Programmation C sous Linux: Compléments

Pierre Senellart

Semaine *Informatique pratique*, septembre 2022

Plan

Jeux de caractères

Unicode : répertoire de caractères, assignant à chaque caractère, de quelque langue que ce soit, un entier

A	$\rightarrow$	65	ε	$\rightarrow$	949
é	$\rightarrow$	233	$\aleph$	$\rightarrow$	1488

Jeu de caractères : moyen de représenter concrètement, par une suite de 0 ou de 1, un caractère Unicode

Par exemple, pour le caractère « é » :

latin1	`11101001 (Seulement pour certains caractères)`
utf8	`11000011 10101001`
utf16	`11101001 00000000`

utf-8 présente l'avantage de pouvoir représenter tous les caractères d'Unicode, de manière compatible avec l'ancien encodage ASCII

Sous Linux, `iconv` peut convertir d'un jeu de caractères à un autre

Chaînes de caractère

- En C, les chaînes de caractères de longueur ℓ sont représentées par des tableaux de caractères de taille $\ell + 1$
- Le dernier caractère du tableau (considéré comme ne faisant pas partie de la chaîne) est le caractère de valeur numérique 0
- Un caractère C n'est pas assez grand (en général, et au moins, 1 octet) pour stocker un vrai caractère Unicode, quel qu'en soit l'encodage, vraiment fait pour stocker des octets

Chaînes Unicode

- La bibliothèque standard C définit un type `wchar_t` mais celui-ci ne garantit pas non plus de pouvoir stocker un caractère Unicode arbitraire !
- Si la macro `__STDC_ISO_10646__` est définie (ce qui est le cas sous Linux avec `gcc`), `wchar_t` est ok pour Unicode ; voir le man de `mbstowcs` pour comment procéder pour convertir un tableau d'octers en tableau de caractères Unicode

Chaînes Unicode

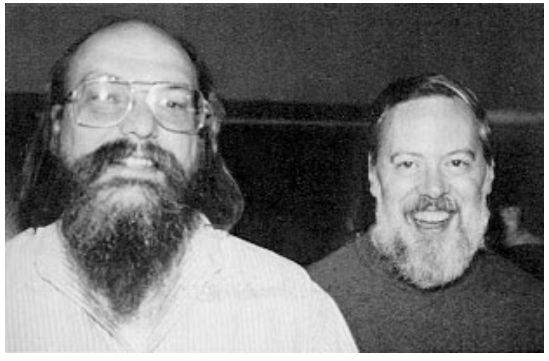
- La bibliothèque standard C définit un type `wchar_t` mais celui-ci ne garantit pas non plus de pouvoir stocker un caractère Unicode arbitraire !
- Si la macro `__STDC_ISO_10646__` est définie (ce qui est le cas sous Linux avec `gcc`), `wchar_t` est ok pour Unicode ; voir le man de `mbstowcs` pour comment procéder pour convertir un tableau d'octers en tableau de caractères Unicode
- Depuis C11, la bibliothèque standard C définit un type `char32_t` qui garantit de pouvoir stocker n'importe quel caractère Unicode
- `mbrtoc32` peut être utilisé pour convertir une chaîne d'octets (typiquement, UTF-8) en une chaîne de caractères

Plan

Historique de C (1/2)

1969 Ken Thompson & Dennis Ritchie développent B, un langage structuré vu comme une simplification du langage BPCL

1972 Dennis Ritchie raffine le langage B, et en propose une nouvelle version, renommée C



Ken Thompson & Dennis Ritchie

1978 Brian Kernighan & Dennis Ritchie publient un ouvrage de référence sur le langage C, étendu depuis, qui en donne une spécification de fait, on parle de C K&R

Historique de C (2/2)

1989 Le langage C évolue et est normalisé par l'ANSI (puis par l'ISO en 1990), devenant le C ANSI ou C89 – cœur des versions actuellement répandues

1999 Nouveau standard C99 avec nombreuses améliorations : nouveaux types `long long`, `_Bool`. `_Complex`, des tableaux automatiques de taille variable, des améliorations à la bibliothèque standard, etc.

2011 Nouveau standard C11 avec support de la programmation multi-threads, meilleur support d'Unicode (`char32_t`), etc.

2017 Standard actuel C17, pas de nouvelles fonctionnalités

Quelques nouveautés de C99 et C11 / C ANSI

- Plus besoin de déclarer les variables en début de bloc !
- En-tête `<stdbool.h>` et type booléen `bool`
- En-tête `<complex.h>` et types `double complex`, `float complex`, `long double complex`
- `int tab[n]` avec `n` pas forcément une constante ok ; mais le tableau ne change pas de taille après allocation !
- En-tête `<threads.h>`, manière standard de définir de la programmation multi-thread ; peut remplacer la bibliothèque POSIX `<pthread.h>`

Et C++ ?

- C et C++ sont deux langages de programmation différents
- C++ peut être vu de manière approximative comme étendant le langage C, mais il n'est pas vrai qu'un programme C est toujours un programme C++ correct
- C++ est un langage très complexe (POO, programmation générique, métaprogrammation, surcharge d'opérateurs, éléments de programmation fonctionnelles, riche bibliothèque standard)
- Souvent possible et désirable de faire du C++ lite, proche du C mais en bénéficiant des conteneurs standard du C++, des chaînes de caractère C++, d'améliorations de syntaxe, etc.
- Relativement facile d'avoir un logiciel qui mélange du code C et du code C++ dans des unités de compilation séparées (ou d'autres langages compilés, comme Rust) : compiler séparément (en faisant attention à viser l'ABI C, p. ex.., via des déclarations `extern "C"`)

Les bonnes options de compilation de GCC

`-Wall -Wextra` pour les messages d'avertissement, permettant de détecter des bugs évidents (variables non utilisées, retour de fonction manquant, = utilisé dans un if(), etc.)

`-pedantic` pour encore plus de messages d'avertissement

`-g` pour inclure les symboles dans l'exécutable généré et permettre le débogage

`-std=c11` pour vérifier autant que possible la conformité au standard C 2011 (ou `-ansi`, `-std=c99` pour versions plus anciennes)

`-pg` pour faire du profilage (voir plus tard)

`-O2` bon niveau d'optimisation, mais à ne pas utiliser quand on veut déboguer : fait disparaître certaines variables ou instructions, etc.

Clang

- `gcc` (du projet GNU) est le compilateur par défaut sous Linux
- `clang` est un compilateur alternatif libre, plus récent, par certains aspects mieux conçu :
  - modulaire, séparation entre le *front end* Clang et le *back end* LLVM qui s'occupe de la génération de code
  - interfaçage plus facile avec outils tiers (p. ex., IDE)
  - souvent, meilleurs messages d'erreur
  - souvent plus rapide
- Par compatibilité, `clang` s'utilise exactement comme `gcc` (mêmes options), donc facile de passer de l'un à l'autre (changer une variable de `Makefile`)

Plan

Deux types de bibliothèques

- Fichiers `.a` : bibliothèques statiques
  - archives de code qui peuvent être intégrés à un programme pendant l'édition de lien
  - regroupent au sein d'un même fichier plusieurs fichiers compilés
  - créées avec la commande `ar`
- Fichier `.so` : bibliothèques dynamiques
  - collections de fonctions qui peuvent être chargées dynamiquement lorsqu'un programme utilise ces fonctionnalités
  - utilisation partagée entre plusieurs processus
  - peuvent servir à implémenter un mécanisme de plug-in avec chargement dynamique après l'exécution
  - créées en utilisant `-shared -fPIC` à l'édition de liens ; les fichiers `.o` doivent avoir été compilés avec `-fPIC`

Édition de liens sous Linux

- L'éditeur de liens par défaut est `ld` ; il est appelé par le compilateur au moment de l'édition de liens (avec éventuellement arguments supplémentaires)
- On indique à l'édition de liens, en plus des fichiers `.o`, des bibliothèques dont on dépend avec `-ltoto` : si l'éditeur de liens peut trouver un `libtoto.so` (de préférence) ou un `libtoto.a` (sinon, ou si `-static` est spécifié), il l'utilise
- L'éditeur de liens cherche dans les répertoires indiqués dans `/etc/ld.so.conf`, dans ceux de la variable d'environnement `LIBRARY_PATH`, ainsi que dans ceux fournis en ligne de commande avec `-L`
- Les fonctions des bibliothèques statiques sont ajoutées au programme ; des références aux bibliothèques dynamiques sont ajoutées à l'exécutable (en fixant une version majeure)
- Seules les bibliothèques utiles sont utilisées !

Bibliothèques dynamiques à l'exécution

- À l'exécution d'un programme, si celui-ci a des dépendances à des bibliothèques partagées, une de ces dépendances est le programme `ld.so`
- Ce programme charge les bibliothèques indiquées dans `LD_PRELOAD`
- Ce programme est utilisé pour trouver les dépendances indiquées dans l'exécutable dans les répertoires :
  - indiqués dans la variable d'environnement `LD_LIBRARY_PATH`
  - indiqués dans le `RUNPATH` fourni à l'édition de liens avec `-rpath` (ou `-Wl,rpath` via `gcc`)
  - indiqués dans `/etc/ld.so.conf` ; voir `ldconfig -p`

Inspecter les dépendances d'un programme

ldd donne la liste des bibliothèques dynamiques référencées par un programme

nm -D donne la liste des symboles définis par une bibliothèque dynamique ou un exécutable, ou référencés par un exécutable

readelf plus bas niveau, inspecte un fichier ELF

c++filt est un filtre permettant de déchiffrer les noms de symboles C++ encodés, quand certains codes ou bibliothèques sont écrites en C++

Plan

Traçage

- On peut toujours mettre des `printf(...)` ou `fprintf(stderr, ...)`
- Utiliser grep pour rapidement repérer l'information utile
- Penser à utiliser intelligemment les deux sorties du programme (stdout, stderr) pour pouvoir rediriger les flux d'information

Intégrer un mode debug dans le Makefile

- Pour utiliser un débogueur, nécessite d'ajouter les symboles (noms de variables, etc.) à l'exécutable, et de désactiver les optimisations
- Courant de devoir compiler alternativement en mode production (optimisé) et en mode debug (pour déboguer)
- Le plus pratique : un flag dans le Makefile
- S'utilise avec `make DEBUG=1`

```
ifndef DEBUG
  CFLAGS += -O2
else
  CFLAGS += -Og -g
endif
```

GDB : utilisation

- On lance GDB avec `gdb ./mon_programme`
- Principales commandes :

r < input	pour démarrer le programme en lisant `input` sur sa sortie standard
r	redémarre le programme
b nom_fonction	pour positionner un point d'arrêt
clear	supprime le point d'arrêt en cours
c	continue jusqu'au prochain point d'arrêt
n, s	avance d'une instruction ; s va à l'intérieur des appels de fonction
bt, u, d	affiche et navigue dans la pile des appels
print expr	affiche la valeur d'une expression
watch expr	arrêtera le programme quand la valeur de l'expression changera
q	quitte GDB

Interfaces graphiques de GDB

- `gdb -tui` : pour une simili-interface graphique, assez pratique
- `ddd, KDbg, Insight` : interfaces graphiques plus ou moins complètes
- Intégration à certains IDE, dont VSCode

Débogage mémoire

Programmes identifiant les fuites mémoires, débordement de pile, accès à de la mémoire non allouée sur le tas. . . :

valgrind est une machine virtuelle exécutant le programme en contrôlant chacun des accès mémoire ; très lent, mais très efficace

```
valgrind ./mon_programme
```

Permet souvent de trouver rapidement la source d'un problème, à essayer tôt.

efence redéfinit les fonctions d'allocation mémoire pour détecter les accès incorrects au tas

```
LD_PRELOAD=libefence.so ./mon_programme
```

Permettent en général d'identifier les bugs au moment où ils apparaissent (mais uniquement pour la mémoire dynamique), contrairement à `gdb` en cas de corruption mémoire

Appels systèmes et appels dynamiques

strace Liste les appels systèmes utilisés par un programme donné ; utile pour voir à quels fichiers on accède ou quelles connexions réseau sont établies

ltrace liste les appels de fonctions depuis des bibliothèques dynamiques (y compris des fonctions de la bibliothèque C) ; attention, seules les fonctions non présentes à l'intérieur de l'exécutable seront indiquées ; en C++, ceci exclut en particulier la plupart des classes avec paramètres de modèle (templates)

Déboguer un programme en cours

- Suivant la configuration du système, il peut être possible d'utiliser `gdb`, `strace`, `ltrace`, sur un programme en cours
- Simplement ajouter l'option « -p *pid* » en ligne de commande, où *pid* est le PID
- Souvent réservé au superutilisateur sur les Linux modernes, voir le paramètre `kernel.yarma.ptrace_scope` du noyau

Profilage avec gnuprof

- Permet de savoir dans quelles fonctions le programme passe son temps
- Compiler avec `-pg`
- Exécuter et attendre la fin normale du programme
- Un fichier `gmon.out` est créé
- Exécuter `gprof ./mon_executable > rapport.txt` pour analyser ce fichier
- Étudier `rapport.txt`
- Aussi simplement utiliser `time ./mon_programme` pour mesurer le temps pris par un programme

Plan

Cycle de développement logiciel

- Développement (collaboratif, avec système de contrôle de version)
- Documentation du code
- Vérification de la qualité du code (norme de codes, tests supplémentaires par rapport aux compilateurs, analyse statique de code) – on parle parfois de linter
- Compilation
- Tests unitaires
- Déploiement sur serveur de test (si application nécessitant un serveur)
- Tests fonctionnels
- Documentation logicielle
- Tests utilisateurs
- Déploiement (sous forme de paquet source, de binaires, de machine virtuelle, d'application sur un serveur de production)

Tests unitaires et fonctionnels

- Test unitaire : test minimal permettant de tester chaque fonction, chaque fonctionnalité élémentaire, en incluant les cas extrêmes
- Test fonctionnel : test en condition réelle de toute l'application, d'un workflow
- De nombreux frameworks de test, cf. `https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`
- Développement mené par le test : on écrit les jeux de tests avant d'écrire le code !

Intégration continue & déploiement continu

Intégration continue Système permettant de vérifier tout au long du processus de développement (par exemple, à chaque nouveau code intégré) :

- que le code compile
- que les linters sont contents
- que les tests passent

Livraison continue Système permettant automatiquement, par exemple à chaque nouvelle version, de déployer la nouvelle version en production, de produire et publier des packages sources et binaires, des machines virtuelles, etc.

Beaucoup de systèmes commerciaux (nécessité de serveurs pour faire tourner l'intégration) ; Jenkins et Gitlab sont deux systèmes libres, avec des options commerciales.

Automatisation de la configuration des sources

- Pour les logiciels distribués sous forme de code source, nécessité de fournir des `Makefile` (ou autre système de compilation) permettant de compiler le code dans les meilleures conditions
- Dépend du système d'exploitation de l'utilisateur, de la version de son compilateur, de l'endroit où il veut installer le logiciel, etc.
- Quand c'est possible : faire au plus simple, Makefile et documentation de comment l'adapter a minima
- Quand nécessaire : utiliser un système d'automatisation de la configuration des sources, comme GNU autotools ou CMake

Installation de code source configuré avec autotools

- Télécharger et décompresser l'archive (avec `tar` ou `unzip`)
- Lire le `README`, fichier `INSTALL` ou similaire
- Lancer `./configure` avec les options appropriées telles :

  `--prefix=`*p* pour indiquer que le logiciel sera installé dans le répertoire *p* (par exemple, `/opt/nom_logiciel` ou `$HOME/opt/nom_logiciel`)

  `--enable-`*f* pour activer une fonctionnalité
- Lancer `make` (gagner du temps avec `-j`)
- Lancer `make test` suivant les cas, pour vérifier que le logiciel fonctionne
- Lancer `make install`