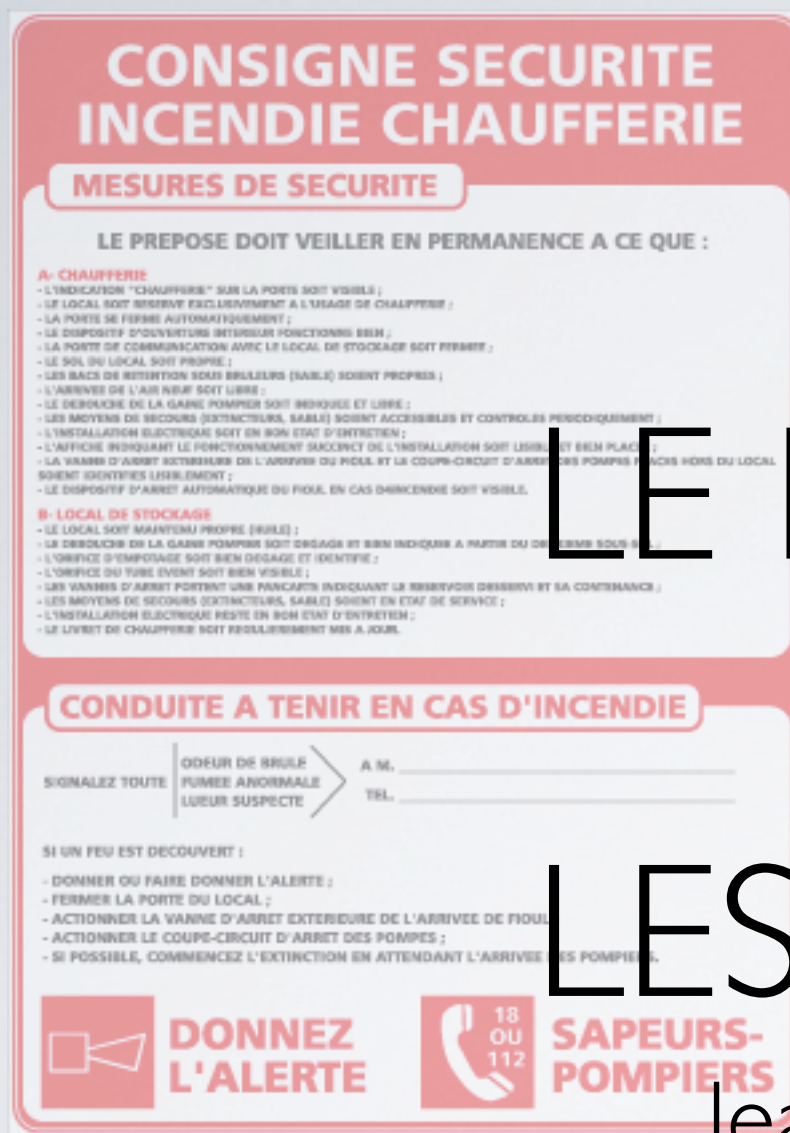


LE LANGAGE C++ MASTER I LES EXCEPTIONS ?

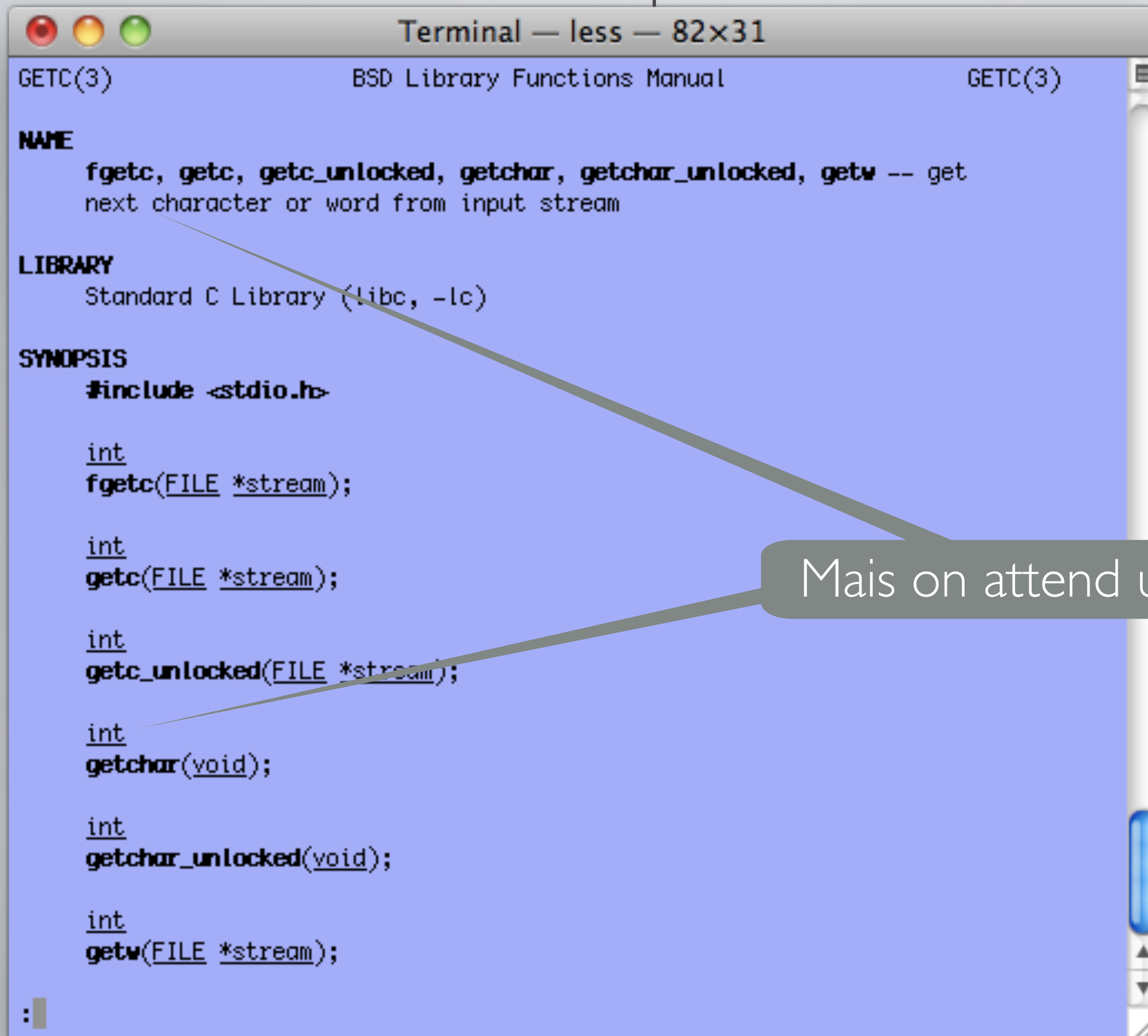
Jean-Baptiste.Yunes@u-paris.fr
U.F.R. d'Informatique
Université de Paris

11/2021



LE TRAITEMENT DES ERREURS SANS LES EXCEPTIONS

- Les bizarreries lorsque le mécanisme n'existe



Terminal — less — 82x31

GETC(3) BSD Library Functions Manual GETC(3)

NAME
fgetc, getc, getc_unlocked, getchar, getchar_unlocked, getw -- get next character or word from input stream

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <stdio.h>

int
fgetc(FILE *stream);

int
getc(FILE *stream);

int
getc_unlocked(FILE *stream);

int
getchar(void);

int
getchar_unlocked(void);

int
getw(FILE *stream);

␣

Mais on attend un char non ?

- Les bizarreries lorsque le mécanisme n'existe

Terminal — less — 82x31

OPEN(2) BSD System Calls Manual OPEN(2)

NAME

`open` -- open or create a file for reading or writing

SYNOPSIS

```
#include <fcntl.h>
```

```
int  
open(const char *path, int oflag, ...);
```

DESCRIPTION

The file name specified by `path` is opened, as specified by the argument `oflag`; returned to the calling process.

The `oflag` argument may indicate that the file does not exist (by specifying the `O_CREAT` flag), which requires a third argument `mode_t mode` as described in `chmod(2)` and `chmod(1)` value (see `umask(2)`).

The flags specified are formed by `or'ing` the following:

<code>O_RDONLY</code>	open for reading only
<code>O_WRONLY</code>	open for writing only
<code>O_RDWR</code>	open for reading and writing
<code>O_NONBLOCK</code>	do not block on open
<code>O_APPEND</code>	append on each write
<code>O_CREAT</code>	create file if it does not exist

Terminal — less — 82x31

The system imposes a limit on the number of file descriptors that can be held open simultaneously by one process. `Getdtablesize(2)` returns the current system limit.

RETURN VALUES

If successful, `open()` returns a non-negative integer, termed a file descriptor. It returns `-1` on failure, and sets `errno` to indicate the error.

ERRORS

The named file is opened unless:

[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	The required permissions (for reading and/or writing) are denied for the file.
[EACCES]	<code>O_CREAT</code> is specified and the file does not exist, and the directory in which it is to be created does not permit writing.
[EACCES]	<code>O_TRUNC</code> is specified and write permission is denied.
[EAGAIN]	<code>path</code> specifies the slave side of a locked pseudo-terminal device.
[EDQUOT]	<code>O_CREAT</code> is specified, the file does not exist, and the user's quota is exceeded.

Mais on attend un descripteur!


```
int d1, d2;

d1 = open("glouglou", O_RDONLY);
if (d1 == -1) {
    switch(errno) {
        case EACCESS:
            fprintf(stderr, "problème d'accès à glouglou\n");
            exit(EXIT_FAILURE);
        case EISDIR:
            fprintf(stderr, "glouglou est un répertoire\n");
            return -1;
    }
}
d2 = open("tagada", O_WRONLY);
if (d2 == -1) {
    switch(errno) {
        case ELOOP:
            close(d1);
            fprintf(stderr, "problème de résolution\n");
            break;
    }
}
```

**On mélange tout!
et la partie fonctionnelle du code
et la partie traitement des erreurs!!!**

- La vision traditionnelle du déroulement d'un programme tel que projeté dans le monde *réel* :

```
// programme de préparation du repas
sortir les patates du placard
si le feu prend dans la maison {
    sortir dehors
    appeler les pompiers
    ...
}
aller chercher l'épluche-légumes
si le feu prend dans la maison {
    sortir dehors
    appeler les pompiers
    ...
}
éplucher une patate
si le feu prend dans la maison {
    lâcher patate et épluche-légumes
    sortir dehors
    appeler les pompiers
    ...
}
...
```

- La vision traditionnelle du déroulement d'un programme tel que projeté dans le monde

```
// programme de préparation du repas
sortir les patates du placard
si le feu prend dans la maison {
    sortir dehors
    appeler les pompiers
    ...
}
aller chercher l'épluche-légumes
si le feu prend dans la maison {
    sortir dehors
    appeler les pompiers
    ...
}
éplucher une patate
si le feu prend dans la maison {
    lâcher patate et épluche-légumes
    sortir dehors
    appeler les pompiers
    ...
}
...
```

Programming
FOR
DUMMIES®

CERTIFIÉ
MODE PARANOÏAQUE

GARANTI INÉFFICACE

VU EN COURS DE C!!!

LE TRAITEMENT DES ERREURS AVEC LES EXCEPTIONS

- Idée : séparer **dans le code** :
 - les instructions qui représentent la partie *fonctionnellement intéressante* du programme
 - des instructions qui servent à traiter/corriger les erreurs rencontrées et qui empêchent de continuer normalement

- Les exceptions dans le monde réel :

```
// programme de préparation du repas
sortir les patates du placard
aller chercher l'épluche-légumes
{épluchage}
    éplucher une patate
    s'il reste une patate non épluchée
        alors continuer l'{épluchage}
sortir friteuse
remplir friteuse d'huile de friture
allumer le gaz
couper les patates en frites
attendre que l'huile atteigne 180°
baigner les frites dans l'huile
attendre que les frites soient cuites
sortir les frites
attendre que l'huile atteigne 180°
baigner les frites dans l'huile
attendre que les frites soient grillées
sortir les frites
```

Consignes en cas d'incendie

sauter par la fenêtre
appeler au secours
hurler à la mort
pleurer tout son saoul
bénir les pompiers
appeler son assureur
pleurer encore

- Les exceptions dans le monde réel :

```
// programme de préparation du repas
sortir les patates du placard
aller chercher l'épluche-légumes
{épluchage}
    éplucher une patate
    s'il reste une patate non épluchée
        alors continuer l'{épluchage}
sortir friteuse
remplir friteuse d'huile de friture
allumer le gaz
couper les patates en frites
attendre que l'huile atteigne 180°
baigner les frites dans l'huile
attendre que les frites soient cuites
sortir les frites
attendre que l'huile atteigne 180°
baigner les frites dans l'huile
attendre que les frites soient grillées
sortir les frites
```

VU EN COURS DE C++

Consignes en cas d'incendie

sauter par la fenêtre
appeler au secours
hurler à la mort
pleurer tout son saoul
bénir les pompiers
appeler son assureur
pleurer encore

intel-
ligence
inside

**CERTIFIÉ
CONFIDENTIEL**

GARANTIE 100% EFFICACE

- Un mécanisme de contrôle du flux d'exécution
 - soit une fonction réussit, lors de son invocation, à réaliser correctement ses calculs alors :
 - la fonction **termine** et **renvoie** une valeur
 - soit quelque chose l'empêche de continuer dans des conditions *normales* et :
 - on **sort précipitamment** de la fonction **en erreur**

- il y a donc deux mécanismes d'exécution :
 - le flux normal de l'exécution
 - le flux de récupération des erreurs
- et un moyen de basculer de l'un à l'autre :
 - de normal à erreur par la levée d'une exception (`throw`) (`stack unwinding`)
 - de erreur à normal par la capture d'une exception (`catch`)

- une exception
 - représente logiquement une erreur détectée de façon synchrone durant l'exécution du programme
 - représente la fiche d'incident qui contient les informations nécessaire pour permettre à l'hypothétique partie adéquate du code de *réparer la faute* initiale



DÉCLENCHER UNE
EXCEPTION

- Le déclenchement s'effectue par :

`throw expression;`

- La valeur de l'*expression* représente l'erreur que l'on souhaite indiquer, toute expression valide est autorisée, en particulier, l'expression de construction d'un objet temporaire :

`throw UneClasse(...);`

- Au déclenchement d'une exception (à la levée d'une exception) l'exécution normale s'interrompt et une recherche de reprise sur erreur est effectuée :
 - le *runtime* remonte la pile des appels :
 - en détruisant les objets temporaires (stack unwinding)
 - et jusqu'à :
 - trouver une reprise sur erreur adéquate
 - l'appel premier du `main()` auquel cas l'exécution s'arrête, et indique qu'une erreur est survenue mais n'a pas été capturée

```
void g(int v) {  
    cout << "debut g()" << endl;  
    if (v==0) throw 666;  
    cout << "fin g()" << endl;  
}
```

```
void f() {  
    cout << "debut f()" << endl;  
    g(3);  
    g(0);  
    cout << "fin f()" << endl;  
}
```

```
int main() {  
    cout << "debut main()" << endl;  
    f();  
    f();  
    cout << "fin main()" << endl;  
    return 0;  
}
```

avec un entier

```
[Trotinette:~] yunes% ./test  
debut main()  
debut f()  
debut g()  
fin g()  
debut g()  
terminate called after  
throwing an instance of  
'int'  
Abort  
[Trotinette:~] yunes%
```

```
class A {};  
  
void g(int v) {  
    cout << "debut g()" << endl;  
    if (v==0) throw A();  
    cout << "fin g()" << endl;  
}  
  
void f() {  
    cout << "debut f()" << endl;  
    g(3);  
    g(0);  
    cout << "fin f()" << endl;  
}  
  
int main() {  
    cout << "debut main()" << endl;  
    f();  
    f();  
    cout << "fin main()" << endl;  
    return 0;  
}
```

avec une instance

```
[Trotinette:~] yunes% ./test  
debut main()  
debut f()  
debut g()  
fin g()  
debut g()  
terminate called after  
throwing an instance of 'A'  
Abort  
[Trotinette:~] yunes%
```

```
class A {};  
  
void g(int v) {  
    cout << "debut g()" << endl;  
    if (v==0) throw new A;  
    cout << "fin g()" << endl;  
}  
  
void f() {  
    cout << "debut f()" << endl;  
    g(3);  
    g(0);  
    cout << "fin f()" << endl;  
}  
  
int main() {  
    cout << "debut main()" << endl;  
    f();  
    f();  
    cout << "fin main()" << endl;  
    return 0;  
}
```

avec un pointeur

```
[Trotinette:~] yunes% ./test  
debut main()  
debut f()  
debut g()  
fin g()  
debut g()  
terminate called after  
throwing an instance of 'A*'  
Abort  
[Trotinette:~] yunes%
```


- Important :
 - le concepteur d'une fonction qui décide de lever une exception ne doit pas se préoccuper de savoir :
 - si l'erreur sera récupérée
 - si elle est récupérée, où l'est-elle ?
 - sa seule préoccupation doit être de fournir des informations appropriées sur l'erreur et qui permettront éventuellement de la corriger...

```
int *uneFonctionCommeUneAutre(int v) {  
    int *t = new int[v];  
    if (t==0)  
        throw ProblemeDAllocation(v);  
    return t;  
}
```

```
int uneAutreFonction(char *fichier) {  
    int descripteur;  
    descripteur = open(fichier,O_RDONLY);  
    if (descripteur==-1)  
        throw ProblemeFichier(fichier,errno);  
    return descripteur;  
}
```

CAPTURER UNE EXCEPTION



- un code faisant appel à des fonctions pouvant lever des exceptions peut :
- **ignorer celles-ci** en ne faisant rien de particulier (attention il ne devrait les ignorer que parce qu'il ne se sent pas en situation de pouvoir les corriger - pas par fainéantise)
- **tenter de corriger** certaines des erreurs qui pourraient se produire en prévoyant une porte de sortie lorsqu'elles se produisent. Ce mécanisme est appelé capture.

- la partie du code dans laquelle on désire tenter de récupérer les erreurs doit être incluse dans un bloc qualifié par `try` :

```
...  
try {  
    bloc d'instructions dans lequel  
    des exceptions peuvent se produire  
}  
...
```

- si une erreur se produit elle peut être récupérée dans l'un des blocs (qui suivent immédiatement) qualifié par `catch(Type e)` :

```
catch(TypeError erreur) {  
    bloc d'instructions de traitement  
    de l'erreur produite  
}
```

- la forme générale est donc :

```
try {  
    bloc d'instructions dans lequel  
    des exceptions peuvent se produire  
} catch(TypeError1 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur de TypeError1  
} catch(TypeError2 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur de TypeError2  
} ...  
...  
    catch(TypeErrorn erreur) {  
        bloc d'instructions de traitement  
        de l'erreur de TypeErrorn  
    }  
}
```

- par exemple la documentation indique :

Documentation de la bibliothèque COOLIB

Fonction d'allocation d'un tableau

Prototype

```
int *alloueTableau(int v);
```

Exceptions

ErreurAllocation (voir la documentation)

Fonction d'ouverture de fichier

Prototype

```
int ouvreFichier(char *fichier);
```

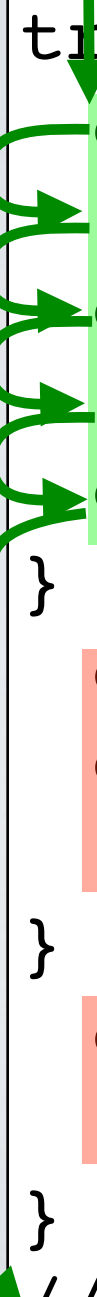
Exceptions

ErreurOuverture (voir la documentation)

- un usage possible est :

```
try {  
    cout << "bonjour tout le monde" << endl;  
    int *t = alloueTableau(100000);  
    cout << "jusqu'ici tout va bien" << endl;  
    int d = ouvreFichier("database.txt");  
    cout << "ici, c'est cool" << endl;  
} catch(ErreurAllocation a) {  
    cerr << "Houlala, problème d'allocation" << endl;  
    cerr << "Taille demandée=" << a.getSize() << endl;  
    //suite du traitement de l'erreur  
} catch(ErreurOuverture b) {  
    cerr << "Impossible d'ouvrir " << b.getName() << endl;  
    //suite du traitement de l'erreur  
}  
//suite du programme
```


- lors de l'exécution, si tout va bien :



```
try {  
    cout << "bonjour tout le monde" << endl;  
    int *t = alloueTableau(100000);  
    cout << "jusqu'ici tout va bien" << endl;  
    int d = ouvreFichier("database.txt");  
    cout << "ici, c'est cool" << endl;  
} catch(ErreurAllocation a) {  
    cerr << "Houlala, problème d'allocation" << endl;  
    cerr << "Taille demandée=" << a.getSize() << endl;  
    //suite du traitement de l'erreur  
} catch(ErreurOuverture b) {  
    cerr << "Impossible d'ouvrir " << b.getName() << endl;  
    //suite du traitement de l'erreur  
}  
//suite du programme
```

- lors de l'exécution, si une erreur se produit lors de l'appel à `alloueTableau(100000)` :

```
try {  
    cout << "bonjour tout le monde" << endl;  
    int *t = alloueTableau(100000);  
    cout << "jusqu'ici tout va bien" << endl;  
    int d = ouvreFichier("database.txt");  
    cout << "ici, c'est cool" << endl;  
} catch(ErreurAllocation a) {  
    cerr << "Houlala, problème d'allocation" << endl;  
    cerr << "Taille demandée=" << a.getSize() << endl;  
    //suite du traitement de l'erreur  
} catch(ErreurOuverture b) {  
    cerr << "Impossible d'ouvrir " << b.getName() << endl;  
    //suite du traitement de l'erreur  
}  
//suite du programme
```

- lors de l'exécution, si une erreur se produit lors de l'appel à `ouvreFichier("database.txt")` :

```
try {  
    cout << "bonjour tout le monde" << endl;  
    int *t = alloueTableau(100000);  
    cout << "jusqu'ici tout va bien" << endl;  
    int d = ouvreFichier("database.txt");  
    cout << "tout va bien, c'est cool" << endl;  
} catch(ErreurAllocation a) {  
    cerr << "Houlala, problème d'allocation" << endl;  
    cerr << "Taille demandée=" << a.getSize() << endl;  
    //suite du traitement de l'erreur  
} catch(ErreurOuverture b) {  
    cerr << "Impossible d'ouvrir " << b.getName() << endl;  
    //suite du traitement de l'erreur  
}  
//suite du programme
```

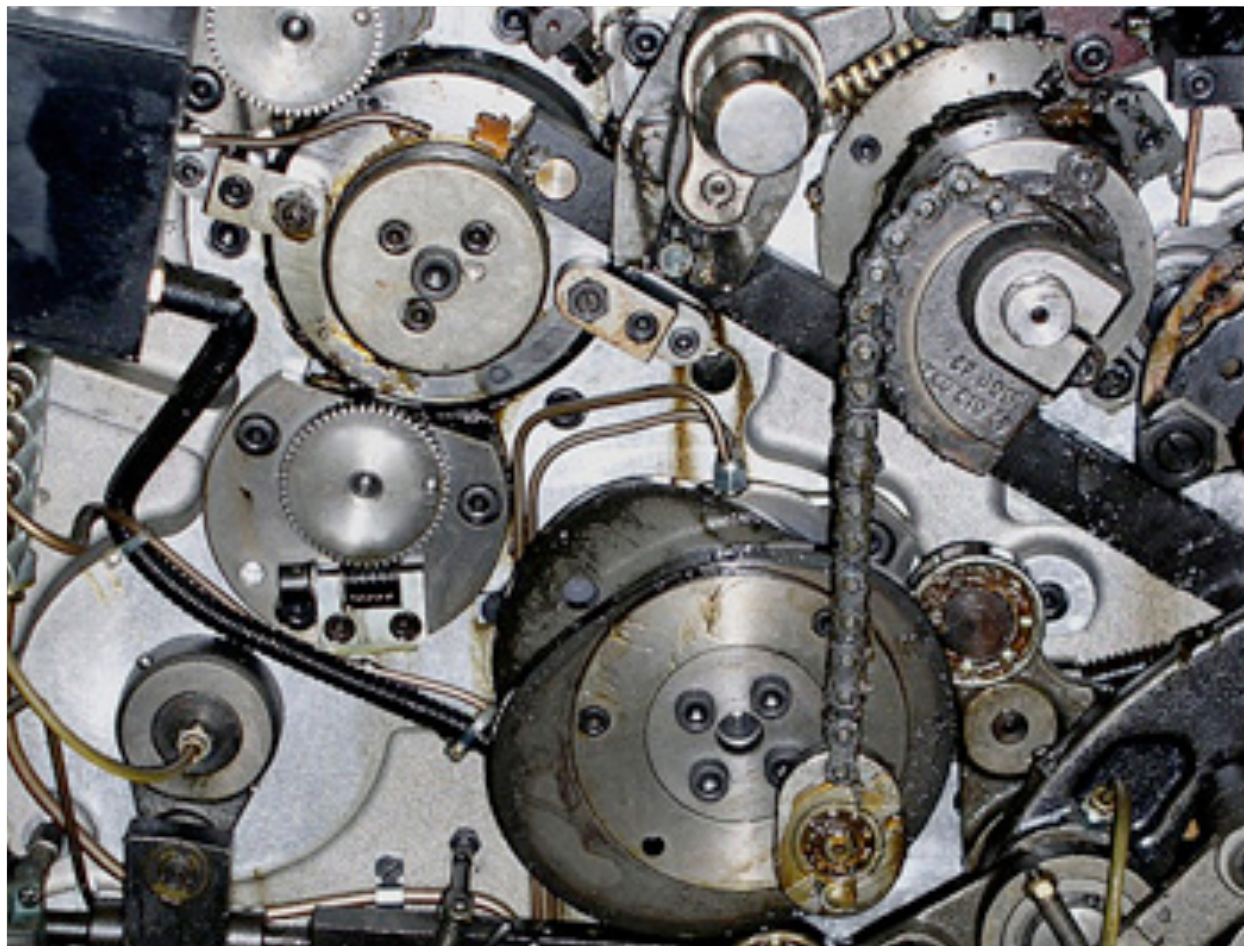
- En cas d'exception rencontrée dans un bloc `try` le *runtime* recherche le bloc de capture correspondant à l'erreur :
 - en examinant les types indiqués dans les entêtes des blocs `catch` et dans l'ordre de leur apparition dans le code source (attention au polymorphisme!)
 - en cas de succès l'exécution reprend son cours normal à la première instruction du bloc
 - en cas d'échec le *runtime* remonte dans la pile des appels...

- de façon équivalente aux `switchs` il existe une capture par défaut

`catch (. . .)`

- celui qui détecte une condition qui empêche de continuer dans des conditions normales et donc nécessitant un traitement particulier doit simplement lever une exception (contenant les informations nécessaires à une prise de décision raisonnable)... Il ne doit pas se préoccuper de la façon dont elle sera effectivement corrigée!
- dans la vie courante... Le concepteur du mécanisme de contrôle de l'injection de carburant dans un avion ne doit pas prendre une quelconque décision concernant le vol lorsqu'il détecte un problème!!!! Il doit faire en sorte qu'un voyant s'allume sur le tableau de bord... La décision effective sera prise par le pilote...

LES CAS PLUS COMPLIQUÉS



- Le mécanisme permet de réaliser des traitements d'erreur compliqués
 - en particulier, il n'est pas toujours possible de corriger l'erreur en un seul endroit
 - on peut donc vouloir corriger partiellement en plusieurs endroits
- il n'est pas toujours possible de fournir toutes les informations nécessaires à la prise de décision à l'endroit même où l'erreur se détecte...

- le redéclenchement (par `throw`) :

```
void ouvreFichiers(char *nom) {  
    FILE *f = fopen(nom, "r");  
    char n[100];  
    int lineno=0;  
    while ( fgets(n,100,f)!=0 ) {  
        try {  
            lineno++;  
            ouvreFichier(n);  
        } catch (ErreurFichier err) {  
            err.setLine(lineno);  
            throw;  
        }  
    }  
    fclose(f);  
}
```

```
void ouvreFichier(char *nom) {  
    ...  
    if (problème)  
        throw ErreurFichier(nom);  
    ...  
}
```

Ici on ne connaît que le nom

par contre ici on connaît le numéro de ligne

allez on reprend
c'est pas fini

```
void uneFonction() {  
    try {  
        ouvreFichiers("app.conf");  
    } catch (ErreurFichier err) {  
        cout << "Impossible d'ouvrir le fichier "  
            << err.getName() << " en ligne " << err.getLineNo()  
            << " du fichier app.conf" << endl;  
    }  
}
```

- le code précédent contient une inconsistance majeure...

```
void ouvreFichiers(char *nom) {  
    FILE *f = fopen(nom, "r");  
    char n[100];  
    int lineno=0;  
    while ( fgets(n,100,f)!=0 ) {  
        try {  
            lineno++;  
            ouvreFichier(n);  
        } catch (ErreurFichier err) {  
            err.setLine(lineno);  
            throw;  
        }  
    }  
    fclose(f);  
}
```

Que devient le
handler de fichier
f en cas
d'exception ?

- bien sûr on peut prendre la précaution de fermer le fichier au bon moment...

```
void ouvreFichiers(char *nom) {  
    FILE *f = fopen(nom, "r");  
    char n[100];  
    int lineno=0;  
    while ( fgets(n,100,f)!=0 ) {  
        try {  
            lineno++;  
            ouvreFichier(n);  
        } catch (ErreurFichier err) {  
            err.setLine(lineno);  
            fclose(f);  
            throw;  
        }  
    }  
    fclose(f);  
}
```

Ok, il est
correctement
fermé

- Cela nécessite d'être très attentif.

- une solution plus sûre est connue sous le nom d'**acquisition de ressources par initialisation** (RAII Resource Allocation Is Initialization)
- cette technique utilise deux caractéristiques essentielles du langage :
 - toute variable locale est détruite à la sortie du bloc qui la déclare
 - toute destruction de variable fait appel au

- on va donc créer une classe intermédiaire!

```
void ouvrirFichiers(char *nom) {  
    FichierOuvert f(nom);  
    char n[100];  
    int lineno=0;  
    while ( fgets(n,100,f)!=0 ) {  
        try {  
            lineno++;  
            ouvrirFichier(n);  
        } catch (ErreurFichier err) {  
            err.setLine(lineno);  
            throw;  
        }  
    }  
}
```

```
class FichierOuvert {  
    private:  
        FILE *file;  
    public:  
        FichierOuvert(char *nom) {  
            file = fopen(nom,"r");  
        }  
        ~FichierOuvert() {  
            if (file!=-1) fclose(file);  
            file=0;  
        }  
        operator FILE*() {  
            return file;  
        }  
};
```

Ok, il est correctement fermé quoi qu'il arrive

- c'est une difficulté majeure que de créer une fonction qui se comporte correctement vis-à-vis de la cohérence globale du programme lorsqu'une exception est déclenchée :
- ce problème est répertorié sous le vocable suivant :
 - exception safe function, **exception safety**
- d'autres langages règlent en partie la question, lors de la capture, à l'aide de la clause `finally` mais pas le C++

- *quid* des initialisations de membres ?

```
class A {  
    public:  
        A(int v) { ... }  
};
```

```
class B {  
    public:  
        B(int v) { ... }  
};
```

```
class C {  
    private:  
        A a;  
        B b;  
    public:  
        C(int x, int y) : a(x), b(y) { ... }  
};
```

- *quid* des initialisations de membres en cas d'exception ?

```
class A {  
    public:  
        A(int v) {  
            throw 666;  
        }  
};
```

```
class C {  
    private:  
        A a;  
        B b;  
    public:  
        C(int x, int y) : a(x), b(y) { ... }  
};
```

```
class B {  
    public:  
        B(int v) { ... }  
};
```

Ok rien de spécial ne se produit
puisque l'on arrive même pas à
construire le premier membre, on
ne construit ni le second membre,
ni l'objet tout entier...

- *quid* des initialisations de membres en cas d'exception ?

Ok, ici on a construit le premier membre, mais lors de la construction du second on rencontre une exception; le premier membre est alors détruit et l'objet n'est pas construit

```
class C {  
    private:  
        A a;  
        B b;  
    public:  
        C(int x, int y) : a(x), b(y) { ... }  
};
```

```
class A {  
    public:  
        A(int v) { ... }  
};
```

```
class B {  
    public:  
        B(int v) {  
            throw 666;  
        }  
};
```


- comment, en plus de cela, faire en sorte qu'un code soit exécuté alors que l'initialisation d'un membre a échoué ?

- utiliser un **function-try-block**

- quoi ?

- un **function-try-block**

- hein ?



*Attention :
C++ avancé*

**PARENTAL
ADVISORY**
EXPLICIT CONTENT

- *quid* des initialisations de membres en cas d'exception ?

```
class C {  
    private:  
        A a;  
        B b;  
    public:  
        C(int x,int y) try : a(x), b(y) {  
            ... // ok : something to do  
        } catch(Type err) {  
            ... // wrong : something else (e. cochrane)  
        }  
};
```

doit terminer par `throw qqe_chose;`
sinon le compilateur rajoute `throw;`

- le function-try-bloc peut-être utilisé dans le cas des fonctions plus ordinaires que les

```
void uneFonction(int &vendredi) try {  
    throw 13;  
} catch(int v) {  
    vendredi = v;  
}
```

```
int main() {  
    int i;  
    uneFonction(i);  
    cout << i << endl;  
    return 0;  
}
```

```
yunes% ./test  
13  
yunes%
```

- évidemment les deux définitions sont

```
void uneFonction(int &vendredi) try {  
    throw 13;  
} catch(int v) {  
    vendredi = v;  
}
```

```
void uneFonction(int &vendredi) {  
    try {  
        throw 13;  
    } catch(int v) {  
        vendredi = v;  
    }  
}
```


- les `auto_ptr` c'est quoi ?
- ils permettent de résoudre les problèmes liés aux exceptions dans les fonctions qui acquièrent des ressources
- problème déjà vu
- solution : Acquisition de Ressource par Initialisation (RAII Resource Acquisition Is Initialization)
- OU `auto_ptr`

**Danger :
C++ avancé**



- il s'agit de pointeurs intelligents (*smart pointers*)
- lorsqu'un `auto_ptr` est détruit, l'objet pointé aussi le sera
- un `auto_ptr` est propriétaire de l'objet pointé qui n'a jamais qu'un seul propriétaire
- la propriété est transférée en cas de copie

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

int main() {
    auto_ptr<int> sp1(new int);
    *sp1 = 134;
    auto_ptr<int> sp2;
    sp2 = sp1;
    cout << *sp2 << endl;
    *sp1; // Arghh! sp1 n'est plus proprio...
    return 0;
}
```

- Attention : l'usage des `auto_ptr` est réservé aux experts... Il faut faire très attention.

```
try {  
    auto_ptr<int> p(new int);  
    // something  
    throw 666;  
} catch (...) {  
    // correction  
}
```

- Dans ce cas, la mémoire allouée est :
 - désallouée lors de la sortie ordinaire (suppression des variables locales en sortie de bloc)
 - désallouée lors de la sortie par exception (*stack unwinding*)

- les exceptions et l'opérateur `new`
 - dans le cas ordinaire de l'utilisation de `new`, lorsqu'une exception est déclenchée, la mémoire allouée est libérée
 - pas de fuite mémoire
- le cas particulier de l'opérateur d'allocation placée (*placement new*)

**Méfiance : C++
super méchant**

`new(adresse) type;`

- qui permet d'utiliser l'adresse mémoire donnée pour placer un objet du *type* indiqué puis de l'initialiser (donc pas d'allocation)

- ce qu'il faut savoir :
- puisqu'il s'agit d'une allocation non standard, alors la désallocation est nécessairement non standard
- si l'allocateur utilisé est celui qui a été surchargé pour la classe (*i.e.* `type::operator new()`)
 - alors `type::operator delete()` sera appelé s'il existe

- la reprise...
- le *runtime* C++ offre un mécanisme de reprise sur erreur pour les allocations
- lorsqu'une allocation n'est pas possible et qu'un handler existe, celui-ci est appelé
 - c'est donc à lui que revient la responsabilité de trouver de la mémoire (bonne implantation d'un appel à un *garbage collector* par exemple)
 - s'il n'en est pas capable il doit :
 - arrêter l'exécution (`abort`, `exit`, etc)

- la fonction

```
typedef void (*new_handler_t)();  
new_handler_t set_new_handler(new_handler_t);
```

- permet de choisir le *handler* qui devra être appelé en cas d'échec d'allocation

- les exceptions dans les destructeurs...
 - attention il est possible de lever des exceptions dans les destructeurs mais
 - dans le cas normal d'une destruction, cela ne pose aucun problème théorique. En pratique l'appelant ne sait pas quoi faire...
 - dans le cas d'une destruction alors qu'une exception est en cours et qu'on dépile les appels (*stack unwinding*)
 - c'est INTERDIT et cela provoque un appel à `std::terminate()`
 - il est plus que formellement déconseillé de lever des exceptions dans les destructeurs...

- il existe une version de new qui ne lève pas d'exception :

```
void * new (std::nothrow) T;
```

RETOUR SUR LA SURCHARGE
DE L'OPÉRATEUR NEW

- nous avons déjà rencontré trois versions de l'opérateur new :

- `void * new T`

- placement new : `void * new(void *) T`

- sans exception : `void * new(std::nothrow) T`

- Rappel :
 - pour une classe donnée la surcharge de l'opérateur **new** est réalisée par implémentation d'une méthode statique de classe :

```
class UneClasse {  
public:  
    static void *operator new(size_t);  
    static void *operator new[](size_t);  
};
```

- l'argument correspond au nombre d'éléments

- des arguments supplémentaires peuvent être utilisés :

```
class UneClasse {  
public:  
    // par exemple  
    static void *operator new(size_t,int);  
};
```

- à l'appel :

```
UneClasse *uc = new (45) UneClasse;
```

LA SPÉCIFICATION D'EXCEPTION

- Le déclenchement d'exception modifie les rapports entre les fonctions...
- Il est alors nécessaire de documenter dans le code l'existence d'instructions de levée d'exception
 - Cela permet aux appelants d'être prévenus...

- La syntaxe de documentation est

type nom(*type id*[,...]) throw (*type*₁[,...]);

- qui indique à l'appelant que les exceptions qui peuvent être levées sont uniquement des types indiqués
- dans le cas contraire (levée d'une type n'apparaissant pas dans la spécification), le *runtime* appelle `std::unexpected()` qui par défaut appelle simplement `std::terminate()`.

- la spécification d'une liste vide permet d'indiquer que la fonction ne lève pas d'exception
 - s'il n'y a pas de liste, n'importe quelle exception peut être levée...
- Attention avec les redéfinitions :
 - la liste des exceptions d'une fonction membre d'une sous-classe ne peut être moins restrictive que la liste de la fonction membre qu'elle redéfinit
 - idem pour les affectations de pointeurs sur fonctions

- la spécification d'une liste contenant `std::bad_exception` modifie le comportement de `std::unexpected()` qui dans ce cas n'appelle plus `std::terminate()` mais déclenche simplement `std::bad_exception`.

LA REPRISE SUR ERREUR

- de la même manière qu'il existe une reprise sur erreur dans `new` (`new_handler`), il existe une reprise sur erreur dans `std::unexpected()` :

```
typedef
```

```
    void (*unexpected_handler_t)();
```

```
unexpected_handler_t
```

```
    set_unexpected(unexpected_handler_t);
```



```
typedef  
    void (*terminate_handler_t)();
```

```
terminate_handler_t  
    set_terminate(terminate_handler_t);
```