
}

CObj CStackObj::pop()
{ return _tab[--_sp];
}
```

La méthode `CStackObj::pop()` retourne un objet extrait de la pile. Elle décrémente le pointeur de pile simultanément. Si une exception arrive dans le constructeur de copie de l'objet `CObj`, l'objet ne sera pas copié, mais le pointeur de la pile sera décrément. Le code de la méthode `pop()` est généré comme suit :

```
void CStackObj::pop(const CStackObj * const this,CObj* _ret)
{ --this->_sp;
  _ret->CObj::CObj(_tab[_sp]); // cctr, exception possible
}
```

Dans le cas d'une exception, `--this->_sp` a été exécuté, ce que l'on cherche à éviter. Si le pointeur de la pile est décrément alors que la copie de l'élément a échoué, celui-ci est définitivement perdu. Comment trouver un moyen de modifier cet index, si et seulement si, la copie a réussi ?

Il faut modifier `sp` après la copie. Dans la méthode, il n'est pas possible d'écrire un traitement exécuté après le `return`. Pour résoudre ce cas, il faut créer un objet supplémentaire permettant de modifier `sp` après la copie.

```
class CStackObjPop : public CObj
{ public:
```

```

    CStackObjPop(CStackObj& stack,CObj& obj)
    : CObj(obj)
    { --stack._sp; }
};

```

La méthode `pop()` retourne un objet de ce type qui est dérivé de `CObj`. Le constructeur de cet objet garde une relation avec l'objet `CStackObj`. C'est lors de l'appel du constructeur de copie de `CObj` que l'exception peut intervenir. Si l'objet `CStackObjPop` est correctement créé, l'utilisateur peut l'utiliser comme s'il s'agissait d'un objet `CObj`. Il peut écrire « `s.pop().a` » ou copier l'objet : « `CObj x=s.pop();` ». Si le constructeur de copie de `CObj` génère une exception, l'instruction « `--stack.sp` » ne sera pas exécutée.

Il faut modifier l'objet `CStackObj` comme suit :

```

class CStackObjPop;
class CStackObj
{ // ...
public:
    friend class CStackObjPop;
    CStackObjPop pop();
};

CStackObjPop CStackObj::pop()
{ return CStackObjPop(*this,_tab[_sp-1]);
}

```

## 4. EXCEPTION DANS UN TEMPLATE

Un `template` utilise des méthodes de l'objet générique. Celles-ci peuvent être amenées à générer une exception. Il faut, lors de la rédaction d'un `template`, identifier tous les appels des méthodes de l'objet générique et se poser la question du comportement du `template` si une exception arrive. Cela comprend, les constructeurs, les opérateurs, les conversions, et surtout, le constructeur de copie. Il faut penser à tout ce que le compilateur fait derrière « notre dos » pour identifier correctement l'appel de toutes les méthodes de l'objet générique.

```

template <class T>
class CList
{ struct CNode
  { CNode* _next;
    T      _obj;
    CNode(CNode* next,const T& obj)
    : _next(next), _obj(obj) {} // cctr de T
  };
  CNode* _first;
public:
    CList() : _first(NULL) {}
  CList<T>& operator +=(const T& x)
  { _first=new CNode(_first,x);
    return *this;
  }
  CList<T>& operator --(const T& x)
  { CNode* opt=NULL;
    for (CNode* pt=_first;pt!=NULL;opt=pt,pt=pt->_next)
    { if (pt->_obj==x)// operator == de T
      { (opt==NULL) ? _first : opt->_next=pt->_next;
        delete pt; // dtr de T
      }
    }
    return *this;
  }
};

```

Chaque utilisation d'une méthode de `T` peut générer une exception. Cette classe doit être entièrement réécrite pour en tenir compte.