Guide de l’utilisateur de Winelib

Guide de l’utilisateur de Winelib

Table des matières

Chapitre 1. Introduction à Winelib

1.1. Qu’est ce que Winelib?

Winelib est un kit de développement qui permet de compiler vos applications Windows sous Unix.

La plus grand partie du code de Winelib est une implémentation de l’API Win32. Heureusement, cette partie est totalement partagée avec Wine. Le reste est composé d’en-têtes compatibles avec Windows et des outils tels que le compilateur de ressources (et même ceux-ci sont utilisés pour compiler Wine).

Grâce à cela, Winelib est compatible avec la plupart des codes sources en 32 bits, fichiers de ressources et de message C et C++, et peut générer des applications en mode graphique ou en mode console ainsi que des bibliothèques dynamiques.

Winelib est incompatible avec les codes sources en 16bits car les types dont ils dépendent (en particulier les pointeurs segmentés) ne sont pas supportés par les compilateurs Unix. Quelques-unes des fonctionnalités les plus exotiques du compilateur de Microsoft telles que le support COM natif et la gestion des exceptions structurées sont aussi absentes. Vous serez donc peut-être amenés à effectuer quelques modifications de votre code pour recompiler votre application avec Winelib. Ce guide est là pour vous aider dans cette tâche.

Le bénéfice apporté par la recompilation de votre application avec Winelib est la possibilité de réaliser des appels aux API d’Unix directement depuis votre code source Windows. Cela permet une meilleure intégration à l’environnement Unix que celle obtenue en lançant une application Windows inchangée et lancée dans Wine. Un autre avantage est qu’une application Winelib peut-être recompilée relativement facilement sur une architecture non-Intel et exécutée sans avoir recours à une émulation du processeur peu performante.

1.2. Configuration requise

La configuration requise par Winelib est identique à celle de Wine.

En gros, si vous pouvez lancer Wine sur votre ordinateur, alors vous pouvez lancer Winelib. Mais la reciproque n’est pas vraie. Il est également possible de construire Winelib et des applications Winelib sur une plateforme incompatible avec Wine, ce qui est généralement le cas pour une plateforme équipée d’un processeur différent d’un i386. Mais on s’aventure alors sur un terrain encore largement inconnu. Il serait plus raisonnable d’opter pour une plateforme i386 classique en premier lieu.

La principale différence est que le compilateur devient beaucoup plus important. Il est vivement

recommandé d'utiliser gcc, g++, et les exécutables [binutils] GNU. Plus le compilateur gcc est récent, mieux c'est. Pour des morceaux de code conséquents, mieux vaut ne pas envisager quelque chose de plus ancien que gcc 2.95.2. Les versions récentes de gcc contiennent quelques corrections de bogues utiles et une compatibilité avec les structures et les unions anonymes bien meilleure. Il se peut que cela permette de réduire le nombre de changements à faire dans le code mais il ne s'agit pas de versions stables du compilateur, c'est pourquoi vous déciderez sans doute de ne pas l'utiliser en production.

1.3. Mise en route

1.3.1. Introduction à Winemaker

Que faut-il donc faire pour compiler une application Windows avec Winelib? En fait, cela dépend surtout de la complexité de votre application mais voici quelques problèmes qui surviennent pour toutes les applications :

- La casse de vos fichiers peut être mauvaise. Par exemple, il est possible que ceux-ci soient tous en majuscule : `BONJOUR.C`. Ce n'est pas ce qu'il y a de mieux ni sans doute ce que vous vouliez faire.
- Ensuite, il se peut que la casse des noms de fichiers dans vos directives d'inclusion soit fausse : peut-être qu'on y trouve 'Windows.h' au lieu de 'windows.h'.
- Vos directives d'inclusion utilisent peut être '\' au lieu de '/'. '\' n'est pas reconnu par les compilateurs Unix alors que '/' est reconnu par les deux environnements.
- Il sera nécessaire d'effectuer les conversions de fichiers textes habituelles de Dos vers Unix sans quoi vous aurez des ennuis lorsque le compilateur considèrera que votre '\' n'est pas à la fin de la ligne puisqu'il est suivi par un satané caractère de retour chariot.
- Il faudra écrire de nouveaux fichiers de compilation.

Le meilleur moyen de traiter de tous ces problèmes est d'utiliser winemaker.

Winemaker est un script perl conçu pour prendre en charge la conversion de vos projets Windows vers Winelib. Pour cela, il analysera votre code, en corrigeant les problèmes listés précédemment et générera automatiquement des fichiers de compilation.

Supposons que Wine/Winelib est installé dans le répertoire `/usr/local/wine`, et que vous êtes déjà dans le répertoire racine de vos sources. La conversion de votre projet vers Winelib se résume simplement au lancement des trois commandes ci-dessous (on notera le point indiquant le répertoire courant à la fin de la première commande) :

```
$ winemaker --lower-uppercase .
$ ./configure --with-wine=/usr/local/wine
$ make
```

Bien évidemment, les choses ne sont pas toujours aussi simples, ce qui explique justement l'existence de ce guide.

1.3.2. Tour de chauffe

Avant de commencer à travailler sur un gros projet, vous voulez peut-être essayer de réaliser le portage d'une petite application. L'application winemime provenant de l'arborescence des sources Wine convient bien à un petit test. Elle se trouve dans le sous-répertoire des programmes. Winemime est une application simple, mais possède quelques fichiers de resources et en-tête en C.

Exécutez un **make clean** dans le répertoire des sources de winemime s'il contient les résultats d'un assemblage précédent. Créez un répertoire séparé nommé winemime2, de façon à ce qu'il n'entre pas en conflit avec la copie de l'application Wine. Copiez les sources de winemime (fichiers *.c, *.h, *.rc) dans ce répertoire. Exécutez maintenant les commandes mentionnées précédemment depuis le répertoire winemime2 :

```
$ winemaker --lower-uppercase .
$ ./configure --with-wine=/usr/local/wine
$ make
```

Vous y êtes! Vous pouvez maintenant exécuter votre application au moyen de la commande **./winemime2**.

Si vous rencontrez des problèmes en préparant ou en assemblant cette application, cela signifie probablement que l'utilitaire winemaker est corrompu par des changements dans Wine. Vous pouvez vous en sortie en demandant de l'aide à l'adresse <`wine-devel@winehq.com`>.

1.3.3. Guide pas à pas

Reprenons les étapes précédentes plus en détails.

`Récupération des sources`

D'abord, essayez si possible d'obtenir les sources ainsi que les fichiers exécutables/bibliothèques qu'elles génèrent. Dans l'état actuel de winemaker, le fait de les avoir peut l'aider à deviner ce que votre projet essaie de construire. Plus tard, dès qu'il sera capable de comprendre les projets Visual C++, et que vous les utiliserez, cela ne sera plus nécessaire. Habituellement, les exécutables et les bibliothèques sont dans un sous-répertoire `Release` ou `Debug` du répertoire où se situent les sources. Le mieux est de pouvoir transférer les fichiers sources et l'un ou l'autre de ces répertoires vers Linux. Notez qu'il ne sera pas nécessaire de transférer les fichiers `.obj`, `.pch`, `.sbr` ou d'autres fichiers qui se trouvent également dans ces répertoires ; en particulier parce qu'ils ont tendance à être assez volumineux.

`cd <rep_racine>`

Allez ensuite dans le répertoire racine où se trouvent les fichiers sources. Winemaker est capable de traiter toute une hiérarchie de répertoires directement, c'est pourquoi il n'est pas nécessaire d'aller dans un répertoire en out de l'arborescence, bien au contraire. Winemaker génèrera automatiquement les fichiers de compilation dans chacun des répertoires où cela est nécessaire, et générera un fichier de compilation global afin de pouvoir reconstruire tous vos exécutables et bibliothèques avec une simple commande **make**.

`Autoriser l'accès en écriture aux sources`

Assurez-vous ensuite que vous possédez un accès en écriture à vos sources. Cela peut paraître évident, mais si vous avez copié vos sources depuis un CD-ROM ou si elles se trouvent dans Source Safe sous Windows, il y a des chances qu'elles soient en lecture seule. Or Winemaker a besoin d'un accès en écriture afin de les corriger. Ca peut se faire en lançant la commande **chmod -R u+w**. De plus, il va de soit que vous vous assurerez que vous avez des copies de sauvegarde de vos sources en cas de grosse catastrophe, ou plus vraisemblablement pour vous y référer plus tard. Si vous utilisez un système de contrôle de version, vous êtes déjà à l'abri.

Si vous avez déjà modifié vos fichiers sources et vous voulez vous assurer que winemaker n'y n'effectuera pas d'autres changements, vous pouvez alors utiliser l'option `--nosource-fix` pour les protéger.

`Lancement de winemaker`

Vous lancerez ensuite winemaker. Voici quelques options que vous utiliserez probablement. Pour obtenir la liste complète dans options, allez voir la page man de winemaker.

`--lower-uppercase`
`--lower-all`

Ces options spécifient comment traiter les fichiers, et les répertoires, qui ont une casse 'incorrecte'. `--lower-uppercase` spéfifie qu'ils seront renommé seulement si leur nom est tout en majuscule. Ainsi, les fichiers possédant une casse mixte, comme 'Hello.c' ne seront pas renommés. `--lower-all` renommera tous les fichiers. Si aucune des deux options n'est spécifiée, aucun fichier ou répertoire ne sera renommé, enfin presque. Comme vous le verrez plus loin, il est toujours possible que winemaker ait à renommer quelques fichiers.

`--nobackup`

Normalement, winemaker réalise une copie de sauvegarde de tous les fichiers dans lesquels il fait plus qu'une simple conversion de Dos vers Unix. Mais si vous possédez déjà des copies (bien pratiques) de ces fichiers autre part, il est possible que vous n'en n'ayez pas besoin, c'est pourquoi il vous faudra utiliser cette option.

`--dll`
`--console`

Ces options permettent d'indiquer à winemaker quelle type d'application cible vous assemblez. Si vous avez la bibliothèque windows dans votre arborescence source, vous ne

devriez pas avoir besoin de spécifier `-dll`. Mais si vous avez des exécutables en mode console, il vous faudra utiliser l'option correspondante.

`--mfc`

Cette option indique à winemaker que vous êtes en train d'assembler une application/bibliothèque MFC.

`-Dmacro[=defn]`
`-Idir`
`-Ldir`
`-idll`
`-llibrary`

Le `-i` indique un bibliothèque Winelib à importer via le mécanisme de fichier spec, à la différence de l'option `-l` qui spécifie une bibliothèque Unix à lier. Les autres options fonctionnent de la même manière qu'avec un compilateur C. Toutes s'appliquent à toutes les cibles trouvées. Lorsque l'on spécifie un répertoire avec soit l'option `-I` ou `-L`, winemaker va préfixer un chemin relatif avec `$(TOPDIRECTORY)/` afin qu'il soit valide depuis n'importe quel répertoire source. Vous pouvez également utiliser vous même une variable dans le chemin si vous le souhaitez (mais n'oubliez d'ajouter un \ devant $). Par exemple, vous pourriez spécifier `-I\$(WINELIB_INCLUDE_ROOT)/msvcrt`.

Ainsi, votre commande pourrait finalement être quelque chose comme : `winemaker --lower-uppercase -Imylib/include.`

`Renommage des fichiers`

Quand vous exécutez winemaker, celui-ci va d'abord modifier la casse des fichiers en fonction de vos attentes et aussi afin qu'ils puissent être traités par les fichiers de compilation. Ce deuxième point implique que les fichiers dont l'extension n'est pas en minuscule vont être renommés pour que l'extension soit en minuscule. Ainsi, par exemple, `HELLO.C` sera renommé en `HELLO.c`. De même, si un nom de fichier ou de répertoire contient un espace ou un dollar, ce caractère sera remplacé par une barre de soulignement. On procède de la sorte car ces caractères posent problème avec les versions actuelles d'autoconf (2.13) et make (3.79).

`Modification des sources et génération du fichier de compilation`

winemaker va ensuite procéder à la modification des fichiers source afin qu'il se compilent plus aisément avec Winelib. Pendant cette opération, il se peut qu'il affiche des messages d'avertissement lorsqu'il doit deviner ou identifier une syntaxe qu'il ne peut pas corriger. Finalement, il générera des fichier de compilation basés sur autoconf. Une fois que tout cela est fait, vous pouvez passer en revue les changements qu'a réalisé winemaker sur vos fichiers en utilisant **diff -uw**. Par exemple : **diff -uw hello.c.bak hello.c**.

`Running the configure script`

Avant d'exécuter la commande **make**, il faut lancer le script de configuration d'autoconf (**configure**). Le but de cette étape est d'analyser votre système et générer des fichiers de compilation personnalisés à partir du fichier `Makefile.in`. C'est également ici qu'il faudra

indiquer où se trouve Winelib dans votre système. Si Wine est installé dans un répertoire seul ou que les sources de Wine ont été compilées quelque part, vous pouvez exécuter respectivement **./configure --with-wine=/usr/local/bin** ou **./configure --with-wine=~/wine**.

`Lancement du make`

Cette étape est relativement simple : tapez juste **make** et bingo, vous devriez avoir tous vos exécutables et bibliothèques. Si cela n'a pas marché, cela signifie qu'il vous faudra lire ce guide plus à fond afin de :

- Revoir les fichiers `Makefile.in` afin de rectifier les options de compilation et d'assemblage par défaut réglées par winemaker. Dans la section *Analyse des sources de Winemaker*, vous trouverez quelques astuces.
- Régler les problèmes de portabilité dans vos sources. Vous trouverez plus d'informations dans la rubrique *Problèmes de portabilité*.

Si vous en arrivez à modifier vous même le fichier Makefile.in pour spécifier l'emplacement des fichiers de bibliothèque ou d'en-têtes Wine, revenez à l'étape précédente (le script de configuration) et utilisez les diverses options de type --with-wine-* pour indiquer où elles se situent.

Chapitre 2. Problèmes de portabilité

2.1. Unicode

Le type `wchar_t` possède différentes tailles standard sous Unix (4 octets) et sous Windows (2 octets). Il est nécessaire d'avoir une version récente de gcc (2.9.7 ou postérieure) qui supporte l'option *`-fshort-wchar`* afin d'adapter la taille du type `wchar_t` à celle attendue par les applications Windows.

Si vous utilisez Unicode et que vous voulez avoir la possibilité de réaliser des appels standard à la bibliothèque (ex. : `wcslen`, `wsprintf`), alors il faudra utiliser la bibliothèque de chargement msvcrt au lieu de glibc. Les fonctions de glibc ne fonctionneront pas correctement avec des chaînes de caractères de 16 bits.

2.2. Bibliothèque C

Deux choix s'offrent à vous en ce qui concerne la bibliothèque C à utiliser : la bibliothèque C native glibc ou la bibliothèque C msvcrt.

Notez que sous Wine, la bibliothèque crtdll est implémentée en utilisant msvcrt, il n'y a donc aucun avantage à essayer de l'utiliser.

En général, utiliser glibc est ce qu'il y a de moins pénalisant, mais cela n'est vraiment important que pour les entrées/sorties fichier, puisque beaucoup de fonctions de msvcrt sont résolues par glibc.

Pour utiliser glibc, il n'est pas nécessaire d'effectuer des changements dans votre application; cela devrait marcher directement. Il y a quelques situations pour lesquelles l'utilisation de glibc n'est pas possible :

1. Votre application utilise des fonctions unicode des bibliothèques Win32 et C.
2. Votre application utilise des appels propres à MS, tels que `beginthread()`, `loadlibrary()`, etc.
3. Vous avez recours à une sémantique précise des appels, par exemple, le fait de retourner `-1` plutôt qu'un entier non nul. Plus vraisemblablement, votre application repose sur des appels tels que `fopen()` prenant en paramètre un chemin Windows plutôt qu'un chemin Unix.

Dans ces cas de figure, vous devriez utiliser msvcrt pour effectuer vos appels au moteur d'exécution C. Pour ce faire, ajoutez la ligne :

```
import msvcrt.dll
```

dans les fichiers `.spec` de vos applications. Ainsi **winebuild** pourra résoudre vos appels à la bibliothèque c vers `msvcrt.dll`. Beaucoup d'appels simples qui se comportent de la même manière ont été qualifiés comme non importables depuis msvcrt; dans ces cas de figure **winebuild** ne va pas les résoudre et à la place, l'éditeur de liens standard **ld** les liera à la version de glibc.

Afin d'éviter les messages d'avertissement en C (et les erreurs potentielles en C++) pour n'avoir pas déclaré de prototypes, il sera peut-être nécessaire d'utiliser un ensemble de fichiers d'en-têtes compatibles MS. Leur inclusion dans Wine est prévue mais elles ne sont pas encore disponibles à l'heure où sont écrites ces lignes. En attendant, vous pouvez essayer de déclarer le propotype des fonctions dont vous avez besoin, ou simplement vous accommoder des messages d'avertissement.

Si vous avez un ensemble de fichiers d'inclusion (ou lorsqu'ils sont disponibles dans Wine), il est nécessaire d'utiliser l'option *`-isystem "chemin_d_inclusion"`* de gcc afin de lui indiquer d'utiliser de préférence vos en-têtes plutôt que celles du système local.

Pour utiliser l'option n°3, ajoutez les noms de n'importe quel symbole que vous ne voulez pas utiliser depuis msvcrt dans les fichiers `.spec` de vos applications. Par exemple, si vous vouliez les fonctions spécifiques MS, mais pas les functions d'entrées/sorties, vous pourriez avoir la liste suivante :

```
@ignore = ( fopen fclose fwrite fread fputs fgets )
```

Evidemment, la liste complète serait beaucoup plus longue. Souvenez-vous également que certaines fonctions son implémentées avec une barre de soulignement [underscore] dans leur nom et définies (avec `#define`) par ce même nom dans les en-têtes MS. Ainsi, il vous sera peut-être nécessaire de retrouver le nom en examinant `dlls/msvcrt/msvcrt.spec` afin d'obtenir le nom correct à indiquer dans la clause `@ignore`.

2.3. Problèmes de compilation

Si vous obtenez des références indéfinies vers les appels à l'API Win32 au moment de construire votre application : si vous avez un fichier `.dsp` de VC++, vérifiez dans celui-ci tous les fichiers `.lib` qu'il importe, et ajoutez les au fichier `.spec` de vos applications. **winebuild** renvoie un avertissement pour tous les fichiers importés non utilisés ; vous pourrez donc effacer ceux dont vous n'avez pas besoin plus tard. A défaut, importez simplement toutes les DLL que vous pouvez trouver dans le répertoire `dlls` de l'arborescence des sources de Wine.

Si vous n'avez pas les GUID au moment de l'édition des liens, ajoutez *`-lwine_uuid`* à la ligne d'édition des liens.

gcc est plus strict que VC++, en particulier pour compiler du C++. Vous pourriez alors avoir besoin d'ajouter des transtypages explicites dans votre code C++ pour éviter les ambigüités de surcharges entre des types similaires (comme par exemple deux surcharges qui prennent respectivement un entier et un caractère en paramètre).

Si vous rencontrez des différences entre les en-têtes Windows et celles de Wine qui rendent impossible la compilation, vous pourrez vous en sortir en demandant de l'aide à l'adresse <wine-devel@winehq.org>.

Chapitre 3. Le kit de développement Winelib

3.1. Winemaker

3.1.1. Compatibilité avec les projets Visual C++

Hélas, Winemaker ne supporte pas (encore) les fichiers de projets Visual C++. La compatibilité avec les fichiers de projets Visual C++ (les fichiers `.dsp` et quelques fichiers `.mak` pour les versions antérieures de Visual C++) est une des priorités sur la liste des améliorations importante à faire de Winemaker car cela permettra de détecter correctement les définitions à utiliser, les chemins d'inclusions personnalisés quels qu'ils soient, la liste des bibliothèques dont on devra éditer les liens, et quelles sources exactement utiliser pour construire une application cible spécifique. Tout ce que la version actuelle de Winemaker doit deviner ou que vous devez lui indiquer, comme cela apparaitra clairement dans la section suivante.

Le moment venu, Winemaker et son système de construction associé auront besoin de quelques extensions pour supporter :

- les définitions et chemins d'inclusion pour chaque fichier. Les projets Visual C++ donnent à l'utilisateur la possibilité de spécifier les options du compilateur pour chaque fichier particulier à compiler. Mais cela n'est probablement pas très fréquent donc il est possible que ce ne soit pas si important.
- configurations multiples. Les projets Visual C++ ont habituellement au moins une configuration 'Debug' et une configuration 'Release' qui sont compilées avec des options de compilation différentes. La manière exacte de traiter ces configurations reste à déterminer.

3.1.2. Analyse des sources de Winemaker

Winemaker peut faire son travail même sans la présence au départ d'un fichier de compilation (makefile) Windows ou d'un projet Visual Studio (de toute façon, il ne saurait pas quoi faire d'un fichier de compilation Windows). Cela implique de faire beaucoup de suppositions bien informées qui pourraient être fausses. Mais dans l'ensemble ça marche. Le but de cette section est de décrire plus en détails comment winemaker procède afin que vous puissiez mieux comprendre pourquoi il commet des erreurs et comment les corriger/éviter.

Au niveau noyau de l'application, winemaker effectue un passage récursif de l'arborescence de vos sources en recherchant les cibles (les choses à construire) et les fichiers sources. Commençons par les cibles.

D'abord il y a les exécutables et les DLL. A chaque fois qu'il trouve l'une d'entre elles dans un répertoire, winemaker la met dans une liste des choses à construire et génèrera plus tard un fichier

`Makefile.in` dans ce répertoire. Notez que Winemaker connaît aussi les répertoires `Release` et `Debug` couramment utilisés , et il va donc attribuer les exécutables et les bibliothèques qu'il y trouve dans leur répertoire parent. Quand il trouve un exécutable ou une DLL, winemaker est content car ceux-ci lui donnent plus d'informations que les cas décrits ci-dessous.

S'il ne trouve pas un exécutable ou une DLL, winemaker cherchera les fichiers portant l'extension `.mak`. S'il ne s'agit pas de projets Visual C++ déguisés (et dans l'état actuel des choses même si c'en est un), winemaker considèrera qu'une cible de ce nom devra être générée dans ce répertoire. Mais il ne saura jamais s'il s'agit d'un exécutable ou d'une bibliothèque. Il considèrera donc qu'il est du type défini par défaut, c'est à dire une application graphique, que vous pouvez outrepasser en utilisant les options `--cuiexe` et `--dll`.

Enfin, winemaker vérifiera s'il existe un fichier nommé `makefile`. S'il y en a un, il considèrera qu'il y a exactement une cible à générer pour ce répertoire. Mais il ne connaîtra pas le nom ou le type de cette cible. Pour le type, il fera la même chose que dans le cas décrit plus haut. Et pour le nom, il utilisera le nom du répertoire. En fait, si le répertoire commence par `src`, winemaker essaiera d'utiliser le nom du répertoire parent à la place.

Une fois que la liste des cibles a été établie pour un répertoire, winemaker vérifiera s'il contient un mélange d'exécutables et de bibliothèques. Si c'est le cas, alors winemaker fera en sorte que chaque exécutable soit lié avec toutes les bibliothèques de ce répertoire.

Si les deux étapes précédentes ne produisent pas le résultat escompté (ou si vous pensez qu'elles n'y parviendront pas), il faudrait alors mettre winemaker en mode interactif (voir *Le mode interactif*). Cela vous permettra de spécifier la liste des cibles (et plus) pour chaque répertoire.

Dans chaque répertoire, winemaker recherche aussi des fichiers sources : C, C++ ou des fichiers de ressources. S'il trouve également des cibles à construire dans ce répertoire, il essaiera d'attribuer à chaque fichier source une de ces cibles en se basant sur leurs noms. Les fichiers sources qui ne semblent pas correspondre à une cible spécifique sont mis dans une liste globale pour ce répertoire, que l'on peut voir dans les variables `EXTRA_xxx` dans le fichier `Makefile.in`, et reliés à chacune des cibles. On considère ici que ces fichiers sources contiennent un code commun, partagé par toutes les cibles. Si aucune cible n'a été trouvée dans le répertoire où sont situés ces fichiers, ils sont attribués au répertoire parent. Donc, si une cible est trouvée dans le répertoire parent, celle-ci va aussi 'hériter' des fichiers sources trouvées dans ses sous-répertoires.

Enfin, winemaker recherche aussi des fichiers plus exotiques comme les en-têtes `.h`, les fichiers `.inl` contenant des fonctions incorporées [inline] et quelques autres. Ceux-ci ne sont pas placés dans des listes de fichiers sources habituelles puisqu'ils ne sont pas compilés directement. Mais winemaker les mémorisera quand même afin qu'ils soient traités au moment venu de corriger les fichiers sources.

La correction des fichiers sources est effectuée dès que winemaker a terminé le parcours récursif du répertoire. Les deux tâches principales dans cette étape sont la correction des problèmes de retour à la ligne et retour chariot [CRLF] et la vérification de la casse des directives d'inclusion.

Winemaker réalise une copie de sauvegarde de chaque fichier source (de telle sorte que les liens symboliques sont préservés), puis lit en corrigeant les problèmes de retour à la ligne et de retour chariot et les autres problèmes au fur et à mesure. Une fois qu'il a terminé sur un fichier, il vérifie s'il a réalisé des modifications qui ne sont pas en rapport avec les retours à la ligne et les retours chariot et détruit le fichier de copie de sauvegarde si ce n'est pas le cas (ou si vous avez utilisé `--nobackup`).

La vérification de la casse des directives d'inclusion (quelle que soit la forme, y compris les fichiers référencés par des fichiers de ressources), est faite dans le contexte du projet de ce fichier source. De cette manière, winemaker peut utiliser le chemin d'inclusion correct lorsqu'il recherche le fichier qui est inclus. Si winemaker ne parvient pas à trouver un fichier dans l'un des répertoires du chemin d'inclusion, il le renommera en minuscules en se basant sur le fait qu'il s'agit très probablement d'une en-tête système et que toutes les en-têtes systèmes ont des noms en minuscules (ceci peut être redéfini en utilisant la commande `--nolower-include`).

Finalement, winemaker génère les fichiers `Makefile.in` ainsi que d'autres fichiers associés (fichiers emballeurs, fichiers spec, `configure.in`, `Make.rules.in`). D'après la description précédente, vous pouvez deviner les éléments pour lesquels winemaker peut échouer durant cette phase : définitions de macro, chemin d'inclusion, chemin d'une DLL, DLL à importer, chemin d'une bibliothèque, bibliothèques à assembler. Vous pouvez traiter ces problèmes en utilisant les options `-D`, `-P`, `-i`, `-I`, `-L` et `-l` de winemaker si elles sont suffisamment homogènes entre toutes vos cibles. Sinon, vous pourriez utiliser le mode interactif de Winemaker afin que vous puissiez spécifier les différentes configurations pour chaque projet/cible.

Par exemple, un des problèmes que vous risquez de rencontrer est celui de la macro `STRICT`. Certaines applications ne vont pas compiler si `STRICT` n'est pas activée, et d'autres ne vont pas compiler si elle l'est. Par chance, tous les fichiers dans une arborescence source donnée utilisent la même configuration de sorte que tout ce que vous avez à faire est d'ajouter `-DSTRICT` à la ligne de commande winemaker ou dans le(s) fichier(s) `Makefile.in`.

Enfin, les raisons les plus fréquentes pour lesquelles les symboles manquent ou se trouvent dupliqués sont les suivantes :

- La cible n'importe pas l'ensemble de DLL qu'il faut, ou n'est pas liée avec la bonne collection de bibliothèques. Vous pouvez éviter cela en utilisant les options `-P`, `-i`, `-L` et `-l` de winemaker ou en ajoutant ces DLL et ces bibliothèques au(x) fichier(s) `Makefile.in`.
- Il est possible que vous ayez plusieurs cibles dans un seul répertoire et que winemaker n'ait pas deviné correctement la correspondance des fichiers sources avec les cibles. La seule manière de corriger ce problème est d'éditer le fichier `Makefile.in` manuellement.
- winemaker considère que vous avez organisé vos fichiers source de manière hiérarchique. Si une cible utilise des fichiers source qui se situent dans un répertoires de même niveau, par exemple si vous éditez les liens avec `../bonjour/toutlemonde.o`, vous aurez des symboles manquants. Encore une fois, la seule solution est d'éditer manuellement le fichier `Makefile.in`.

3.1.3. Le mode interactif

Ce que c'est, quand et comment l'utiliser.

3.1.4. Les fichiers Makefile.in

Le fichier `Makefile.in` est votre fichier de compilation. Plus précisément, il s'agit du modèle à partir duquel le fichier de compilation réel sera généré au moyen du script `configure`. Il repose également sur le fichier `Make.rules` pour la grande partie de la logique en elle-même. De cette manière, il contient seulement une description relativement simple de ce qui a besoin d'être construit, et non la logique complexe qui définit comment les choses sont effectivement construites.

C'est donc le fichier à modifier si vous voulez personnaliser les choses. Voici une description détaillée de son contenu :

```
### Generic autoconf variables

TOPSRCDIR             = @top_srcdir@
TOPOBJDIR             = .
SRCDIR                = @srcdir@
VPATH                 = @srcdir@
```

Le code ci-dessus fait partie du passe-partout standard d'autoconf. Ces variables permettent d'avoir un répertoire par architecture pour les fichiers compilés et autres bonnes choses du même genre (Mais notez que ce genre de fonctionnalité n'a pas encore été testé avec les fichiers `Makefile.in` générés par winemaker).

```
SUBDIRS               =
DLLS                  =
EXES                  = hello.exe
```

Il s'agit de l'endroit où se trouve la liste des cibles pour ce répertoire. Les noms se passent tout à fait d'explication. `SUBDIRS` n'est habituellement présent que dans le fichier de compilation du répertoire du plus haut niveau. Pour les bibliothèques et les exécutables, indiquez le nom complet, y compris l'extension `.dll` ou `.exe`. Notez que ces noms doivent tous être en minuscule.

```
### Global settings

DEFINES               = -DSTRICT
INCLUDE_PATH          =
DLL_PATH              =
LIBRARY_PATH          =
LIBRARIES             =
```

Cette section contient les paramètres de compilation globaux : ils s'appliquent à toutes les cibles de ce fichier de compilation. La variable `LIBRARIES` (bibliothèques) permet de spécifier des librairies UNIX supplémentaires à lier à l'application. Notez qu'il ne faudrait normalement pas spécifier les bibliothèques Winelib à cet endroit. Pour lier une bibliothèque Winelib, on utilise les variables `DLLS` du fichier de compilation, à l'exception des bibliothèques C++ où l'on a actuellement d'autre choix que de les lier au sens Unix. Une bibliothèque que vous risquez fort de trouver ici est `mfc` (notez, le '-l' est omis).

Les autres noms de variables devraient se passer d'explication. Vous pouvez aussi utiliser trois variables supplémentaires qui ne sont pas habituellement présentes dans ce fichier : `CEXTRA`, `CXXEXTRA` et `WRCRXTRA` qui permettent de spécifier des balises [flags] supplémentaires respectivement pour le compilateur C, le compilateur C++ et le compilateur de ressources. Enfin, notez que toutes ces variables contiennent le nom de l'option.

Ensuite, on trouve une section par cible, chacune d'entre elles décrivant les divers composants dont la cible est constituée.

```
### hello.exe sources and settings

hello_exe_C_SRCS          = hello.c
hello_exe_CXX_SRCS        =
hello_exe_RC_SRCS         =
hello_exe_SPEC_SRCS       =
```

Chaque section commencera par un commentaire indiquant le nom de la cible. Ensuite vient une série de variables préfixées par le nom de la cible. Notez que le nom du préfixe peut être légèrement différent de celui de la cible à cause des restrictions sur les noms de variables.

Les variables ci-dessus donnent la liste des sources qui sont utilisées pour générer la cible. Notez qu'il devrait y avoir seulement un fichier de ressource dans `RC_SRCS`, que `SPEC_SRCS` sera habituellement vide pour les exécutables, et contiendra un simple fichier spec pour les bibliothèques.

```
hello_exe_DLL_PATH        =
hello_exe_DLLS            =
hello_exe_LIBRARY_PATH    =
hello_exe_LIBRARIES       =
hello_exe_DEPENDS         =
```

Les variables ci-dessus spécifient comment sont liées les cibles. Notez qu'elles s'ajoutent aux paramètres globaux que nous avons vus au début de ce fichier.

Le champs `DLLS` est l'endroit où vous pourrez énumérer la liste des DLL que l'exécutable importe. Il doit contenir le nom de DLL complet, y compris l'extension `.dll`, mais pas l'option `-l`.

`DEPENDS`, lorsqu'il est présent, spécifie une liste d'autres cibles dont cette cible dépend. Winemaker remplira automatiquement ce champs quand un exécutable et une bibliothèque sont construits dans le même répertoire.

```
hello_exe_OBJS        = $(hello_exe_C_SRCS:.c=.o) \
                        $(hello_exe_CXX_SRCS:.cpp=.o) \
                        $(EXTRA_OBJS)
```

L'instruction ci-dessus contruit just une liste de tous les fichiers objets qui correspondent à la cible. Cette liste est utilisée plus tard pour la commande d'édition des liens.

```
### Global source lists

C_SRCS                = $(hello_exe_C_SRCS)
CXX_SRCS              = $(hello_exe_CXX_SRCS)
RC_SRCS               = $(hello_exe_RC_SRCS)
SPEC_SRCS             = $(hello_exe_SPEC_SRCS)
```

Cette section construit des listes 'récapitulatives' des fichiers source. Ces listes sont utilisées par le fichier `Make.rules`.

Note : CORRIGEZMOI : Le paragraphe suivant n'est pas à jour.

```
### Generic autoconf targets

all: $(DLLS:%=%.so) $(EXES:%=%.so)

@MAKE_RULES@

install::
        for i in $(EXES); do $(INSTALL_PROGRAM) $$i $(bindir); done
        for i in $(EXES:%=%.so) $(DLLS); do $(INSTALL_LIBRARY) $$i $(libdir); done

uninstall::
        for i in $(EXES); do $(RM) $(bindir)/$$i;done
        for i in $(EXES:%=%.so) $(DLLS); do $(RM) $(libdir)/$$i;done
```

Le bloc de code ci-dessus définit tout d'abord la cible par défaut de ce fichier de compilation. Cela consiste ici à essayer de construire toutes les cibles. Ensuite, il inclut le fichier `Make.rules` qui contient la logique de construction, et fournit quelques cibles standard de plus pour installer / désinstaller les cibles.

```
### Target specific build rules

$(hello_SPEC_SRCS:.spec=.tmp.o): $(hello_OBJS)
        $(LDCOMBINE) $(hello_OBJS) -o $@
```

```
        -$(STRIP) $(STRIPFLAGS) $@

$(hello_SPEC_SRCS:.spec=.spec.c): $(hello_SPEC_SRCS:.spec) $(hello_SPEC_SRCS:.spec=.tmp.o)
        $(WINEBUILD) -fPIC $(hello_LIBRARY_PATH) $(WINE_LIBRARY_PATH) -sym $(hello_SPEC_SRC

hello.so: $(hello_SPEC_SRCS:.spec=.spec.o) $(hello_OBJS) $(hello_DEP
ENDS)
        $(LDSHARED) $(LDDLLFLAGS) -o $@ $(hello_OBJS) $(hello_SPEC_SRCS:.spec=.spec.o) $(he
        test -f hello || $(LN_S) $(WINE) hello
```

Viennent ensuite des directives supplémentaires pour éditer les liens des exécutables et des bibliothèques. Celles-ci sont quasiment standard et vous ne devriez pas avoir besoin de les modifier.

3.1.5. Le fichier Make.rules.in

Ce qui se trouve à l'intérieur du fichier Make.rules.in...

3.1.6. Le fichier configure.in

Ce qui se trouve dans le fichier configure.in...

3.2. Compilation des fichiers de ressources : WRC

Pour compiler des ressources, il faudrait utiliser le compilateur de ressouce de Wine [Wine Resource Compiler], wrc pour les intimes, qui produit un fichier binaire `.res`. Ce fichier de ressource est ensuite utilisé par winebuild lors de la compilation du fichier spec (voir *Le fichier spec*).

Encore une fois, les fichiers de compilation générés par winemaker s'en occupent pour vous. Mais dans le cas où vous écririez votre propre fichier de compilation, vous mettriez quelque chose comme ça :

```
WRC=$(WINE_DIR)/tools/wrc/wrc

WINELIB_FLAGS = -I$(WINE_DIR)/include -DWINELIB -D_REENTRANT
WRCFLAGS      = -r -L

.SUFFIXES: .rc .res

.rc.res:
$(WRC) $(WRCFLAGS) $(WINELIB_FLAGS) -o $@ $<
```

Il existe deux problèmes que vous êtes susceptibles de rencontrer avec les fichiers de ressource.

Le premier problème est avec les en-têtes des bibliothèques C. WRC ne sait pas où sont situées ces en-têtes. Donc, si un fichier RC, d'un fichier qu'il inclut, référence une telle en-tête, vous obtiendrez une erreur 'file not found' (fichier non trouvé) provenant de wrc. Voici quelques solutions pour s'en sortir :

- La solution traditionnellement utilisée par les en-têtes Winelib est d'adjoindre la directive d'inclusion qui pose problème dans une directive `#ifndef RC_INVOKED` où `RC_INVOKED` est le nom d'une macro qui est automatiquement définie par wrc.
- Ou encore, vous pouvez ajouter une ou plusieurs directives `-I` à votre commande wrc afin que celui-ci trouve vos fichiers système. Par exemple, vous pourriez utiliser `-I/usr/include -I/usr/lib/gcc-lib/i386-linux/2.95.2/include` pour approvisionner les en-têtes C ainsi que C++. Mais cela suppose que vous sachiez où ces fichiers d'en-têtes résident, ce qui diminue la portabilité de vos fichiers de compilation vers d'autres plateformes (à moins que vous ne détectiez automatiquement tous les répertoires nécessaires dans votre script autoconf).

  Ou bien vous pourriez utiliser le compilateur C/C++ afin de réaliser le prétraitement. Pour cela, modifiez simplement votre fichier de compilation comme suit :

  ```
  .rc.res:
  $(CC) $(CC_OPTS) -DRC_INVOKED -E -x c $< | $(WRC) -N $(WRCFLAGS) $(WINELIB_FLAGS) -o $@
  ```

Le second problème est que les en-têtes pourraient contenir des constructions syntaxiques que WRC ne parvient pas à comprendre. Un exemple type est une fonction qui retourne un type 'const'. WRC s'attend à ce qu'une fonction soit représentée par deux identifiants suivis par une parenthèse ouvrante. Avec le 'const', on a trois identifiants suivis d'une parenthèse et WRC est ainsi dans la confusion (note : En fait, WRC devrait ignorer tout cela comme le fait le compilateur de ressources windows). Actuellement, pour contourner le problème on joint la (ou les) directive(s) incriminée(s) dans un `#ifndef RC_INVOKED`.

L'utilisation des fichiers GIF dans les ressources est problématique. Pour de meilleurs résultats, convertissez les en BMP et changez votre fichier `.res`.

Si vous utilisez des dialogues/contrôles communs dans vos fichiers de ressource, il vous faudra ajouter `#include <commctrl.h>` après la ligne `#include <window.h>`, afin que **wrc** connaisse les valeurs des balises [flags] de contrôle spécifiques.

3.3. Compilation des fichiers de message : WMC

Comment l'utilise t-on ???

3.4. Le fichier spec

3.4.1. Introduction

Sous Windows, la vie d'une application commence soit lorsque son programme principal est appelé `main`, pour une application en mode console, ou lorsque sa fenêtre principale `Winmain` est appelée, pour les applications windows dans le sous-système 'windows'. Sous UNIX, c'est toujours le `main` qui est appelé. De plus, dans Winelib, il y a toujours des tâches particulières à accomplir, telles que l'initialisation de Winelib, qu'un programme principal `main` normal n'a pas à faire.

De plus, les applications et les bibliothèques windows contiennnent des informations qui sont nécessaires pour que des API telles que `GetProcAdress` fonctionnent. Il est donc nécessaire de dupliquer ces structures de données dans le monde UNIX pour que ces mêmes API fonctionnent avec les applications et les bibliothèques Winelib.

Le fichier Spec est là pour combler le fossé sémantique décrit plus haut. Il fournit une fonction `main` qui initialise Winelib et qui appelle le `WinMain`/ `DllMain` du module, et il contient des informations au sujet de chaque API exportée depuis une Dll afin que les tables appropriées puissent être générées.

Un fichier spec habituel ressemblera à peu près à ça :

```
init    WinMain
rsrc    resource.res
```

Et voici les entrées que vous aurez sans doute envie de changer :

init

`init` définit quel est le point d'entrée de ce module. On a coutume de fixer sa valeur à `Dllmain` pour une bibliothèque, `main` pour une application en mode console et `WinMain` pour une application graphique.

rsrc

Cet élément spécifie le nom du fichier de ressource compilé à relier à votre module. Si votre fichier de ressource s'appelle `bonjour.rc`, alors l'étape de compilation wrc (voir *Compilation des fichiers de ressources : WRC*) génèrera un fichier appelé `bonjour.res`. Il s'agit du nom que vous devez renseigner ici. Notez qu'à cause de cela, vous ne pouvez pas compiler le fichier spec avant d'avoir compilé le fichier de ressource. Vous devriez donc mettre dans votre fichier de compilation une règle comme ceci :

```
bonjour.spec.c: bonjour.res
```

Si votre projet ne comporte pas de fichier de ressource, vous devez totalement omettre cette entrée.

@

Note : CORRIGEZMOI : Vous devez maintenant exporter les fonctions depuis des DLL.

Cette entrée n'est pas présentée plus haut car elle n'est pas toujours nécessaire. En fait Il est nécessaire d'exporter des fonctions seulement quand vous comptez charger la bibliothèque dynamiquement avec `LoadLibrary` et faire un `GetProcAddress` sur ces fonctions. Ce n'est pas nécessaire si vous comptez juste éditer les liens avec la bibliothèque et appeler les fonctions normalement. Pour plus d'information à ce sujet : *Informations supplémentaires*

3.4.2. Compilation

Note : CORRIGEZMOI : Cette section n'est plus dutout à jour et ne décrit pas correctement l'utilisation actuelle de winebuild et des fichiers spec. En particulier, avec les versions récentes de winebuild, la plupart des informations qui étaient habituellement dans les fichiers spec est maintenant spécifiée dans la ligne de commande.

La compilation d'un fichier spec est un processus en 2 étapes. Celui-ci est d'abord converti en un fichier C par winebuild, et ensuite compilé en un fichier objet en utilisant un compilateur C classique. Ceci est entièrement pris en charge par les fichiers de compilation de winemaker évidemment. Mais voici ce qu'il aimerait bien qu'on lui indique si l'on devait le faire à la main :

```
WINEBUILD=$(WINE_DIR)/tools/winebuild

.SUFFIXES: .spec .spec.c .spec.o

.spec.spec.c:
$(WINEBUILD) -fPIC -o $@ -spec $<

.spec.c.spec.o:
$(CC) -c -o $*.spec.o $<
```

Il n'y a rien de bien compliqué là dedans. Il ne faut juste pas oublier l'instruction `.SUFFIXES`, faire attention au caractère de tabulation si vous faites un copier-coller de ce morceau de code directement dans votre fichier de compilation [Makefile].

3.4.3. Informations supplémentaires

Voici une description plus détaillée du format de fichier spec.

```
# comment text
```

Tout ce qui se trouve après un '#' correspond à un commentaire et sera ignoré.

```
init    FUNCTION
```

Ce champ est facultatif et spécifique aux modules Win32. Il spécifie une fonction qui sera appelée lorsque la DLL est chargée ou qu'un exécutable est démarré.

```
rsrc    RES_FILE
```

Ce champ est facultatif. S'il est présent, il spécifie le nom d'un fichier `.res` contenant les ressources compilées. Voir : *Compilation des fichiers de ressources : WRC* pour plus d'information sur la compilation d'un fichier de ressource.

```
ORDINAL VARTYPE EXPORTNAME (DATA [DATA [DATA [...]]])
2 byte Variable(-1 0xff 0 0)
```

Ce champ peut être présent zéro fois ou plus. Chaque instance définit un stockage de données à l'ordinal spécifié. Vous pouvez stocker des éléments en octets, en mots de 16 bits ou en mots de 32 bits. `ORDINAL` est remplacé par le nombre ordinal correspondant à la variable. `VARTYPE` sera `byte`, `word`, ou `long` pour respectivement 8, 16 et 32 bits. `EXPORTNAME` sera le nom disponible pour l'édition dynamique des liens. `DATA` peut être un nombre décimal ou un nombre hexadécimal précédé par "0x". L'exemple définit la variable `Variable` à l'ordinal 2 et contenant 4 octets.

```
ORDINAL equate EXPORTNAME DATA
```

Ce champ peut être présent zéro fois ou plus. Chaque instance définit un ordinal en valeur absolue. On remplace `ORDINAL` par le nombre ordinal correspondant à la variable. `EXPORTNAME` sera le nom disponible pour l'édition dynamique des liens. `DATA` peut être un nombre décimal ou un nombre hexadécimal précédé par "0x".

```
ORDINAL FUNCTYPE EXPORTNAME([ARGTYPE [ARGTYPE [...]]]) HANDLERNAME
100 pascal CreateWindow(ptr ptr long s_word s_word s_word s_word
                        word word word ptr)
           WIN_CreateWindow
101 pascal GetFocus() WIN_GetFocus()
```

Ce champ peut être présent zéro fois ou plus. Chaque instance définit le point d'entrée d'une fonction. Le prototype défini par `EXPORTNAME ([ARGTYPE [ARGTYPE [...]]])` spécifie le nom disponible pour l'édition dynamique des liens et le format des arguments. On remplace `ORDINAL` par le nombre ordinal correspondant à la fonction, ou `@` pour une allocation dynamique de l'ordinal (pour Win32 seulement).

`FUNCTYPE` sera l'une des expressions suivantes :

pascal16
: pour une fonction Win16 retournant une valeur sur 16 bits

pascal
: pour une fonction Win16 retournant une valeur sur 32 bits

register
: pour une fonction utilisant le registre du processeur pour passer des arguments

interrupt
: pour les routines d'interruptions Win16

stdcall
: pour une fonction Win32 normale

cdecl
: pour une fonction Win32 utilisant la convention d'appel en C

varargs
: pour une fonction Win32 prenant un nombre variable d'arguments

`ARGTYPE` sera l'une des expressions suivantes :

word
: pour un mot de 16 bits

long
: une valeur en 32 bits

ptr
: pour un pointeur linéaire

str
: pour un pointeur linéaire sur une chaine de caractères terminée par null

s_word
: pour un mot de 16 bits signé

segptr

pour un pointeur segmenté

segstr

pour un pointeur segmenté vers une chaîne de caractères finissant par null

wstr

pour un pointeur linéaire vers une chaîne de caractères large (en 16 bits unicode) finissant par null

Seuls `ptr`, `str`, `wstr` et `long` sont valables pour les fonctions Win32. `HANDLERNAME` est le nom de la fonction Wine qui va effectivement traiter la requête en mode 32 bits.

Les chaînes de caractères doivent toujours correspondre à str, les chaînes larges à wstr. En règle générale cela dépend si le paramètre est en entrée [IN], sortie [OUT] ou entrée/sortie [IN/OUT].

- Entrée [IN]: str/wstr
- Sortie [OUT]: ptr
- Entrée/Sortie [IN/OUT]: str/wstr

C'est pour les messages de correction de défauts [debug]. S'il s'agit d'un paramètre en sortie, il se pourrait qu'il ne soit pas initialisé et de ce fait, il ne devrait pas s'afficher comme une chaîne de caractères.

Les deux exemples définissent un point d'entrée pour les appels `CreateWindow` et `GetFocus` respectivement. Les ordinaux utilisés sont juste des exemples.

Pour déclarer une fonction utilisant un nombre variable d'arguments en Win16, spécifiez la fonction comme ne prenant pas d'arguments. Les arguments sont alors accessibles avec `CURRENT_STACK16->args`. En Win32, spécifiez la fonction comme étant de type `varargs` et déclarez la avec un paramètre '...' dans le fichier C. Voir les fonctions wsprintf* dans `user.spec` et `user32.spec` pour trouver un exemple.

```
ORDINAL stub EXPORTNAME
```

Ce champ peut être présent zéro fois ou plus. Chaque instance définit une fonction souche [stub]. Il rend l'ordinal accessible pour l'édition dynamique des liens, mais terminera l'exécution avec un message d'erreur si par hasard on appelle la fonction.

```
ORDINAL extern EXPORTNAME SYMBOLNAME
```

Ce champ peut être présent zéro fois ou plus. Chaque instance définit une entrée qui établit simplement une correspondance avec un symbole Wine (variable ou fonction); `EXPORTNAME` pointera sur le symbole `SYMBOLNAME` qu'il faut définir dans le code C. Ce type fonctionne seulement avec Win32.

```
ORDINAL forward EXPORTNAME SYMBOLNAME
```

Ce champs peut être présent zéro fois ou plus. Chaque instance définit une entrée qui est retransmise à un autre point d'entrée (un genre de lien symbolique). `EXPORTNAME` sera retransmis au point d'entrée `SYMBOLNAME` qui doit être de la forme `DLL.Function`. Ce type fonctionne seulement avec Win32.

3.5. Relier tout ensemble

Note : CORRIGEZMOI : La partie suivante n'est pas à jour.

Pour éditer les liens d'un exécutable, il est nécessaire de relier ensemble : vos fichiers objets, le fichier spec, tout bibliothèque Windows dont votre application dépend, gdi32 par exemple, et toute bibliothèque supplémentaire que vous utilisez. Toutes les bibliothèques auxquelles vous reliez votre application doivent être disponibles en tant que bibliothèques '.so'. Si l'une d'entre elle n'est disponible que sous la forme '.dll', consultez alors la section *Construction des DLL Winelib*.

C'est aussi lorsque vous tenterez d'éditer les liens de votre exécutable que vous découvrirez si vous avez des symboles manquants ou non dans vos bibliothèques personnalisées. Sous Windows, quand vous construisez une bibliothèque, l'éditeur de liens vous dira automatiquement si un symbole qu'il est supposé exporter est indéfini. Sous Unix, et dans Winelib, ce n'est pas le cas. Le symbole sera discrètement marqué comme indéfini et c'est seulement quand vous essaierez de produire un exécutable que l'éditeur de liens vérifiera si tous les symboles correspondent à quelque chose.

Donc, avant de crier victoire lors de la première conversion d'une bibliothèque vers Winelib, vous devriez essayer de la relier à un exécutable (mais vous l'auriez quand même fait pour la tester, hein ?). A ce stade, vous pourriez découvrir quelques symboles indéfinis que vous pensiez être implémentés par la bibliothèque. Alors vous devez aller dans les sources de la bibliothèque et corriger ça. Mais vous pourriez aussi découvrir que les symboles manquants sont définis dans, disons, gdi32. C'est parce que vous n'avez pas relié la-dite bibliothèque avec gdi32. Une manière de corriger cela est de relier cet exécutable, et tout autre exécutable qui utilise votre bibliothèque, avec gdi32. Mais il est préférable de revenir au fichier de compilation de votre bibliothèque et la relier explicitement avec gdi32.

Comme vous pourrez rapidement le remarquer, cela n'a malheureusement pas été (complètement) fait pour les propres bibliothèques de Winelib. Donc, si une application doit être reliée avec ole32, il vous faudra également la relier avec advapi32, rpcrt4 et d'autres même si vous ne les utilisez pas directement. Cela peut être pénible et espérons-le, ce sera corrigé bientôt (n'hésitez pas à soumettre un correctif).

Chapitre 4. Utilisation de la MFC

4.1. Introduction

Pour utiliser la MFC dans une application Winelib vous devrez d'abord recompiler la MFC avec Winelib. En théorie, il devrait être possible d'écrire un enveloppeur [wrapper] pour la MFC de Windows comme cela est décrit dans la section *Construction des DLL Winelib*. Mais en pratique, cela ne semble pas être une approche réaliste pour la MFC :

- Le très grand nombre d'API fait que l'écriture d'un enveloppeur représente une lourde tâche en soi.
- De plus, la MFC contient un très grand nombre d'API qui ne sont pas évidentes à traiter lorsqu'il s'agit de réaliser un enveloppeur.
- Même une fois que l'enveloppeur est écrit, il sera nécessaire de modifier les en-têtes MFC afin que le compilateur ne s'interrompe pas lorsqu'il les rencontre.
- Une grande partie du code MFC est en réalité dans votre application sous la forme de macros. Cela signifie qu'encore plus d'en-têtes MFC doivent en fait fonctionner afin que vous soyez capable de compiler une application basée sur MFC.

C'est pourquoi une section de ce guide est consacrée aux problèmes de la compilation de la MFC avec Winelib.

4.2. Aspects juridiques

Le but de cette section est de vous faire prendre conscience des problèmes légaux potentiels. Veillez bien à lire vos licences et à consulter vos avocats. Dans tous les cas, vous ne devez pas prendre pour argent comptant le reste de cette section puisqu'elle n'a pas été écrite par un avocat.

Pendant la compilation de votre programme, vous allez intégrer du code à partir de plusieurs sources : votre code, le code de Winelib, le code de la MFC Microsoft, et eventuellement du code provenant d'autres sources. Ainsi, vous devez vous assurer que les licences de tous les code sources sont respectées. Ce que vous êtes autorisé ou non à faire peut varier selon la manière dont vous intégrez le code et si vous le distribuez. Par exemple, si vous publiez votre code sous la licence GPL ou LGPL, vous ne pouvez pas utiliser la MFC car ces licences n'autorisent pas le code qu'elles couvrent à dépendre de bibliothèques dont les licences ne sont pas compatibles. Il y a une solution de rechange dans la licence pour votre code, vous pouvez faire une exception pour la bibliothèque MFC. Pour plus d'infomation, allez voir la FAQ GPL GNU (http://www.gnu.org/licenses/gpl-faq.html)

Wine/Winelib est distribué sous la Licence Publique Générale GNU Limitée [GNU Lesser General Public licence]. Vous pouvez aller voir cette licence pour connaître les restrictions sur les modifications et la distribution du code de Wine/Winelib. En général, il est possible de respecter ces restrictions dans

n’importe quel type d’application. D’un autre côté, MFC est distribuée sous une licence très restrictive et les restrictions varient d’une version à une autre et entre les ensembles de modifications provisoires [service pack]. En gros, il y a trois aspects auxquels vous devez être sensible quand vous utilisez la MFC.

D’abord, vous devez acquérir le code source MFC sur votre ordinateur légalement. Le code source MFC est fourni en tant que partie de Visual Studio. La licence pour Visual Studio implique qu’il s’agit d’un produit unique qui ne peut pas être divisé selon ses composants. Donc, la manière la plus appropriée d’installer MFC sur votre système est d’acheter Visual Studio et de l’installer sur une machine Linux ayant une partition d’amorçage double.

Ensuite, vous devez vérifier que vous êtes autorisé à recompiler MFC sur un système d’exploitation non-Microsoft ! Cela varie avec la version de MFC. Voici ce que l’on peut lire dans la licence MFC de Visual Studio 6.0 :

> 1.1 Concession de la licence générale. Microsoft vous accorde, en tant que particulier, une licence personnelle non-exclusive permettant de réaliser et d’utiliser des copies du LOGICIEL uniquement dans le cadre de la conception, le développement et les tests de votre (vos) logiciels(s) conçus pour fonctionner conjointement avec tout système d’exploitation Microsoft. [Autres parties sans rapport avec le sujet supprimées]

Il apparaît donc que vous ne pouvez même pas compiler MFC pour Winelib en utilisant cette licence. Heureusement, on peut lire sur le licence de l’ensemble de modifications provisoires [service pack] n°3 de Visual Studio 6.0 (la licence de Visual Studio 5.0 est identique) :

> 1.1 Concession de la licence générale. Microsoft vous accorde en tant que particulier, une licence personnelle non-exclusive permettant de réaliser et d’utiliser des copies du LOGICIEL uniquement dans le cadre de la conception, le développement et les tests de votre (vos) logiciels(s). [Autres parties sans rapport avec le sujet supprimées].

Ainsi, sous cette licence, il apparaît que vous pouvez compiler MFC pour Winelib.

Enfin, vous devez vérifier si vous avez le droit de distribuer une bibliothèque MFC. Consultez la section correspondante de la licence sur « redistribuables et vos droits de redistribution ». La licence semble spécifier que vous avez seulement le droit de distribuer les exécutables de la bibliothèque MFC si elle ne présente aucune information de correction de défaut [debug] et si vous la distribuez avec une application qui fournit une fonctionnalité significative ajoutée à la bibliothèque MFC.

4.3. Compilation de la MFC

Voici un ensemble de recommandations pour compiler la MFC avec Winelib :

Nous recommandons de lancer winemaker en mode interactif (`--interactive`) pour spécifier les options correctes pour la MFC et la partie ATL (afin que les chemins d’inclusion soient corrects, que la MFC ne soit pas considérée comme étant basée sur MFC et pour faire en sorte qu’elle construise des bibliothèques et non des exécutables).

Ensuite, lors de sa compilation, un certain nombre de macros `_AFX_NO_XXX` seront en effet nécessaires. Mais cela n'est pas suffisant et il faudra supprimer certaines choses en utilisant `#ifdef`. Par exemple, la compatibilité avec le « richedit » de Wine n'est pas très bonne. Voici les options AFX que j'utilise :

```
#define _AFX_PORTABLE
#define _FORCENAMELESSUNION
#define _AFX_NO_DAO_SUPPORT
#define _AFX_NO_DHTML_SUPPORT
#define _AFX_NO_OLEDB_SUPPORT
#define _AFX_NO_RICHEDIT_SUPPORT
```

Vous aurez également besoin de macros personnalisées pour `CMonikerFile`, `OleDB`, `HtmlView`, ...

Nous recommandons l'utilisation des en-têtes msvcrt de Wine (`- isystem $(WINE_INCLUDE_ROOT)/msvcrt`), cependant cela signifie que vous devrez désactiver temporairement le support pour winsock (retirer l'instruction `#ifdef` correspondante dans `windows.h`).

Il faut utiliser un compilateur g++ plus récent que g++ 2.95. g++ 2.95 n'est pas compatible avec les structures non-nommées alors que les versions les plus récentes le sont, et ça aide beaucoup. Voici les options intéressantes à mentionner :

- `-fms-extensions` (aide à compiler plus de code)
- `-fshort-wchar -DWINE_UNICODE_NATIVE` (facilite la compatibilité avec UNICODE)
- `-DICOM_USE_COM_INTERFACE_ATTRIBUTE` (pour faire fonctionner le code COM)

Quand vous atteindrez l'étape d'édition des liens pour la première fois, vous obtiendrez beaucoup d'erreurs de symboles indéfinis. Pour les corriger, il sera nécessaire de retourner dans la source et retirer encore des instructions `#ifdef` jusqu'à ce que vous obteniez une 'fermeture'. Il y a également certains fichiers qui n'ont pas besoin d'être compilés.

Peut-être que là, un jour, nous aurons un fichier de compilation prêt à l'emploi.

Chapitre 5. Construction des DLL Winelib

5.1. Introduction

Pour une raison ou pour une autre vous pourriez vous retrouver avec une bibliothèque Linux que vous voulez utiliser comme s'il s'agissait d'une Dll Windows. Il existe plusieurs raisons à cela dont les suivantes :

- Vous êtes en train de réaliser le portage d'une application conséquente qui utilise plusieurs bibliothèques provenant d'un tiers. L'une est disponible sous Linux mais vous n'êtes pas encore prêt à relier votre application à celle-ci directement comme une bibliothèque Linux partagée.
- Il y a une interface disponible très bien définie et il existe plusieurs solutions Linux disponibles pour cette interface (ex. : l'interface ODBC dans Wine).
- Vous avez une application Windows uniquement exécutable qui peut être étendue par des plugins, comme par exemple un éditeur de texte ou un environnement de développement intégré [IDE].

La procédure pour faire face à ces situations est en fait très simple. Il est nécessaire d'écrire un fichier spec qui décrira l'interface de la bibliothèque dans le même format qu'une Dll (principalement, quelles fonctions elle exporte). Vous aurez aussi à écrire un petit enveloppeur autour de la bibliothèque. Vous les combinez afin de former une Dll intégrée à Wine qui est reliée à la bibliothèque Linux. Ensuite, vous modifiez le 'DllOverrides' dans le fichier de configuration de Wine afin de vous assurer que c'est cette nouvelle Dll intégrée qui est appelée plutôt qu'une version windows.

Dans cette section nous allons nous regarder deux exemples. Le premier exemple est extrêmement simple et nous amène dans le vif du sujet en suivant des mini-étapes. Le second exemple est proxy de l'interface ODBC dans Wine. Les fichiers auxquels nous renvoyons pour l'exemple ODBC sont actuellement dans le répertoire `dlls/odbc32` des sources Wine.

Le premier exemple est basé de très près sur un cas réel (les noms et les fonctions etc. ont été changés afin de protéger les innocents). Une application Windows conséquente comprend une DLL qui est reliée à une DLL d'un tiers. Pour différentes raison la DLL du tiers ne fonctionne pas très bien sous Wine. Cependant, la bibliothèque du tiers est également disponible pour un environnement Linux. Chose pratique, la DLL et la bibliothèque Linux partagée exportent seulement un faible nombre de fonctions et l'application utilise seulement l'une d'entre elles.

Pour être précis, l'application appelle une fonction :

```
signed short WINAPI MyWinFunc (unsigned short a, void *b, void *c,
        unsigned long *d, void *e, int f, char g, unsigned char *h);
```

et la bibliothèque linux exporte une fonction correspondante :

```
signed short MyLinuxFunc (unsigned short a, void *b, void *c,
        unsigned short *d, void *e, char g, unsigned char *h);
```

5.2. Ecriture du fichier spec

Commencez par écrire le fichier spec. Ce fichier décrira l'interface comme s'il s'agissait d'une DLL. Vous trouverez ailleurs des informations précises sur le format d'un fichier spec. (ex. : man winebuild).

Dans l'exemple simple nous voulons une Dll intégrée à Wine qui corresponde à la Dll MyWin. Le fichier spec est `MyWin.dll.spec` et ressemble à peu près à ça :

```
#
# File: MyWin.dll.spec
#
# some sort of copyright
#
# Wine spec file for the MyWin.dll built-in library (a minimal wrapper around the
# linux library libMyLinux)
#
# For further details of wine spec files see the Winelib documentation at
# www.winehq.org

2 stdcall MyWinFunc (long ptr ptr ptr ptr long long ptr) MyProxyWinFunc

# End of file
```

Remarquez que les arguments sont marqués avec un type long même si ils sont plus petits que cela. Avec cet exemple, nous allons directement être reliés à la bibliothèque partagée Linux alors qu'avec l'exemple ODBC, nous chargerons la bibliothèque partagée Linux dynamiquement.

Dans le cas de l'exemple ODBC, vous pouvez voir ça dans le fichier `odbc32.spec`

5.3. Ecriture de l'enveloppeur

Nous regarderons en premier l'exemple simple. La principale complication ici est la liste d'arguments légèrement différente. Le paramètre f n'a pas à être passé à la fonction Linux et le paramètre d doit (théoriquement) être converti d'un `unsigned long *i` à un `unsigned short *`. On s'assure ainsi que les bits de point fort de la valeur retournée sont correctement fixés. En outre, contrairement à l'exemple ODBC, nous allons directement relier l'exemple à la bibliothèque partagée Linux.

```
/*
 * File: MyWin.c
 *
```

```
 * Copyright (c) The copyright holder.
 *
 * Basic Wine wrapper for the Linux <3rd party library> so that it can be
 * used by <the application>
 *
 * Currently this file makes no attempt to be a full wrapper for the <3rd
 * party library>; it only exports enough for our own use.
 *
 * Note that this is a Unix file; please don't go converting it to DOS format
 * (e.g. converting line feeds to Carriage return/Line feed).
 *
 * This file should be built in a Wine environment as a WineLib library,
 * linked to the Linux <3rd party> libraries (currently libxxxx.so and
 * libyyyy.so)
 */

#include < <3rd party linux header> >
#include <windef.h> /* Part of the Wine header files */

/* This declaration is as defined in the spec file.  It is deliberately not
 * specified in terms of <3rd party> types since we are messing about here
 * between two operating systems (making it look like a Windows thing when
 * actually it is a Linux thing).  In this way the compiler will point out any
 * inconsistencies.
 * For example the fourth argument needs care
 */
signed short WINAPI MyProxyWinFunc (unsigned short a, void *b, void *c,
                unsigned long *d, void *e, int f, char g, unsigned char *h)
{
    unsigned short d1;
    signed short ret;

    d1 = (unsigned short) *d;
    ret = <3rd party linux function> (a, b, c, &d1, e, g, h);
    *d = d1;

    return ret;
}

/* End of file */
```

Pour une explication plus approfondie, nous pouvons utiliser l'exemple ODBC. Il est implémenté sous forme d'un fichier d'en-tête (`proxyodbc.h`) et le fichier source C même (`proxyodbc.c`). Bien que le fichier soit assez long, sa structure est extrêmement simple.

`Dllmain` : la fonction est utilisée pour initialiser la DLL. Quand survient l'évènement d'attachement de processus, la fonction se relie dynamiquement à la bibliothèque ODBC Linux désirée (car plusieurs sont

disponibles) et construit une liste de pointeurs sur des fonctions. Elle se libère lors de l'évènement de détachement de processus.

Ensuite, chacune des fonctions appelle simplement la fonction Linux appropriée par le biais de pointeur de fonction configuré lors de l'initialisation.

5.4. Construction

Alors, comment construit-on donc la bibliothèque intégrée à Winelib ? La manière la plus simple est de faire faire à Winemaker le sale boulot à notre place. Pour l'exemple simple, nous avons deux fichiers sources (l'enveloppeur et le fichier spec). Nous avons aussi une en-tête d'un tiers et bien sûr les fichier de bibliothèque.

Mettez les deux fichiers sources dans un répertoire approprié et utilisez ensuite winemaker pour construire l'ossature, y compris le script de configuration, le fichier de compilation, etc. Vous serez amenés à utiliser les options suivantes de winemaker :

- `--nosource-fix` et `--nogenerate-specs` (requiert winemaker en version 0.5.8 ou supérieure) afin de s'assurer que les deux fichiers ne sont pas modifiés. (Si vous utilisez une version antérieure de winemaker, alors rendez les fichiers accessibles en lecture seule et ignorez les messages rabat-joie qui concernent l'impossibilité de les modifier).
- `--dll --single-target MyWin --nomfc` pour spécifier la cible
- `-DMightNeedSomething -I3rd_party_include -L3rd_party_lib -lxxxx -lyyyy` où xxxx et yyyy sont les emplacements des fichiers d'en-têtes etc.

Après le lancement de Winemaker, j'aime bien éditer le fichier Makefile.in pour ajouter la ligne `CEXTRA = -Wall` juste avant le `DEFINES=`.

Ensuite, lancez simplement le "configure" et le "make", normalement (décrit ailleurs).

5.5. Installation

Alors, comment installer le proxy et s'assurer que tout se connecte correctement ? Vous aurez une bonne marge de manoeuvre dans cette partie donc ce qui suit ne mentionne pas toutes les options disponibles.

Assurez vous que l'objet Linux partagé en question est placé quelque part où le système Linux sera en mesure de le trouver. En général, cela signifie qu'il devrait être placé dans un des répertoires mentionnés dans le fichier `/etc/ld.so.conf` ou quelque part dans le chemin spécifié par `LD_LIBRARY_PATH`. Si vous pouvez vous relier à cet objet à partir d'un programme linux, ça devrait être bon.

Mettez l'objet partagé de substitution [proxy] (`MyWin.dll.so`) au même endroit que le reste des DLL intégrées (si vous avez utilisé winemaker pour configurer votre environnement de construction alors l'exécution de **make install** en tant que root devrait le faire pour vous). Sinon, assurez vous que le `WINEDLLPATH` inclut le répertoire contenant l'objet partagé de substitution.

Si vous avez à la fois la DLL Windows et la DLL de substitution Linux, vous devrez vous assurer que c'est la bonne DLL qui reçoit les appels. La méthode la plus simple est sans doute de juste renommer la version Windows afin qu'elle ne soit pas détectée. Une solution alternative consisterait à spécifier dans la section DllOverrides (ou dans la section AppDefaults\\myprog.exe\\DllOverrides) du fichier de configuration (dans votre répertoire `.wine`) que la version intégrée soit utilisée. Notez que si la version Windows de la DLL est présente et se trouve dans le même répertoire que le fichier exécutable (à la différence d'être dans le répertoire windows), il sera nécessaire à ce moment là de spécifier le chemin complet vers la DLL, et pas simplement son nom.

Une fois que vous avez fait cela, vous devriez utiliser la bibliothèque Linux partagée avec succès. Si vous avez des problèmes, mettez la variable d'environnement `WINEDEBUG=+module` avant de lancer wine pour voir ce qui se passe en fait.

5.6. Conversiond des noms de fichiers

Supposez que vous vouliez convertir les noms de fichier au format entrant DOS vers leur équivalent Linux. Bien sûr, il n'existe pas de fonction appropriée dans la vrai API Microsoft Windows, mais Wine fournit une fonction juste pour cette tâche et l'exporte depuis sa copie du fichier Kernel32.dll. Cette fonction est `wine_get_unix_file_name` (définie dans `winbase.h`).