POLYTECH PARIS-SUD

Et3 - Tronc Commun

Programmation C

TP 1 : Informations Générales

1 Votre compte utilisateur

Lors de la première séance de TP, vous allez découvrir votre environnement de travail. Vous devez tout d'abord vous connecter en utilisant le login et le mot de passe par défaut qui vous ont été fournis. Ensuite, vous devrez modifier votre mot de passe. N'oubliez pas que :

1. Votre mot de passe est strictement personnel.
2. Vous ne devez jamais le communiquer (ni à l'équipe enseignante, ni à l'administration, ni à vos proches).
3. Si vous perdez votre mot de passe, prenez contact avec l'équipe système qui vous redonnera un nouveau mot de passe.
4. Enfin, les environnements Linux et Windows sont déconnectés... vous devez donc changer vos mots de passe sous les deux environnements pour éviter que votre mot de passe par défaut reste actif sur l'un des deux environnements.

La procédure de changement de mot de passe (sous Linux et Windows) est automatique lors de la première connection. Faites bien attention aux touches que vous utilisez, vérifiez que le clavier n'est pas vérouillé (touche majuscule/minuscule).

Pour activer votre compte mail, connectez-vous sur la page web `http://www.annuaire.u-psud.fr/` puis choisissez l'onglet « Activation étudiant » dans la rubrique « Services Liés ».

Lorsque ces deux opérations sont effectuées, vous pourrez débuter votre TP.

2 Utiliser Linux

Voici une liste de commandes vous permettant d'exploiter au mieux votre espace de travail via la console.

Commande	**Action**
`mkdir DOSSIER`	Création d'un nouveau répertoire nommé DOSSIER
`cd DOSSIER`	Entrer dans le répertoire DOSSIER
`cd ..`	Remonter dans le répertoire parent
`pwd`	Afficher à quel niveau de l'arborescence vous vous situez
`ls`	Lister le contenu du répertoire courant
`cp FICHIER_SOURCE DOSSIER`	Copier FICHIER_SOURCE dans DIRECTORY
`cp FICHIER_SOURCE DESTINATION`	Copier FICHIER_SOURCE dans le fichier/dossier DESTINATION
`mv SOURCE DESTINATION`	Déplacer/renommer un fichier SOURCE dans une DESTINATION
`rm FICHIER`	Supprimer le FICHIER

FIGURE 1 – Récapitulatif des commandes du terminal

Pour les chemins vers les dossiers ou les fichiers, vous pouvez écrire des chemins absolus (à partir de la racine /, p. ex. /Users/nomUtilisateur/Documents/C/monProg) ou relatifs (à partir de l'endroit actuel). Le dossier actuel est désigné par le point « . », le dossier parent par « .. ». Votre dossier 'home' peut être nommé par le raccourci '∼' (ex : `cd ∼`).

Nous utiliserons systématiquement la console pour compiler (transformer le texte du programme en code exécutable) et exécuter le programme obtenu.

La première opération à réaliser est de structurer votre espace de travail pour pouvoir y organiser vos données. Vous allez pour cela créer un répertoire nommé `tps_c` en utilisant la commande suivante : `mkdir tps_c`, puis changer de répertoire pour aller dans le répertoire `tps_c` : `cd tps_c` , et enfin y créer un nouveau répertoire `tp1`.

Vous venez de structurer votre espace de travail de manière cohérente. Cela vous permettra de retrouver facilement vos TP par la suite. Dans la mesure du possible, essayez systématiquement de structurer vos données de la manière la plus cohérente possible.

3 Éditeur de texte, compilation et débogage

3.1 Éditeur de texte

Afin de pouvoir écrire et tester votre premier programme, vous devrez commencer par choisir un éditeur de texte pour saisir votre programme. La plupart de ces éditeurs proposent la coloration syntaxique, ainsi que l'indentation automatique du code. L'important est de choisir un éditeur qui vous convienne et que vous arrivez à bien utiliser (raccourcis clavier, fonctionnalités diverses, ...).

On s'interdira dans ce module l'utilisation des IDE, afin d'apprendre à compiler correctement. Vous utiliserez l'éditeur de texte de votre choix parmi les suivants : gedit, kedit, geany, xemacs, emacs, nano, vim

3.2 Compilation

Après avoir choisi un éditeur, vous pourrez écrire votre programme. Ensuite, dans une console, placez-vous dans votre répertoire de travail et exécutez la commande permettant de compiler le programme pour obtenir une version exécutable :

```
gcc -g -Wall -std=c99 -o programme programme.c
```

Ce qui suit après `-o` est le nom de l'exécutable généré.

Si la compilation s'est bien déroulée (il n'y a pas de message d'erreur dans la console), vous avez obtenu un programme exécutable qui se trouve dans votre répertoire courant. Vous pouvez simplement l'exécuter en utilisant la commande : `./programme`

Les deux options de compilation suivantes vous permettent de détecter des erreurs dans votre code :

- `-Wall` : pour activer tous les *warnings*. Dans la mesure du possible, il faut corriger le code afin de ne plus avoir de *warnings*. En cas de doute, demandez l'aide de votre enseignant.
- `-ansi` : pour activer la vérification de la norme ANSI 2. Ceci vous permettra de garantir que votre code respecte bien la norme ANSI et vous garantira ainsi la portabilité de votre programme sur d'autres environnements respectant également cette norme (Windows, Mac, Unix, ...).

3.3 Débogage

Lorsque votre programme ne se comporte pas correctement, il faut trouver la source de vos erreurs. La façon la plus productive est de passer par un débogueur qui va rejouer pas à pas votre code. Pour ce faire, il faut compiler votre programme avec l'option `-g` qui va demander au compilateur d'insérer des informations de traçage à l'intérieur de votre programme.

L'outil `gdb` peut ensuite être utilisé pour lancer votre programme : `gdb mon_programme`. Une invite de commande se met à votre disposition.

Quelques options :

`run` ou `r`	Exécuter le programme sous GDB
`continue` ou `c`	Continuer l'exécution du programme sous GDB (lorsque le programme est suspendu. Voir ci-dessous « breakpoints » et « watchpoints »).
`print nomVariable`	Inspecter le contenu d'une variable
`step` ou `s`	Avancer d'un pas dans votre code. Cette commande 'descendra' aussi dans les fonctions
`next` ou `n`	Avancer d'un pas dans votre code. Contrairement à `step`, cette commande ne 'descendra' pas dans les fonctions
`help` ou `h`	Afficher l'aide
`quit` ou `q`	Sortir de GDB

Points d'arrêts « breakpoints »

Pour mieux étudier le déroulement du programme à différentes étapes de l'exécution, on peut placer des points d'arrêts qui vont arrêter le programme à l'endroit désigné, pendant une exécution lancée par la

commande `run`.

Insérer un « breakpoint » :
- `break numéroLigne` pour le placer à ce numéro de ligne
- `break nomFonction` pour le placer au début de cette fonction
- `break nomFichier:numéroLigne` pour le placer à ce numéro de ligne dans ce fichier

Lister les « breakpoints » actuels : `info break`
Supprimer un « breakpoint » : `delete numéroBreakpoint`

Vouez pouvez insérer un « breakpoint » *conditionnel* comme suit : `break numéroLigne if expr`, où `expr` est une expression booléenne (valant vrai ou faux)

Ex : `break numéroLigne if nomVar == 100`

Points d'arrêts « watchpoints »

Les « watchpoints » permettent de surveiller non un numéro de ligne mais une variable ou une expression. Poser un « watchpoint » sur une variable signifie que le programme se mettra en pause lorsque la valeur de la variable change.

Insérer un « watchpoint » : `watch nomVariable`

Vous pouvez alors posez des points d'arrêt via la commande `break`. Un point d'arrêt va forcer l'exécution du programme à se mettre en pause. Vous pourrez alors :

De même que pour un « breakpoint », la commande `continue` ou `c` permet de reprendre l'exécution après que GDB s'est arrêté à un watchpoint.

4 Formatage d'affichage

Lorsqu'on affiche les valeurs des variables au terminal (p. ex. avec `printf`), vous pouvez les formater de différentes manières.

Les types

%d ou %i	int	entier relatif
%u	unsigned int	entier naturel
%c	char	caractère
%f	float/double	flottant
%s	char*	chaîne de caractères

Ex. : `printf("Voici un int %d et un float: %f", monInt, monFloat);`

Largeur

- nombre : nombre minimum de caractères à afficher (caractères d'espacement ajoutés si la valeur est plus courte que l'affichage demandé)
- 0nombre : comme ci-dessus, sauf que les caractères 0 sont ajoutés si la valeur est plus courte que l'affichage demandé

Ex. : `printf("Voici un int %03d", 3` donnera « Voici un int 003 »

Précision

- i, d, u : nombre minimum de chiffres à afficher, ex : `printf("%.2d", 3);` donnera « 03 »
- f : nombre de chiffres à afficher après le point décimal, ex : `printf("%.2f", 3.576);` donnera « 3.58 »
- s : nombre maximum de caractères a afficher, ex : `printf("%.2s", "hello");` donnera « he »

Quelques caractères spéciaux

— \n : saut de ligne
— \t : tabulation
— \" : affichage du caractère '"'

Pour plus d'options, découvrez l'entrée `printf` dans le manuel en tapant dans un terminal :

```
man printf
```

5 Qualité du code

Une partie non négligeable de votre évaluation se fera sur la lisibilité de votre code. Un code illisible est un code inutile. Voila quelques éléments de style à respecter :

— Donnez des noms explicites à vos variables et à vos fonctions. Par exemple, préférez `int nb_pixels_noirs = compter_pixel(image, 0)` à `int n = cpxl(i, 0)`.
(Cf. Partie 5)
— Commentez les actions non triviales de votre code afin que votre relecteur n'ait pas à réfléchir trop longtemps pour comprendre votre raisonnement. En C, tout ce que vous écrivez entre les symboles `/*` et `*/` est un commentaire.
— Aérez votre code en sautant des lignes et en utilisant des espaces.
— Utilisez l'indentation pour le contenu des boucles, des conditions et des structures pour rendre votre code plus lisible.
— Décomposez votre problème en fonctions spécialisées orchestrées par la fonction `main()`
— Ne faites pas de fonctions de plus d'un écran de long

6 Comment nommer vos variables

Les noms des variables et des fonctions peuvent contenir uniquement des caractères non accentués (miniscules ou majuscules), des chiffres et le tiret du 8 « _ ». Les noms ne doivent pas commencer par un chiffre et ne doivent pas être un des mots clés du langage (p. ex. `while`, `for`, `if`, `int`, `float` etc.).

En général, les noms des variables i, j, k sont utilisés pour les noms des variables dans les boucles, qui ne servent que comme indices à incrementer itérativement.

Pour les variables qui seront utilisées à plus grande portée, pensez à donner des noms clairs et explicatifs.

7 Votre moteur de recherche préféré est votre meilleur ami...

Vraiment.

Running

`# gdb <program> [core dump]`
Start GDB (with optional core dump).

`# gdb --args <program> <args…>`
Start GDB and pass arguments

`# gdb --pid <pid>`
Start GDB and attach to process.

`set args <args...>`
Set arguments to pass to program to be debugged.

`run`
Run the program to be debugged.

`kill`
Kill the running program.

Breakpoints

`break <where>`
Set a new breakpoint.

`delete <breakpoint#>`
Remove a breakpoint.

`clear`
Delete all breakpoints.

`enable <breakpoint#>`
Enable a disabled breakpoint.

`disable <breakpoint#>`
Disable a breakpoint.

Watchpoints

`watch <where>`
Set a new watchpoint.

`delete/enable/disable <watchpoint#>`
Like breakpoints.

<where>

`function_name`
Break/watch the named function.

`line_number`
Break/watch the line number in the current source file.

`file:line_number`
Break/watch the line number in the named source file.

Conditions

`break/watch <where> if <condition>`
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

`condition <breakpoint#> <condition>`
Set/change the condition of an existing break- or watchpoint.

Examining the stack

`backtrace`
`where`
Show call stack.

`backtrace full`
`where full`
Show call stack, also print the local variables in each frame.

`frame <frame#>`
Select the stack frame to operate on.

Stepping

`step`
Go to next instruction (source line), diving into function.

`next`
Go to next instruction (source line) but don‘t dive into functions.

`finish`
Continue until the current function returns.

`continue`
Continue normal execution.

Variables and memory

`print/format <what>`
Print content of variable/memory location/register.

`display/format <what>`
Like „print“, but print the information after each stepping instruction.

`undisplay <display#>`
Remove the „display“ with the given number.

`enable display <display#>`
`disable display <display#>`
En- or disable the „display“ with the given number.

`x/nfu <address>`
Print memory.
n: How many units to print (default 1).
f*:* Format character (like „print“).
u: Unit.

Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

Format

a Pointer.
c Read as integer, print as character.
d Integer, signed decimal.
f Floating point number.
o Integer, print as octal.
s Try to treat as C string.
t Integer, print as binary (*t* = „two“).
u Integer, unsigned decimal.
x Integer, print as hexadecimal.

<what>

expression
Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).

file_name::*variable_name*
Content of the variable defined in the named file (static variables).

function::*variable_name*
Content of the variable defined in the named function (if on the stack).

{type}address
Content at *address*, interpreted as being of the C type *type*.

$*register*
Content of named register. Interesting registers are $esp (stack pointer), $ebp (frame pointer) and $eip (instruction pointer).

Threads

`thread <thread#>`
Chose thread to operate on.

Manipulating the program

`set var <variable_name>=<value>`
Change the content of a variable to the given value.

`return <expression>`
Force the current function to return immediately, passing the given value.

Sources

`directory <directory>`
Add *directory* to the list of directories that is searched for sources.

`list`
`list <filename>:<function>`
`list <filename>:<line_number>`
`list <first>,<last>`
Shows the current or given source context. The *filename* may be omitted. If *last* is omitted the context starting at *start* is printed instead of centered around it.

`set listsize <count>`
Set how many lines to show in „list“.

Signals

`handle <signal> <options>`
Set how to handle signles. Options are:

(no)print: (Don‘t) print a message when signals occurs.

(no)stop: (Don‘t) stop the program when signals occurs.

(no)pass: (Don‘t) pass the signal to the program.

Informations

`disassemble`
`disassemble <where>`
Disassemble the current function or given location.

`info args`
Print the arguments to the function of the current stack frame.

`info breakpoints`
Print informations about the break- and watchpoints.

`info display`
Print informations about the „displays“.

`info locals`
Print the local variables in the currently selected stack frame.

`info sharedlibrary`
List loaded shared libraries.

`info signals`
List all signals and how they are currently handled.

`info threads`
List all threads.

`show directories`
Print all directories in which GDB searches for source files.

`show listsize`
Print how many are shown in the „list“ command.

`whatis variable_name`
Print type of named variable.