



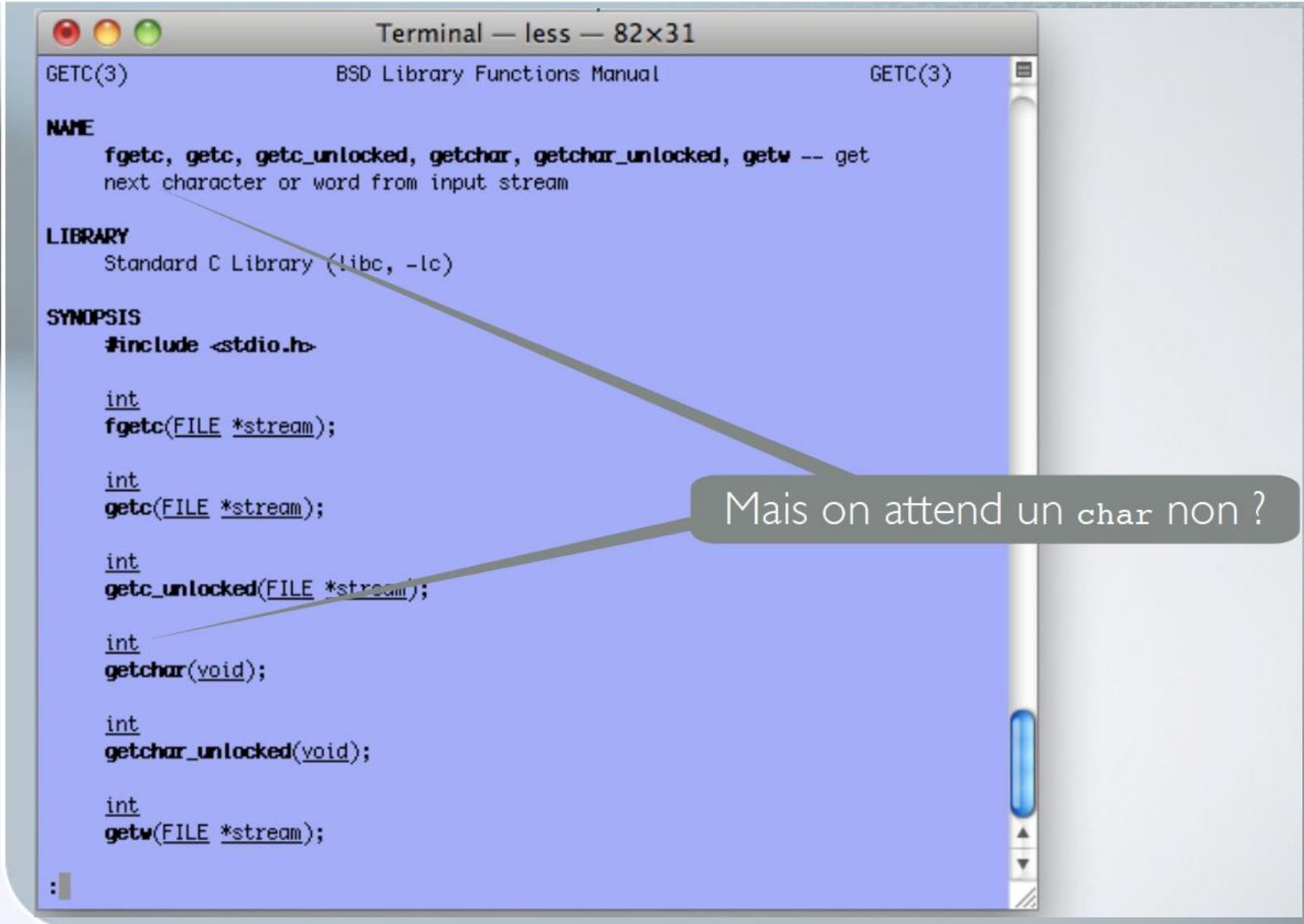
ISIE4 – LOO C++

La gestion des exceptions

Avant de commencer...

- Essayons de nous rappeler comment étaient (très/trop souvent) traitées les erreurs « avant »...
 - Notamment dans la libc

Un premier exemple...



A terminal window titled "Terminal — less — 82x31" displays the BSD Library Functions Manual for GETC(3). The window has a light blue background. The text is as follows:

```
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);
```

A grey callout box with the text "Mais on attend un char non ?" is positioned over the `getc(FILE *stream);` line. Two lines from the box point to the `getc` function name and the `FILE *stream` argument.

Un second...

Terminal — less — 82×31

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). If `O_CREAT` is specified, `open` requires a third argument `mode_t mode`, the permissions to be used to create the file, as described in `chmod(2)` and `chmod(1)`. The `mode` argument is masked with the value returned by `umask(2)`.

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

<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

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:

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

Mais on attend un descripteur!

Le problème ici

- Tout est mélangé
 - La partie fonctionnelle du code
 - La partie traitement des erreurs
- Eviter à tout prix !
 - Même lorsque l'on ne bénéficie pas d'un mécanisme dédié (exceptions)
- La solution (en C par exemple)
 - Les fonctions ne retournent QUE des codes d'états
 - Les E/S exploitent le passage de paramètres par adresse et les structures
 - Notion de descripteur

Autre problème

■ La sécurité...

```
// 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 philosophie de la solution

- Séparer, au niveau code (et donc de l'analyse)
 - La partie fonctionnellement intéressante de l'application
 - La partie dédiée au traitement des modes dégradés
- Certains langages intègrent nativement des mécanismes permettant de réaliser aisément cette séparation
 - Les exceptions

Ça devient comme ça...

```
// 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

Le mécanisme de contrôle du flux d'exécution

- Une fonction/méthode réussit lors de son invocation à réaliser sa tâche
 - Elle se **termine** et **renvoie** une valeur
 - Sortie « normale » (nominale)
- Une fonction/méthode rencontre quelque chose l'empêchant de réaliser sa tâche
 - Une sortie **immédiate** (et précipitée) est opérée
 - Sortie en **erreur**
 - C'est un événement exceptionnel !

En résumé

- 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*
 - Une exception porte de l'information, ce n'est pas que le signalement d'une erreur

Déclencher une exception

- ***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(...);**

Ce qu'il se passe...

- 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.

Le plus important

- Pour le concepteur d'une méthode/fonction qui décide de lever une exception
- Il ne doit pas se préoccuper de savoir
 - si l'erreur sera récupérée
 - de comment l'erreur sera récupérée
 - ou l'erreur est récupérée, si elle l'est
- Mais il doit fournir les informations appropriées sur l'erreur, permettant éventuellement de la corriger

Rappel technique

- L'expression en lien avec la levée d'une exception peut être de n'importe quelle nature
- Par exemple
 - Un entier
 - `throw 666;`
 - Une instance de classe (un objet)
 - `throw A();` (A est une classe)
 - Un pointeur
 - `throw new A()` (A est une classe)
 - Non conseillé cependant...

Les exceptions standard

- La bibliothèque standard définit des exceptions par « domaines »
 - `Logic_error`
 - `Invalid_argument`
- Il est possible de s'appuyer dessus
 - En les utilisant de manière directe
 - `throw std::logic_error{};`
 - Ou en spécialisant une exception spécifique pour la rendre encore plus adaptée au context
- Simple et efficace !

Capturer une exception

- Face à une potentielle levée d'exception par un service invoqué, deux choix sont possibles
 - Ignorer l'exception
 - On ne fait rien de particulier
 - Uniquement lorsque l'on n'est pas en situation de faire quoi que ce soit !
 - Tenter de corriger certaines erreurs
 - Ce mécanisme s'appelle la capture

Le mécanisme

- la partie du code dans laquelle on désire tenter de récupérer les erreurs doit être incluse dans un bloc qualifié par ***try***
- si une erreur se produit elle peut être récupérée dans l'un des blocs (qui suivent immédiatement) qualifié par ***catch(Type e)***

Forme générale

```
try {  
    bloc d'instructions dans lequel  
    des exceptions peuvent se produire  
} catch(TypeErreur1 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur de TypeErreur1  
} catch(TypeErreur2 erreur) {  
    bloc d'instructions de traitement  
    de l'erreur de TypeErreur2  
} ...  
...  
    catch(TypeErreurn erreur) {  
        bloc d'instructions de traitement  
        de l'erreur de TypeErreurn  
    }  
}
```

Exemple

```
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
```

Trois situations possibles

```
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
```

```
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
```

```
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
```

Précisons le mécanisme

- 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...
- Remarque : capture par défaut
 - catch(...)

Des cas plus compliqués...

- Polymorphisme dans les classes d'exception
 - Il suffit d'y prendre garde
- Ajouter sa pierre à la chaîne de responsabilité
 - Redéclenchement
- Ne pas laisser de bazar même en cas de départ précipité
 - RAI
- Faire quelque chose même quand a priori c'est impossible...
 - Function-Try-Block

Polymorphisme dans les classes d'exception

- Les séquences « catch » sont évaluées dans l'ordre
 - Attention au sous-typage par polymorphisme
 - `class E2:public E{ };`
- Séquences possibles
 - `catch (E e){};catch (E2 e){}`
 - Problème
 - `catch (E2 e){};catch (E e){}`
 - OK

Redéclenchement (throw)

- 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...

Redéclenchement (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() << endl;  
    }  
}
```

RAII

- L'exemple 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 ?

RAII

- Solution a priori
 - prendre la précaution de fermer le fichier au bon moment...
 - Mais, cela nécessite d'être très attentif
- Solution plus sûre
 - Acquisition de Ressources par Allocation
 - RAII : Ressource Allocation Is Initialization
- Exploitation de deux caractéristiques du langage:
 - toute variable locale est détruite à la sortie du bloc qui la déclare
 - toute destruction de variable fait appel au destructeur

RAII

■ Création d'une classe intermédiaire

```
void ouvreFichiers(char *nom) {
    FichierOuvert f(nom);
    char n[100];
    int lineno=0;
    while ( fgets(n,100,f)!=0 ) {
        try {
            lineno++;
            ouvreFichier(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

Exception Safety

- 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++

function-try-block - Contexte

■ Quid de l'initialisation des 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) { ... }  
};
```

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...

Mais...

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;  
        }  
};
```

function-try-block

- 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

```
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. cochran)  
        }  
};
```

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

Pour les fonctions « ordinaires »

- Le function-try-block peut aussi être exploité avec des fonctions « ordinaires »
 - Pas réservé aux constructeurs

```
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%
```

Conclusion

- Le mécanisme de gestion des erreurs basé sur les exceptions est très intéressant, utile, efficace...
- Il faut cependant bien maîtriser celui-ci pour pouvoir en tirer parti de la meilleure manière qui soit
 - Sinon, cela peut vite devenir contre-productif et générer plus de problèmes que de solutions
- A utiliser avec minutie dans l'embarqué