Programmation avancée

AB – v3.3 (20/09/2022)

Visual Studio

Interface de développement de Visual Studio

- La solution (.sln)
  - Les projets (.vcxproj)
    - Les fichiers (.cpp, h., etc.)

- Voir les propriétés associés à chaque objet

Quelques commandes

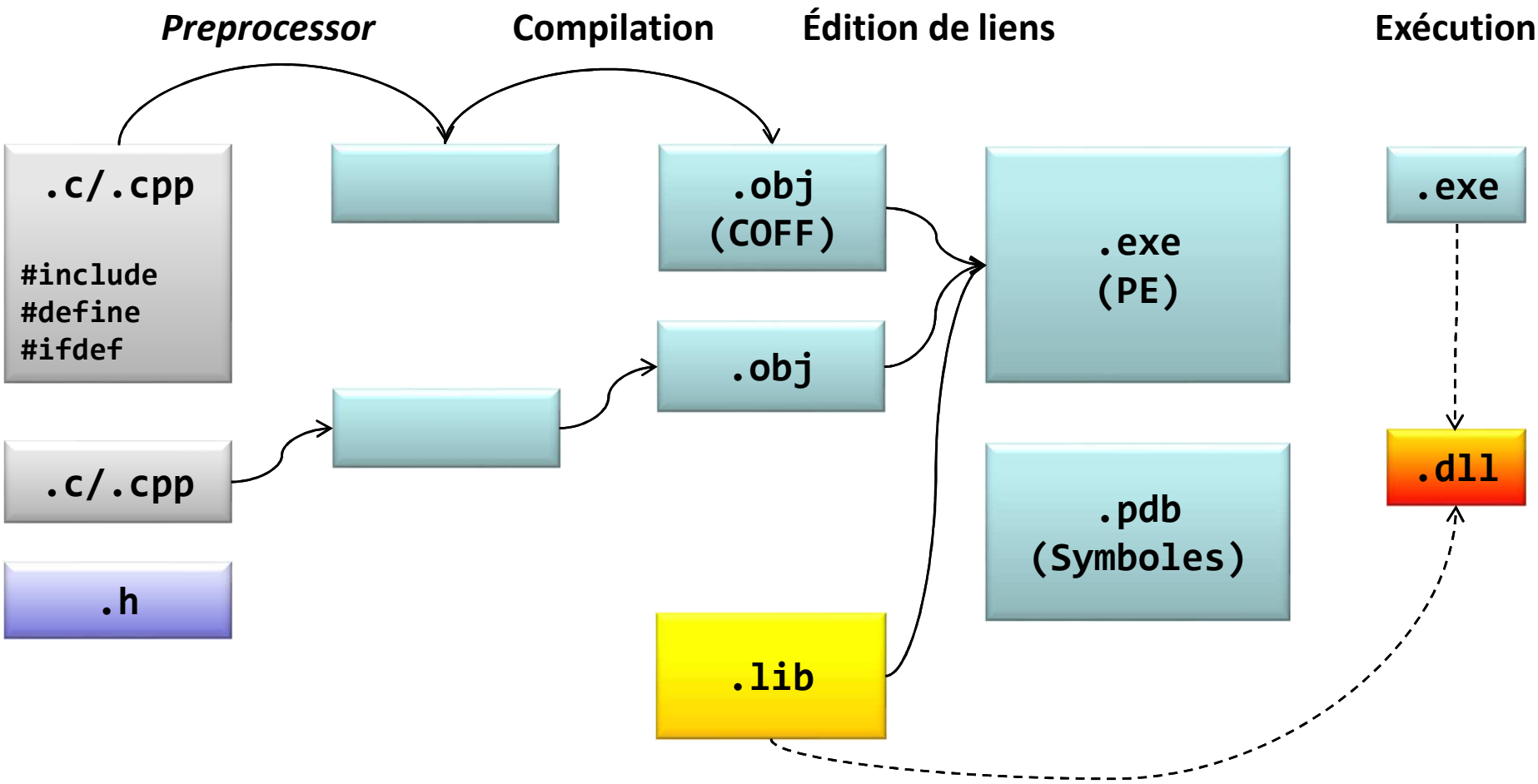
- La compilation (F7)
- L'exécution (F5)
- Le pas-à-pas principal (F10) et détaillé (F11)
- Les points d'arrêts (F9)
  - Points d'arrêts conditionnels

- « Atteindre la définition » (F12)

Quelques vues

- Code machine (Alt+8)
- Registres (Ctrl+Atl+G)
- Mémoire (Alt+6)
- Pile des appels (Alt+7)
  - Code externe
  - Valeurs des paramètres
- Les espions

Le processus de compilation



Configurations DEBUG/RELEASE

- **DEBUG :**
  - Preprocessor : _DEBUG
  - LinkIncremental
  - /Od (Optimization: Disabled)
  - /RTC1 (stack frame run-time error checking, reports variable used without having been initialized)
  - /ZI (Edit and Continue feature)
  - /Gm (Minimal Rebuild)
- **RELEASE :**
  - Preprocessor : NDEBUG
  - /GL (WholeProgramOptimization)
  - /O2 (Optimization: MaxSpeed)
  - /Gy (FunctionLevelLinking: true)
  - /Oi (IntrinsicFunctions: true)
  - OptimizeReferences: true

L’API Win32

L'API Windows

`#include <windows.h>`

- Définition des types de données (*Windows Data Types*) spécifiques à l'API Win32
  - `typedef unsigned long DWORD`
  - `typedef int BOOL`
  - `typedef BOOL *PBOOL`
  - `typedef BOOL far *LPBOOL`
- Inclusion des principaux fichiers d'en-tête de l'API pour le prototypage de base

Définition d'une fonction
Exemple : MessageBox

Requirements

Minimum supported client	Windows 2000 Professional
Minimum supported server	Windows 2000 Server
Header	Winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
Unicode and ANSI names	**MessageBoxW** (Unicode) and **MessageBoxA** (ANSI)

Variantes de fonctions

- Sleep
  - SleepEx
- LogonUser
  - LogonUserEx
    - LogonUserExExW
- LsaApLogonUser
  - LsaApLogonUserEx
    - LsaApLogonUserEx2
- CopyFile
  - CopyFile2

La notation hongroise

docs.microsoft.com/en-us/windows/win32/stg/coding-style-conventions

- Fonctions et variables de la forme :

```
CeciEstUneFonction ()
```

- Préfixe aux noms des variables pour indiquer leur type :
  - DWORD `dwTest`
  - BOOL `bResult`
  - HANDLE `hFile`
  - LPTSTR `szNom`
  - PBYTE `pData` ou `pbData`

C Runtime
(CRT)

Versions de Visual Studio / C

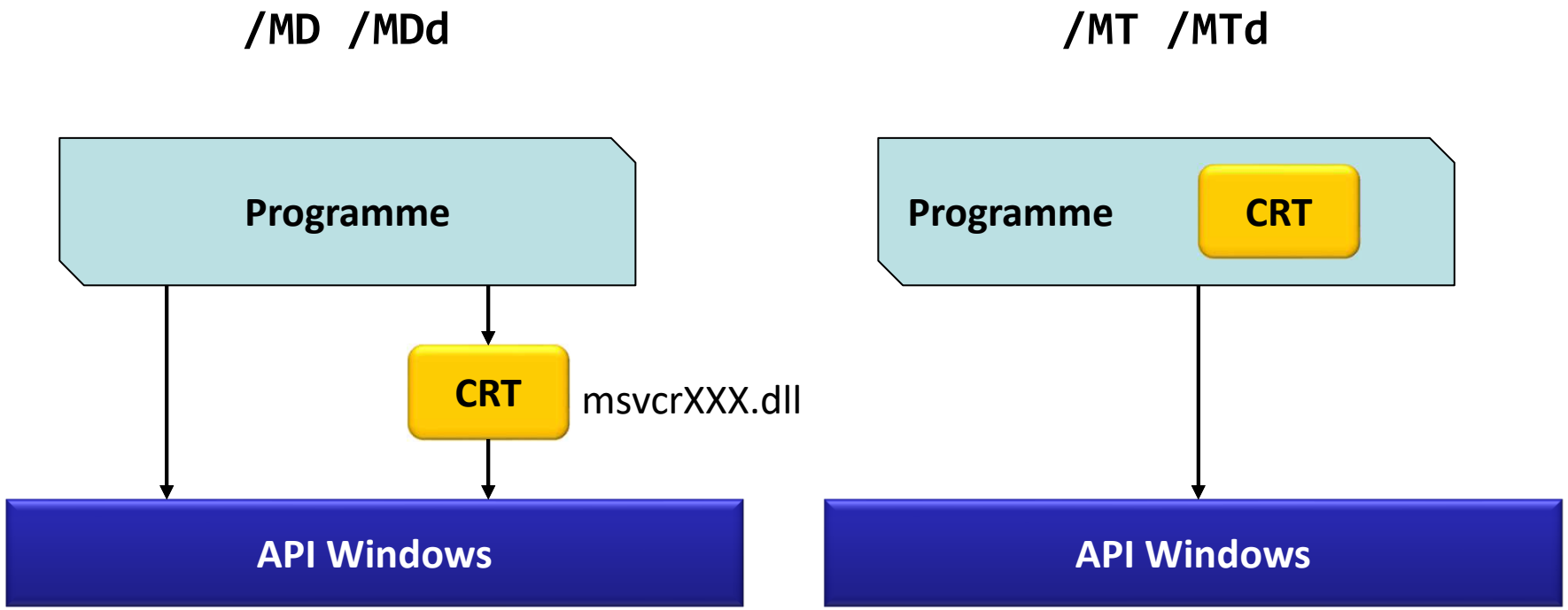
Visual Studio	Année de sortie	Build VS	Version C (_MSC_VER)
Visual Studio 97	1997	5.0	5.0
Visual Studio 6.0	1998	6.0	6.0
Visual Studio .NET	2002	7.0	7.0
Visual Studio .NET 2003	2003	7.1	7.1
Visual Studio 2005	2005	8.0	8.0
Visual Studio 2008	2008	9.0	9.0
Visual Studio 2010	2010	10.0	10.0
Visual Studio 2012	2012	11.0 (RTM → update 5)	11.0
Visual Studio 2013	2013	12.0 (RTM → update 5)	12.0
Visual Studio 2015	2015	14.0 (RTM → update 3)	14.0
Visual Studio 2017	2017	15.0 → 15.9	14.1 → 14.16
Visual Studio 2019	2019	16.0 → 16.11	14.2 → 14.29
Visual Studio 2022	2022	17.0 → 17.2	14.3

C Runtime (CRT)

- Tout programme C/C++ nécessite un CRT qui est une bibliothèque de support implémentant des fonctions basiques nécessaires pour l'exécution d'un programme C/C++ (gestion des *threads*, des exceptions, manipulation des chaînes de caractères)
- Celui-ci peut être :
  - Dynamique (recommandé) ou statique
  - En mode RELEASE (recommandé) ou mode DEBUG
- Les CRT ne sont pas inclus de base dans le système Windows ce qui nécessite l'installation des *Visual C++ Redistributable Packages*

https://support.microsoft.com/en-us/topic/the-latest-supported-visual-c-downloads-2647da03-1eea-4433-9aff-95f26a218cc0

C Runtime (CRT)
Dynamique / Statique



Option de sélection du CRT

Option	Directives de préprocesseur	Type	Bibliothèque	DLL
/MT	_MT	Multithread Statique / Release	libcmt.lib	N/A (statique)
/MD	_MT, _DLL	Multithread Dynamique / Release	msvcrt.lib	msvcrXXX.dll
/MTd	_DEBUG, _MT	Multithread Statique / Debug	libcmtd.lib	N/A (statique)
/MDd	_DEBUG, _MT, _DLL	Multithread Dynamique / Debug	msvcrtd.lib	msvcrXXXd.dll

- C : msvcrXXX.dll
- C++ : msvcpXXX.dll

Universal CRT

- Avec VS 2015, le CRT a été re-architecturé (*refactoring*) pour supporter les applications universelles en fusionnant 3 CRT (*desktop apps*, *Windows Store apps* et *Windows Phone apps*)
- Le CRT est désormais composé de 2 parties :
  - **VCRuntime** : fonctions liées au compilateur (création de processus, exception, etc.)
    - Composant lié au compilateur
  - **Universal CRT** (AppCRT et DesktopCRT) : autres fonctions
    - Composant lié au système

Universal CRT

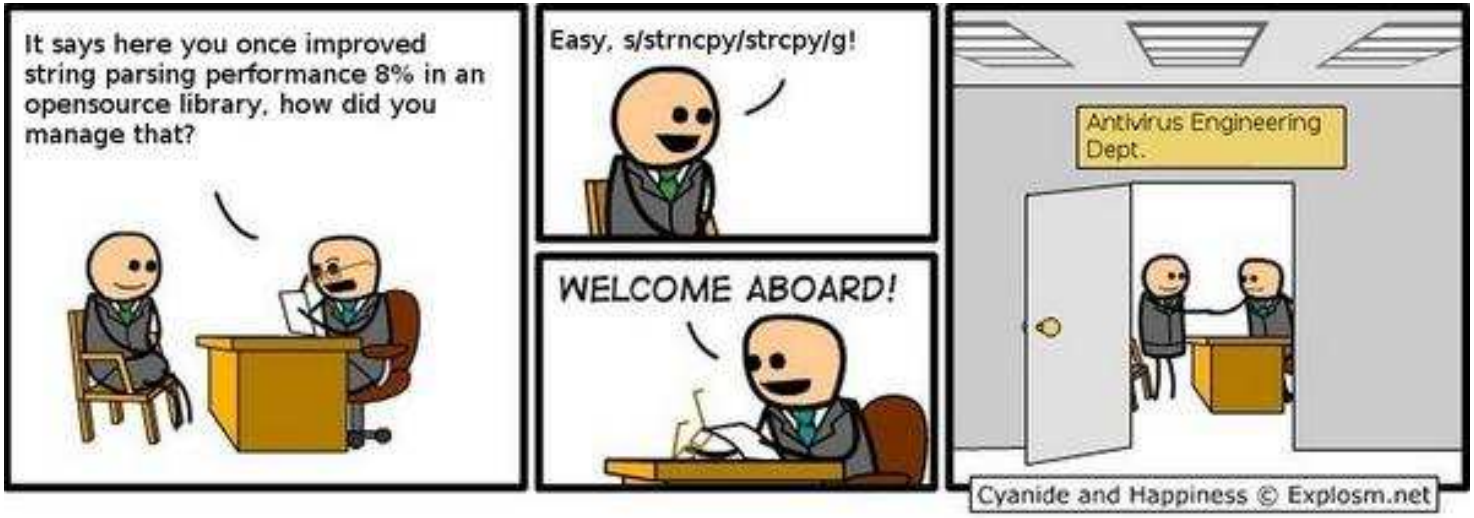
- L'UCRT est inclus depuis Windows 10
- Il doit être installé sur les systèmes antérieurs (Windows Vista/Server 2008 à Windows 8.1/Server 2012 R2)
  - KB2999226

- LIB : msvcrt.lib, vcruntime.lib et ucrt.lib
- DLL : vcruntime140.dll et ucrtbase.dll

Amélioration de la sécurité du CRT
(*Security Features in the CRT*)

- Fonctions « sécurisés » : `printf` -> `printf_s`
- Ne pas utiliser _CRT_SECURE_NO_WARNINGS
- Substitution automatique (*secure template overloads*) :

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
```



Symboles

Symboles

- Les symboles sont créés par l'éditeur de liens
- Ils sont utiles pour déboguer un programme, une bibliothèque ou un pilote (*driver*)
- Ils peuvent contenir :
  - le nom et type des variables : globales (*Global syms*), locales ou statiques
  - le nom des fonctions et des paramètres
  - *FPO records* si applicable
  - le nom et les numéros de lignes de code (*Line numbers*)
  - les définitions des structures internes (*Type info*)

Symboles

- Afin d'optimiser la taille du binaire, les symboles sont stockés dans un fichier séparé
- Les fichiers des symboles sont de type *Program Database* (extension `.pdb`, anciennement `.dbg`)
- Il est possible de générer une version « publique » d'un fichier de symboles (`/PDBSTRIPPED`), ce qui supprime les informations sur :
  - le code source (*SourceIndexed* dg *LineNumbers*)
  - les types de données (*TypeInfo*)
  - les données privées

Syntaxe des symboles

- La casse n’a pas d’importance

- Syntaxe :

Module!NomFonction

Serveur de symbole

- L'API des symboles est implémentée dans `dbghelp.dll`
- Le SDK de Windows fournit une version améliorée de la DLL apportant des fonctionnalités avancées (comme la gestion des serveurs de symboles)
- Les symboles sont recherchés dans un dépôt spécifié :
  - Explicitement (par programmation)
  - Via une variable d'environnement :
    - `_NT_SYMBOL_PATH`
    - `_NT_ALT_SYMBOL_PATH`

Format de la source du dépôt

- **Répertoire(s) simple(s) :**
  - `c:\symbols`
  - `\\partage`
  - `c:\symbols1;c:\symbols2`
- **Mise en cache** : **`cache*`***`localsymbolcache`*
  - `cache*c:\symbols;\\partage`
- **Serveur de symboles** : **`srv*`***`localsymbolcache`*`*`***`symbolstore`***
  - `srv*c:\symbols*http://msdl.microsoft.com/download/symbols`
    - `(srv = symsrv*symsrv.dll)`

Divers symboles

- dbh.exe
  - Info (cf. `IMAGEHLP_MODULE64`)
  - enum
  - etypes (private)
- pdbcopy.exe
- symchk.exe
- Ajouter des structures à un symbole existant
- https://github.com/Microsoft/microsoft-pdb

Static Analysis Tools

Principales technologies

- **Code Analysis tool (PREfast, /analyze) :**
  - Vérifications supplémentaires effectuées par le compilateur
  - Vérification du code fonction par fonction (rapide)
  - PREfast for Drivers (PFD) -> Code Analysis for Drivers
- **Static Driver Verifier (SDV) :**
  - Uniquement pour le code des pilotes
  - Vérification globale d'un pilote (lent) par exécution symbolique
- Initialement sous forme de produits séparés, ces outils sont intégrés à Visual Studio depuis VS2 012

Principales technologies

- **Dynamic Driver Verifier :**
  - Intégré à Windows (verifier.exe)
- **Global Flags**
  - Intégré à Windows mais outils de gestion dans le SDK (gflags.exe)

AddressSanitizer (ASan)

- Basé sur les travaux de Google
- Intégré depuis Visual Studio 2019 v16.9
  - `/fsanitize=address`
  - `/fsanitize=fuzzer`
  - `/fsanitize-address-use-after-return`
- *Kernel Address Sanitifer* (KASAN)
  - Prévu pour être ajouté à Windows 23H2

SAL
(*Source Code Annotation Language*)

- Annotation des fonctions afin d'améliorer la vérification statique du code (/analyze)
- Permet de préciser finement au moteur d'analyse le rôle et l'utilisation des paramètres des fonctions
- Intégré depuis Visual Studio 2005

- Deux formes de notation :
  - ***declspec syntax*** : forme initiale (`__in`)
  - ***attribute syntax*** : forme améliorée plus précise (`_In_`)

Principales notations

- Notation de base : _In_, _Inout_, _Out_, _Outptr_
- Paramètre optionnel : _X_**opt**_
- Spécification des tailles :
  - _In_reads_(s), _In_reads_bytes_(s)
  - _Inout_updates_(s), _Inout_updates_bytes_(s)
  - _Out_writes_(s), _Out_writes_bytes_(s)

SAL
Annotation Syntax (voir SAL.h)

Usage	Nullness	ZeroTerminated	Extent
_In_	<>	<>	<>
Out	opt_	z_	[byte]cap_[c_|x_](size)
Inout			[byte]count_[c_|x_](size)
_Deref_out_			ptrdiff_cap_(ptr)
			ptrdiff_count_(ptr)
Ret			
_Deref_ret_			
Pre			
Post			
_Deref_pre_			
_Deref_post_			

Erreurs

Types d’erreurs

- **Win32 Error Codes**
  - https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/18d8fbe8-a967-4f1c-ae50-99ca8e491d2d
  - 0x0000 de 0xFFFF
  - Utilisé par Win32 (WinError.h)
  - ERROR_SUCCESS
  - net helpmsg <code>
- **NTSTATUS**
  - https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebca55
  - Sev / C / N / Facility / Code
  - Utilisé par le noyau (ntstatus.h)
  - NT_SUCCESS(x) / NT_ERROR(x) / RtlNtStatusToDosError(x)
- **HRESULT**
  - https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/705fb797-2175-4a90-b5a3-3918024b10b8
  - S / R / C / N / X / Facility / Code
  - SUCCEEDED(x) / FAILED(x) / HRESULT_FROM_NT(x) / HRESULT_FROM_WIN32(x)

Divers

Débogage distant avec VS

- VS utilise le *Remote Debugger* (`msvsmon.exe`) pour déboguer des processus
- Les *Remote Tools* peuvent être installés sur un système où VS n'est pas installé afin d'effectuer du débogage distant

Vérification des fuites de mémoire

- Le CRT, en version DEBUG, inclut de base un détecteur de fuite de mémoire (`_CrtDumpMemoryLeaks`)
- Il peut être complété par un projet externe (*Visual Leak Detector for Visual C++*) afin de tracer d'autres fuites de mémoire

Inline Assembly

- Il n'est plus possible, en x64, d'utiliser des instructions assembleur dans le code (*inline assembly*)
- Certaines instructions (en particulier celles liées à la performance) peuvent être substituées par les fonction intrinsèques (*intrinsic functions*) :

```
#include <intrin.h>
__debugbreak();                    // int 3
```