Ecole Nationale d'Ingénieurs de Brest

Le langage C++

— Travaux Pratiques —

J. Tisseau , S. Morvan

Th. Duval , F. Harrouet
A. Nedelec , R. Oussin
P. Reignier , V. Rodin

– 1994,1997 –

Avant–propos

Ces Travaux Pratiques accompagnent le cours de **Programmation par objets** dispensé à l'Ecole Nationale d'Ingénieurs de Brest (ENIB). Ce cours fait suite à un cours d'**Algorithmique** illustré à l'aide du langage C+ (C++ sans les classes).

Tisseau J., Morvan S. (1996) *Programmation par objets : le langage C++*
Notes de Cours, ENIB — 1996

Tisseau J. (1994) *Programmation par objets : les concepts*
Notes de Cours, ENIB — 1994

Les Travaux Pratiques sont organisés par séances de 3 h.
La réalisation de ces TP et la lecture des notes de cours correspondantes complètent la consultation d'ouvrages plus détaillés tels que ceux référencés ci–dessous.

Carroll M.D., Ellis M.A. (1995) *Designing and coding reusable C++.*
Addison–Wesley — 1995

Coplien J.O. (1992) *Programmation avancée en C++ : styles et idiomes.*
Addison–Wesley — 1992

Ellis M.J., Stroustrup B. (1990) *The annoted C++ Reference Manual.*
Addison–Wesley — 1990

Lippman S.B. (1992) *L'essentiel du C++.*
Addison–Wesley — 1992

Lippman S.B. (1997) *Le modèle objets du C++.*
International Thomson Publishing — 1997

Murray R.B. (1994) *Stratégies et tactiques en C++.*
Addison–Wesley — 1994

Musser D.R., Saini A. (1996) *STL tutorial and reference guide.*
Addison–Wesley — 1996

Stroustrup B. (1992) *Le langage C++.*
Addison–Wesley — 1992

T.P. 1

Rappels UNIX

1.1 Le système UNIX de travail

1.1.1 Organisation des TP

Les séances de Travaux Pratiques se déroulent par groupes de 24 étudiants d'une même promotion. Les étudiants travaillent en binômes.

1. Identifier la promotion. **Promotion :**
2. Identifier le groupe de TD. **Groupe :**
3. Choisir un numéro de binôme. **Binome :**

1.1.2 Connexion/Déconnexion

1. Quel est le code d'accès du binôme ? **Code :**
2. Quel est le mot de passe actuel correspondant à ce code d'accès ?
3. Répéter deux fois de suite les opérations de connexion et de déconnexion.
4. Changer de mot de passe, puis répéter deux fois de suite les opérations de connexion et de déconnexion.

1.1.3 Environnement de travail

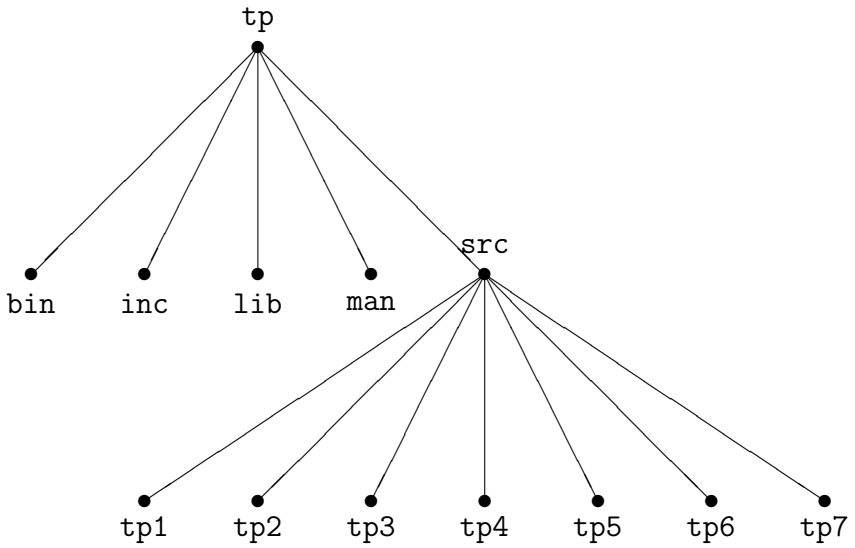
1. Identifier le répertoire de connexion du binôme.
2. Identifier l'interpréteur UNIX courant (le *"shell"* courant).

3. Identifier le groupe UNIX auquel appartient le binôme.

1.2 Organisation des répertoires

1.2.1 Création de l'arborescence de travail

Les répertoires seront organisés selon l'arborescence ci–dessous.



`tp`	répertoire principal pour toutes les séances de T.P.
`bin`	répertoire des fichiers exécutables
`inc`	répertoire des fichiers en–tête (`*.h`)
`lib`	répertoire des bibliothèques de fonctions et de classes (`lib*.a`)
`man`	répertoire des manuels
`src`	répertoire principal pour tous les fichiers sources
`src/tp1`	répertoire des fichiers sources (`*.C`) du T.P. n°1
`src/tp2`	répertoire des fichiers sources (`*.C`) du T.P. n°2
`...`	...

1. Vérifier que le répertoire de connexion ne contient pas de répertoire `tp`. Si ce n'est pas le cas, effacer ce répertoire et son contenu.

2. Créer l'arborescence de travail décrite ci–dessus.

1.2.2 La variable `PATH`

1. Afficher le contenu de la variable d'environnement `PATH`.

2. Ajouter le répertoire `$HOME/tp/bin` aux répertoires de recherche des exécutables (variable `PATH` dans le fichier `.profile`).

1.3 Corrigés des TP

1.3.1 Recherche d'un corrigé

Au début de chaque nouveau TP, un corrigé du TP précédent est disponible pour chaque groupe dans un répertoire particulier.

1. Vérifier l'existence d'un répertoire `corriges` et noter son chemin dans la variable d'environnement `CORRIGES`.

   ```
   CORRIGES :
   ```

2. Copier tous les fichiers d'extension `.C` du répertoire `corriges/tp1` dans le répertoire du binôme `tp/src/tp1` .

3. Copier tous les fichiers d'extension `.h` du répertoire `corriges/tp1` dans le répertoire du binôme `tp/include` .

1.3.2 Script de recherche

1. Ecrire un script UNIX (fichier `corriges`) qui généralise la commande précédente en copiant tous les fichiers d'un sous–répertoire du répertoire `corriges` dans le sous–répertoire correspondant du répertoire `tp/src` pour les fichiers d'extension `.C` et dans le répertoire `tp/include` pour les fichiers d'extension `.h` (fonctionnement par défaut), ou pour les fichiers d'extension `.C`, dans un sous–répertoire de `tp/src` précisé dans la ligne de commande.

2. Vérifier le bon fonctionnement du script `corriges` à l'aide des exemples suivants.

   (a) La commande `corriges` affiche le message d'erreur suivant :
   `usage : corriges tpsource [tpdestination]`

   (b) La commande `corriges tp1` copie tous les fichiers du répertoire `corriges/tp1` dans le répertoire `src/tp1`.

   (c) La commande `corriges tp1 tp2` copie tous les fichiers du répertoire `corriges/tp1` dans le répertoire `src/tp2`.

   (d) La commande `corriges tpbidon` affiche le message d'erreur suivant :
   `corriges : tpbidon repertoire source inconnu`

   (e) La commande `corriges tp1 tpbidon` affiche le message d'erreur suivant :
   `corriges : tpbidon repertoire destination inconnu`

T.P. 2

La compilation en C++

2.1 Le compilateur C++

2.1.1 Définition d'un compilateur

1. Qu'est–ce qu'un compilateur ?

2. Identifier les principales phases de la compilation.

3. Citer des exemples de compilateurs.

2.1.2 La commande UNIX

1. Identifier la commande correspondant à l'appel du compilateur C (dans la suite, cette commande sera notée `<cc>`).

   Quelle doit être l'extension des fichiers sources en langage C ?

2. Identifier la commande correspondant à l'appel du compilateur C++ (dans la suite, cette commande sera notée `<cc++>`).

   Quelle doit être l'extension des fichiers sources en langage C++ ?

2.1.3 Les principales options du compilateur C++

On consultera le manuel de la commande `<cc++>` pour trouver la signification des options suivantes.

1. `<cc++> -P -I<...> -D<...>`

2. `<cc++> -c -g -O`

3. `<cc++> -L<...> -l<...> -o<...>`

2.2 La chaîne de traitement

On considère le programme ci–dessous (fichier `tp2.2.C`).

```
// tp2.2.C

#include <iostream.h>
#define MAX 10

int main(void) {
      cout << "Mon premier programme affiche " << MAX << endl;
      cout << MAX << endl;

      return 0;
}
```

2.2.1 L'édition du code source

1. Identifier les éditeurs de textes du système.
2. Choisir un éditeur pour éditer le fichier `tp2.2.C` dans le répertoire `tp/src/tp2` .

2.2.2 Le préprocesseur

1. Rappeler le rôle des directives suivantes :
   (a) `#define`
   (b) `#include`
2. Obtenir le fichier `tp2.2.i` correspondant au fichier `tp2.2.C` juste après l'action du préprocesseur.
3. Expliquer pourquoi le préprocesseur lui–même peut être considéré comme un compilateur.

2.2.3 La compilation

1. Obtenir le fichier `tp2.2.o` correspondant au fichier `tp2.2.i` juste après la phase de compilation.
2. Expliquer pourquoi le fichier `tp2.2.o` n'est pas exécutable.

2.2.4 L'édition de liens

1. Quel est le rôle de l'édition de liens ?
2. Obtenir le fichier `tp2.2` correspondant au fichier `tp2.2.o` après l'action de l'éditeur de liens.

2.2.5 L'exécution du programme

1. Comparer les tailles des fichiers `tp2.2.C`, `tp2.2.i`, `tp2.2.o` et `tp2.2` .
2. Lancer l'exécution du programme `tp2.2` .

2.3 Les erreurs de compilation

On considère le programme ci–dessous (fichier `tp2.3.C`) qui contient volontairement un certain nombre d'erreurs.

```
// tp2.3.C                                          // ce fichier contient des erreurs !

#include <iostream.h>
#define min(x,y)
      ( ((x) < (y)) ?  (x) :  (y) ;)

double sin(double);

int main(void) {
      double x = sin(2.5);
      double y = sin(3.8);

      cout << "x = " << x << endl;
      cout << "y = " << y << endl;
      cout << min(x,y) << endl;
}
```

2.3.1 Erreurs liées au préprocesseur

1. Identifier, dans le fichier `tp2.3.C`, les erreurs liées au préprocesseur.
2. Quels sont les avantages d'une fonction sur une macro–instruction ?
3. Quels sont les "désavantages" d'une fonction sur une macro–instruction ?

2.3.2 Erreurs liées au compilateur

1. Identifier, dans le fichier `tp2.3.C`, les erreurs liées au compilateur.
2. Identifier les différents niveaux de messages d'erreur du compilateur.

2.3.3 Erreurs liées à l'éditeur de liens

1. Identifier, dans le fichier `tp2.3.C`, les erreurs liées à l'éditeur de liens.
2. Identifier les librairies standards chargées par l'éditeur de liens.

T.P. 3

Compilation conditionnelle, compilation séparée

On considère les fichiers `fonction.h` et `fonction.C` donnés respectivement en sections 3.3.1 page 12 et 3.3.2 page 13.

3.1 La compilation conditionnelle

3.1.1 Manipulation de fonctions

1. Expliquer le rôle des directives suivantes (voir `fonction.h`).

   ```
   #ifndef FONCTION_H
   #define FONCTION_H
   ...
   #endif
   ```

2. Que représente le champ `double (*fct)(double)` de la structure `Fonction` (voir `fonction.h`) ?

3. Quel est le rôle de la fonction `double (*trouverFonction(char* nom, Fonction* tab))(double)` (voir `fonction.C`) ?

4. Ecrire un programme (`tp3.1.C`) qui calcule la valeur d'une fonction d'un nombre donné, la fonction et le nombre étant passés sur la ligne de commande. A ce stade, on incluera directement `fonction.C` dans `tp3.1.C` (`#include "fonction.C"`).

   ```
   $ tp3.1 sin 1.57
   f = sin
   x = 1.57
   f(x) = sin(1.57) = 1
   $ tp3.1 ceil 1.57
   f = ceil
   x = 1.57
   f(x) = ceil(1.57) = 2
   ```

3.1.2 Test d'un fichier

1. Incorporer la fonction `main` du fichier `tp3.1.C` dans le fichier `fonction.C` de telle manière qu'elle ne soit compilée que si la macro `TESTFONCTION` est définie.

2. Comparer les résultats obtenus par les 2 commandes suivantes :

   (a) `<cc++> -I../../include -DTESTFONCTION fonction.C`

   (b) `<cc++> -I../../include fonction.C`

3.2 La compilation séparée

3.2.1 Edition de liens

1. Reprendre le fichier `tp3.1.C` en remplaçant la directive `#include "fonction.C"` par `#include "fonction.h"` .

   Quelle est la principale conséquence de ce changement de directive ?

2. Compiler séparemment `tp3.1.C` et `fonction.C`, puis lier les fichiers ainsi compilés pour obtenir un exécutable.

3.2.2 Utilisation d'une librairie

1. Créer la librairie `tp/lib/libtp3.a` contenant le fichier `fonction.o` créé alors que la macro `TESTFONCTION` n'était pas définie (pas de fonction `main` dans `fonction.o`).

2. Recompiler `tp3.1.C` en utilisant cette librairie à l'édition de liens.

3.3 Fichiers de travail

3.3.1 Fichier `fonction.h`

```
// fonction.h

#ifndef FONCTION_H
#define FONCTION_H

struct Fonction {
      char* nom;
      double (*fct)(double);
};

double (*trouverFonction(char* nom, Fonction* tab))(double);

extern Fonction fonctions[];              // tableau defini dans fonction.C

#endif
```

3.3.2 Fichier `fonction.C`

```
// fonction.C

#include <math.h>
#include <string.h>
#include "fonction.h"

Fonction fonctions[] = {
      "acos", acos, "asin", asin, "atan", atan, "ceil", ceil,
      "cos", cos, "cosh", cosh, "exp", exp, "fabs", fabs,
      "floor", floor, "log", log, "sin", sin, "sinh", sinh,
      "sqrt", sqrt, "tan", tan, "tanh", tanh, 0, 0 };

double (*trouverFonction(char* nom, Fonction* tab))(double) {
      for(Fonction* f = tab; f->nom && strcmp(nom,f->nom); f++) ;
      return f->fct;
}
```

T.P. 4

La commande make

4.1 Le manuel UNIX

4.1.1 $ man make

Consulter le manuel en ligne de la commande `make` .

1. Décrire la structure d'une règle `make` .
2. Expliciter les règles suivantes :

```
fonction: fonction.o
        xlC fonction.o -o fonction

fonction.o: fonction.h fonction.C
        xlC -c -DTESTFONCTION -I../../include fonction.C
```

3. Modifier les règles précédentes en tenant compte des macros suivantes :

```
CCFLAGS = -c -DTESTFONCTION
INCDIR  = ../../include
```

4.1.2 *"The* `make` *command"*

Consulter les extraits du manuel `UNIX` de la commande `make` reproduits en section 4.3 page 17 *("The* `make` *command").*

1. Expliciter la règle suivante extraite de l'exemple de la section 4.3.7 page 31.

```
make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) -o make
        @size make
```

2. Expliciter la règle suivante :

```
gram.o:  lex.c
```

4.2 Evaluation de fonctions

On considère à nouveau les fichiers `fonction.h`, `fonction.C` et `tp3.1.C` du TP précédent (TP 3).

4.2.1 Un “ make ” explicite

1. Ecrire un fichier `makefile` qui automatise la compilation de `tp3.1.C` en utilisant la librairie `libtp3.a` (voir section 3.2.2 page 12).

   Toutes les étapes de cette compilation seront explicitement décrites par des règles `make` .

2. Montrer l'intérêt d'utiliser la commande `make`.

4.2.2 Un “ make ” implicite

1. Quelles sont les règles par défaut utilisées par la commande `make` pour générer un `.o` à partir d'un `.C`, un `.a` à partir d'un `.C` (voir section 4.3.5 page 24) ?

2. Modifier le fichier `makefile` précédent pour utiliser ces règles par défaut.

4.3 The make command

4.3.1 Command overview

Managing projects

The make program is a useful utility that can save you time when managing projects.
The make command assists you in maintaining a set of programs, usually pertaining to a particular software project. It does this by building up–to–date versions of programs.
In any project, you normally link programs from object files and libraries. Then, after modifying a source file, you recompile some of the sources and relink the program as often as required.
The make command simplifies the process of recompiling and relinking programs. It allows you to record, once only, specific relationships among files. You can then use the make command to automatically perform all updating tasks.
Using the make command to maintain programs, you can :

- Combine instructions for creating a large program in a single file
- Define macros to use within the make command description file
- Use shell commands to define the method of file creation, or use the make program to create many of the basic types of files
- Create libraries

The make program is most useful for medium–sized programming projects. It does not solve the problems of maintaining more than one source version and of describing large programs (see sccs command).
The make command requires a description file, file names, speci- fied rules to tell the make program how to build many standard types of files, and time stamps of all system files.
The make program uses information from a description file that you create to build a file containing the completed program, which is then called a target file.

Creating a Description File

The description file tells the make command how to build the target file, which files are involved, and what their relationships are to the other files in the procedure.
The description file contains the following information :

- Target file name
- Parent file names that make up the target file
- Commands that create the target file from the parent files
- Definitions of macros in the description file
- User-specified rules for building target files

By checking the dates of the parent files, the `make` program determines which files to create to get an up–to–date copy of the target file. If any parent file was changed more recently than the target file, the `make` command creates the files affected by the change, including the target file.
If you name the description file `makefile` or `Makefile` and are working in the directory containing that description file, enter :

```
make
```

to update the first target file and its parent files. Updating occurs regardless of the number of files changed since the last time the `make` command created the target file. In most cases, the description file is easy to write and does not change often.
To keep many different description files in the same directory, name them differently. Then, enter :

```
make -f Desc-File
```

substituting the name of the description file for the `Desc-File` variable.

How the `make` command creates a target file ?

The `make` command creates a file containing the completed program called a target file, using a step–by–step procedure. The `make` program :

1. Finds the name of the target file in the description file or in the `make` command
2. Ensures that the files on which the target file depends exist and are up–to–date
3. Determines if the target file is up–to–date with the files it depends on.

If the target file or one of the parent files is out–of–date, the `make` program creates the target file using one of the following :

- Commands from the description file
- Internal rules to create the file (if they apply)
- Default rules from the description file.

If all files in the procedure are up–to–date when running the `make` program, the `make` command displays a message to indicate that the file is up–to–date, and then stops. If some files have changed, the `make` command builds only those files that are outi–of–date. The command does not rebuild files that are already current.
When the `make` program runs commands to create a target file, it replaces macros with their values, writes each command line, and then passes the command to a new copy of the shell.

4.3.2 Synopsis

Purpose :

Maintains up–to–date versions of programs.

Syntax :

```
make [-deiknpqrSst] [-f MakeFile ...]  [TargetFile ...]
```

You can also include macro definitions on the command line after all of the flags. Macro definitions have the form :

```
macro-name = string
```

The make command considers all entries on the command line that follow the flags and that do not contain an equal sign to be target file names.

Description :

The make command reads `MakeFile` for information about the specified `TargetFile` and for the commands necessary to update them. The make command does not change the `TargetFile` if you have not changed any of the source files since you last built it. It considers a missing file to be a changed file (out-of-date).

The default shell is determined by :

- The `MAKESHELL` environment variable, if defined. (BSD users should set this to `/usr/bin/sh`.)
- If `MAKESHELL` is undefined, the program uses the value of `SHELL` that you set from the description file or the command line. Note that a `SHELL` set from the environment is always ignored.
- If `MAKESHELL` and `SHELL` are undefined, the program uses the default value `/usr/bin/sh`.

The make command stores its default rules in the file `/usr/ccs/lib/make.cfg`. The `/usr/ccs/lib/make.cfg` file is a configuration file that contains default rules for the make command. These defaults may be overriden in the user `Makefile`. You may substitute your own rules file by setting the `MAKERULES` variable to your own file name from the command line. The following lines show how to change the rules file from the command line :

```
make MAKERULES=/pathname/filename
```

Exit Status :

When the `-q` flag is specified, this command returns the following exit values :

`0`	:	Successful completion
`1`	:	The target was not up–to–date
`>1`	:	An error occurred

Otherwise, this command returns the following exit values :

`0`	:	Successful completion
`>0`	:	An error occurred

Examples :

1. To make the file specified by the first entry in the description file :

```
make
```

2. To display, but not run, the commands that the `make` command would use to make a file :

```
make -n search.o
```

You may want to do this to verify that a new description file is correct before using it.

Files :

`makefile`	:	contains `make` command description rules
`Makefile`	:	contains `make` command description rules
`s.makefile`	:	contains `make` command description rules
`s.Makefile`	:	contains `make` command description rules
`/usr/ccs/lib/make.cfg`	:	contains `make` command default rules

4.3.3 Description file

The `make` command searches the current directory for the first description file it finds that is named `makefile`, `Makefile`, `s.makefile`, or `s.Makefile`, in that order. The `-f` option may be used to override this naming convention.

Target files

The description file contains a sequence of entries specifying the files that the `target` files depend on. The general form of an entry is :

```
target [target ...]  :[:]  [file ...]  [‘command‘] [;command]
[(tab)Command ...]
```

The first line of an entry (called the dependency line), contains a list of `target`s followed by a `:` (colon) and an optional list of prerequisite `file`s or dependencies or shell command expressions, which are commands enclosed in single backquotes (‘) whose output is treated as dependencies. If you put shell commands on the dependency line, they must be preceded by a `;` (semicolon). All commands that follow the semicolon and all following lines that begin with a tab contain shell `command`s that the `make` command uses to build the `target`s.
To specify more than one set of commands for a `TargetFile`, you must enter more than one dependency definition. In this case, each definition must have the `TargetFile` name followed by two colons (`::`), a dependency list, and a command list.

Special Target Names

You can use some special target names in the description file to tell the make command to process the file in a different manner. These are :

.DEFAULT The commands that appear after this name in the description file tell the make command what to do if it can find no commands or default rules to tell it how to create a specific file.

.IGNORE If this name appears on a line by itself, the make command does not stop when errors occur. If this has prerequisites, the make command ignores errors associated with them. Using a - (minus) as the first character on a command line in the description file tells the make command to ignore errors for the command on that line only.

.POSIX The make command processes the makefile as specified by the POSIX standard.

.PRECIOUS The files named on the same line as this special name are not removed when the make command is interrupted. If no file names are specified, all files are treated as if specified with .PRECIOUS .

.SILENT If this name appears on a line by itself, the make command does not display any of the commands that it performs to build a file. If this has prerequisites, the make command does not display any of the commands associated with them.

.SUFFIXES Use this name to add more suffixes to the list of file suffixes that make recognizes.

Command lines

The first line that does not begin with a tab or # (pound sign) begins a new dependency or a macro definition. Command lines are performed one at time, each by its own subshell. Thus, the effect of some shell commands, such as cd, does not extend across new-line characters. You can, however, put a \ (backslash) at the end of a line to continue it on the next physical line.
Commands are executed by passing the command line to the command interpreter. If commands are executed in the background, the make command does not wait for the termination of the background processes before it exits.
The first one through three characters in a command can be one of the following special characters:

- `-` ignores errors returned by the command on this line
- `@` does not display this command line
- `+` executes this command line even if -n, -q or -t is specified

The `VPATH` variable

The `VPATH` variable is set with a list of directory path names separated by a `:` (colon). The `make` command uses `VPATH` to search for a dependency-related file within these directories. If this macro is defined, then these directories are searched. If this macro is not defined or just defined with a `.` (dot), only the current directory is searched. The default for `VPATH` is not defined.

Comments

A `#` (pound sign) is used to put comments in a description file. All characters following a `#` and continuing to the end of the line are considered to be part of the comment. Comments may be specified on individual lines as well as on dependency lines and command lines in the description file. The description file can contain blank lines, which will be ignored by the `make` program. However, no blank lines are allowed in a continued list.

4.3.4 Macros

User macros

Entries of the form `String1 = String2` are macro definitions. `String2` consists of all characters that occur on a line before a comment character (`#`) or before a new-line character that is not a continuation line. After this macro definition, the make command replaces each `$(String1)` in the file with `String2`.
The `make` command also handles nested macro names, in the form of `$($(String1))`. Where the innermost macro name is expanded, then its value is taken as another macro name that must also be expanded. Ten levels of nesting for macro expansion is supported. You do not have to use the parentheses around the macro name if the macro name is only one character long and there is no substitute sequence (see the next paragraph). If you use the following form, you can also replace characters in the macro string with other characters for one time that you use the macro :

```
$(String1[:Substitute1=[Substitute2]])
```

The optional `:Substitute1 = Substitute2` specifies a substitute sequence. If you specify a substitute sequence, the `make` program replaces each `Substitute1` pattern in the named macro with `Substitute2` (if `Substitute1` is a suffix in `String1`). Strings in a substitute sequence begin and end with any of the following : a blank, tab, new-line character, or beginning of line.

Note : Because the `make` command uses the dollar sign symbol (`$`) to designate a macro, do not use that symbol in filenames of targets and parents, or in commands in the description file unless you are using a defined `make` macro.

Internal macros

The `make` command has six internal macros. It assigns values to these macros under one or more of the following conditions :

- when it uses an internal rule to build a file
- when it uses a .DEFAULT rule to build a file
- when it uses rules in the description file to build a file
- when the file is a library member.

The six internal macros are defined as follows :

1. $* : the file name (without the suffix) of the source file.
2. $@ : the full target name of the current target.
3. $< : the source files of an out-of-date module.

   The make command evaluates this macro when applying inference rules or the .DEFAULT rule.

   For example :

```
.c.o:
        cc -c $<
```

   Here, $< refers to the .c file of any out-of-date .o file.

4. $? : the list of out-of-date files.

   The make command evaluates this macro when it evaluates explicit rules from MakeFile.

5. $% : the name of an archive library member.

   The make command evaluates this macro only if the target is an archive library member of the form lib(file.o). In this case, $@ evaluates to lib and $% evaluates to the library member, file.o .

   You can add an uppercase D or F to indicate "directory part" or "file part", respectively, to all internal macros except for $? . Thus, $(@D) refers to the directory part of the name $@ . If there is no directory part, the make command uses ./.

6. $$: the dollar sign in a description file.

   A single $ is interpreted by the make command as a macro to be evaluated, as in the above definitions.

Environment

When you run the make command, it reads the environment and treats all variables as macro definitions.
The make command processes macro definitions in the following order :

1. The make command's own default rules
2. Environment variables

3. Description files

4. Command line.

Therefore, macro assignments in a description file normally override duplicate environment variables. The `-e` flag instructs the `make` command to use the environment variables instead of the description file macro assignments.
The `make` command recognizes the `MAKEFLAGS` and `MFLAGS` environment macros, which the user can set only as environment variable macros that can be assigned any `make` flag value except `-f`, `-p`, and `-d`. The make command uses the value of `MFLAGS` only if the `MAKEFLAGS` macro is not defined. When the `make` program starts, it invokes itself with the options specified with these environment variables, if defined, in addition to those specified on the command line. It passes these options to any command it invokes, including additional invocations of the `make` command itself. Thus you can perform a `make -n` recursively on a software system to see what would have been performed. The `-n` is passed to further copies of the shell that runs the next level of `make` commands. In this way, you can check all of the description files for a software project without actually compiling the project.

4.3.5 Suffix Rules

The `make` command has default rules that govern the building of most standard files. These rules depend on the standard suffixes used by the system utility programs to identify file types. These rules define the starting and ending file types so that, for example, given a specified `.o` file, the `make` command can infer the existence of a corresponding `.c` file and knows to compile it using the `cc -c` command.
A rule with only one suffix (that is, `.c:`) defines the building of prog from all its source files. Use a `~` (tilde) in the suffix to indicate an `SCCS` file. For example, the `.c~.o` rule governs changing an `SCCS` C source file into an object file. You can define rules within the description file. The `make` command recognizes as a rule any target that contains no slashes and starts with a dot.
You can also add suffixes to the list of suffixes recognized by the make command and add to the default dependency rules. Use the target name `.SUFFIXES` followed by the suffixes you want to add. Be careful of the order in which you list the suffixes. The `make` command uses the first possible name for which both a file and a rule exist. The default list is :

```
.SUFFIXES: .o .c .c~.f .f~.y .y~.l .l~.s .s~.sh .sh~.h .h~.a
```

You can clear the list of suffixes by including `.SUFFIXES:` with no following list.
If a target name contains parentheses, the `make` command considers it an archive library. The string within parentheses refers to a library member. Thus, `lib(File.o)` and `$(LIB)(File.o)` both see an archive library which contains `File.o` . (You must have already defined the `LIB` macro.) The expression `$(LIB) + (File1.o File2.o)` is not legal. Rules that apply to archive libraries have the form `.x.a`, where `.x` is the suffix of the file you want to add to an archive library. For example, `.c.a` indicates a rule that changes any C source file to a library file member.

Internal rules for the make program

The internal rules for the make program are in a file that looks like a description file. When the -r flag is specified, the make program does not use the internal rules file. You must supply the rules to create the files in your description file. The internal-rules file contains a list of file-name suffixes (such as .o, or .a) that the make command understands, plus rules that tell the make command how to create a file with one suffix from a file with another suffix. If you do not change the list, the make command understands the following suffixes :

.a	Archive library
.c	C source file
.c~	Source Code Control System (SCCS) file containing C source file
.C	C++ source file
.C~	Source Code Control System (SCCS) file containing C++ source file
.f	FORTRAN source file
.f~	SCCS file containing FORTRAN source file
.h	C language header file
.h~	SCCS file containing C language header file
.l	lex source grammar
.l~	SCCS file containing lex source grammar
.o	Object file
.s	Assembler source file
.s~	SCCS file containing assembler source file
.sh	Shell-command source file
.sh~	SCCS file containing shell-command source file
.y	yacc-c source grammar
.y~	SCCS file containing yacc-c source grammar

The list of suffixes is similar to a dependency list in a description file, and follows the fake target name .SUFFIXES. Because the make command looks at the suffixes list in left–to–right order, the order of the entries is important. The make program uses the first entry in the list that satisfies the following two requirements :

- The entry matches input and output suffix requirements for the current target and dependency files.
- The entry has a rule assigned to it.

The make program creates the name of the rule from the two suffixes of the files that the rule defines. For example, the name of the rule to transform a .c file to an .o file is .c.o .

To add more suffixes to the list, add an entry for the fake target name .SUFFIXES in the description file. For a .SUFFIXES line without any suffixes following the target name in the description file, the make command erases the current list. To change the order of the names in the list, erase the current list and then assign a new set of values to .SUFFIXES .

```
# Create a .o file from a .c
# file with the cc program.
c.o:
        $(CC) $(CFLAGS) -c $<
# Create a .o file from
# a .s file with the assembler.
s.o:
        $(AS) $(ASFLAGS) -o $@ $<
.y.o:
        # Use yacc to create an intermediate file
        $(YACC) $(YFLAGS) $<
        # Use cc compiler
        $(CC) $(CFLAGS) -c y.tab.c
        # Erase the intermediate file
        rm y.tab.c
        # Move to target file
        mv y.tab.o $@.
.y.c:
         # Use yacc to create an intermediate file
         $(YACC) $(YFLAGS) $<
         # Move to target file
         mv y.tab.c $@
```

The `make` program uses macro definitions in the rules file. To change these macro definitions, enter new definitions for each macro on the command line or in the description file. The `make` program uses the following macro names to represent the language processors that it uses :

`AS`	For the assembler
`CC`	For the C compiler
`CCC`	For the C++ compiler
`YACC`	For the yacc command
`LEX`	For the lex command

The `make` program uses the following macro names to represent the flags that it uses:

`CFLAGS`	For C compiler flags
`CCFLAGS`	For C++ compiler flags
`YFLAGS`	For yacc command flags
`LFLAGS`	For lex command flags

Therefore, the command:

```
make "CC=NEWCC"
```

directs the `make` command to use the `NEWCC` program in place of the usual C language compiler. Similarly, the command:

```
make "CFLAGS=-O"
```

directs the `make` command to optimize the final object code produced by the C language compiler.
To review the internal rules, refer to `/usr/ccs/lib/make.cfg` file.

/usr/ccs/lib/make.cfg file

```
.SUFFIXES: .o .c .c~ .f .f~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~ .C .C~
MAKE    = make
AR      = ar
ARFLAGS = -rv
YACC    = yacc
YFLAGS  =
LEX     = lex
LFLAGS  =
LD      = ld
LDFLAGS =
CC      = cc
CFLAGS  = -O
FC      = xlf
FFLAGS  = -O
AS      = as
ASFLAGS =
GET     = get
GFLAGS  =
.c:
        $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
.c~:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
        -rm -f $*.c
.f:
        $(FC) $(FFLAGS) $(LDFLAGS) $< -o $@
.f~:
        $(GET) $(GFLAGS) -p $< > $*.f
        $(FC) $(FFLAGS) $(LDFLAGS) $*.f -o $*
        -rm -f $*.f
.sh:
        cp $< $@; chmod 0777 $@
.sh~:
        $(GET) $(GFLAGS) -p $< > $*.sh
        cp $*.sh $*; chmod 0777 $@
        -rm -f $*.sh
.c.o:
        $(CC) $(CFLAGS) -c $<
.c~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
.c~.c:
        $(GET) $(GFLAGS) -p $< > $*.c
.s.o:
        $(AS) $(ASFLAGS) -o $@ $<
```

```
.s~.o:
	$(GET) $(GFLAGS) -p $< > $*.s
	$(AS) $(ASFLAGS) -o $*.o $*.s
	-rm -f $*.s

.f.o:
	$(FC) $(FFLAGS) -c $<

.f~.o:
	$(GET) $(GFLAGS) -p $< > $*.f
	$(FC) $(FFLAGS) -c $*.f
	-rm -f $*.f

.f~.f:
	$(GET) $(GFLAGS) -p $< > $@

.y.o:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
	rm y.tab.c
	mv y.tab.o $@

.y~.o:
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	$(CC) $(CFLAGS) -c y.tab.c
	rm -f y.tab.c $*.y
	mv y.tab.o $*.o

.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm lex.yy.c
	mv lex.yy.o $@

.l~.o:
	$(GET) $(GFLAGS) -p $< > $*.l
	$(LEX) $(LFLAGS) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c $*.l
	mv lex.yy.o $*.o

.y.c :
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@

.y~.c :
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	mv y.tab.c $*.c
	-rm -f $*.y

.l.c :
	$(LEX) $<
	mv lex.yy.c $@
```

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
.c~.a:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) -c $(CFLAGS) $*.c
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.[co]
.s~.a:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[so]
.f.a:
        $(FC) -c $(FFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
.f~.a:
        $(GET) $(GFLAGS) -p $< > $*.f
        $(FC) -c $(FFLAGS) $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.[fo]
.h~.h:
        $(GET) $(GFLAGS) -p $< > $*.h
markfile.o:     markfile
        cc -c markfile.c
        rm -f markfile.c
```

IBM C++ Compiler rules

```
CCC     = xlC
CCFLAGS = -O
.C:
        $(CCC) $(CCFLAGS) $(LDFLAGS) $< -o $@
.C~:
        $(GET) $(GFLAGS) -p $< > $*.C
        $(CCC) $(CCFLAGS) $(LDFLAGS) $*.C -o $*
        -rm -f $*.C
.C.o:
        $(CCC) $(CCFLAGS) -c $<
.C~.o:
        $(GET) $(GFLAGS) -p $< > $*.C
        $(CCC) $(CCFLAGS) -c $*.C
        -rm -f $*.C
.C~.C:
        $(GET) $(GFLAGS) -p $< > $*.C; chmod 444 $*.C
```

```
.C.a:
        $(CCC) -c $(CCFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
.C~.a:
        $(GET) $(GFLAGS) -p $< > $*.C
        $(CCC) -c $(CCFLAGS) $*.C
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.[Co]
```

4.3.6 Flags

`-d` Displays detailed information about the files and times that `make` examines (debug mode).

`-e` Uses environment variables in place of any assignments made within description files. These assignments normally replace environment variables.

`-f MakeFile` Reads `MakeFile` for a description of how to build the target file. If you give only a `-` (minus sign) for `MakeFile`, the `make` command reads standard input. If you do not use the `-f` flag, the `make` command searches the current directory for the first description file named `makefile`, `Makefile`, `s.makefile`, or `s.Makefile`, in that order. You can specify more than one description file by entering the `-f` flag more than once (with its associated `MakeFile` parameter).

`-i` Ignores error codes returned by commands. The `make` command normally stops if a command returns a nonzero code. Use this flag to compile several modules only if you want the `make` command to continue when an error occurs in one of the modules.

`-k` Stops processing the current target if an error occurs, but continues with other branches that do not depend on that target.

`-n` Displays commands, but does not run them. However, lines beginning with a `+` (plus sign) are executed. Displays lines beginning with an `@` (at sign). If the special target `.POSIX` is not specified and the command in the description file contains the string `$(MAKE)`, perform another call to the `make` command. Use this flag to preview the performance of the `make` command.

`-p` Displays the complete set of macro definitions and target descriptions before performing any commands.

`-q` Returns a zero status code if the target file is up–to–date; returns a one status code if the target file is not up–to–date. However, a command lines with the `+` (plus sign) prefix will be executed.

`-r` Does not use the default rules. Uses only rules specified in the description file.

`-S` Terminates the `make` command if an error occurs. This is the default and the opposite of `-k` flag. If the `-k` and `-S` flags are both specified, the last one evaluated works.

`-s` Does not display commands or touch messages on the screen as they are performed.

`-t` Changes only the date of the files, rather than performing the listed commands. Use this flag if you have made only minor changes to a source file that do not affect anything outside of that file. This flag changes the date of all target files that appear on the command line or in the description file. However, command lines beginning with a `+` (plus sign) are executed.

4.3.7 Example

Description file

The following example description file could maintain the make program. The source code for the `make` command is spread over a number of C language source files and a yacc grammar.

```
# Description file for the Make program

# Macro def: send to be printed
P = qprt

#  Macro def: source filenames used
FILES = Makefile version.c defs main.c \
        doname.c misc.c files.c \
        dosy.c gram.y lex.c gcos.c

# Macro def: object filenames used
OBJECTS = version.o main.o doname.o \
        misc.o files.o dosys.o \
        gram.o

# Macro def: lint program and flags
LINT = lint -p

# Macro def: C compiler flags
CFLAGS = -O

# make depends on the files specified
# in the OBJECTS macro definition
make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) -o make# Build make with the cc program
        @size make                    # Show the file sizes

# The object files depend on a file
# named defs
$(OBJECTS):  defs
```

```
# The file gram.o depends on lex.c
# uses internal rules to build gram.o
gram.o:  lex.c

# Clean up the intermediate files
clean:
        -rm *.o gram.c
        -du

# Copy the newly created program
# to /usr/bin and deletes the program
# from the current directory
install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

# Empty file "print" depends on the
# files included in the macro FILES
print: $(FILES)
        pr $? | $P  # Print the recently changed files
        touch print # Change the date on the empty file,
                    # print, to show the date of the last
                    # printing

# Check the date of the old
# file against the date
# of the newly created file
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

# The program, lint, depends on the
# files that are listed
lint: dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys. doname.c files.c main.c \
                misc.c version.c gram.c # Run lint on the files listed
                                        # LINT is an internal macro
        rm gram.c

# Archive the files that build make
arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Output results

The `make` program usually writes out each command before issuing it.

The following output results from entering the simple make command in a directory containing only the source and description file.

```
$ make
        cc -O -c version.c
        cc -O -c main.c
        cc -O -c doname.c
        cc -O -c misc.c
        cc -O -c files.c
        cc -O -c dosys.c
        yacc  gram.y
        mv y.tab.c gram.c
        cc -O -c gram.c
        cc version.o main.o doname.o misc.o files.o dosys.o
                gram.o -o make
13188+3348+3044 = 19580b = 046174b
```

None of the source files or grammars are specified in the description file. However, the make command uses its suffix rules to find them and then issues the needed commands. The string of digits on the last line of the previous example results from the size make command. Because the @ (at sign) on the size command in the description file prevented writing of the command, only the sizes are written.
The output can be sent to a different printer or to a file by changing the definition of the P macro on the command line, as follows :

```
make print "P = print -sp"
```

or

```
make print "P = cat > zap"
```

T.P. 5

Entrées/sorties en C++

5.1 Les classes d'entrées/sorties

5.1.1 Le graphe d'héritage

1. Consulter les extraits du manuel UNIX reproduits en section 5.3 page 37 .

2. Représenter le graphe d'héritage des principales classes C++ d'entrées/sorties.

5.1.2 Injection dans un flot

1. Consulter l'interface de la classe `ostream` (section 5.3.7 page 41).

2. Ecrire un programme C++ (`cp0.C`) qui vérifie que l'opérateur d'injection (`<<`) est bien surdéfini pour les `int`, les `double` et les `char*` .

3. Compléter ce programme pour vérifier que l'opérateur `<<` est bien associatif à gauche.

5.1.3 Extraction d'un flot

1. Consulter l'interface de la classe `istream` (section 5.3.8 page 42).

2. Compléter le programme C++ précédent (`cp0.C`) pour vérifier que l'opérateur d'extraction (`>>`) est bien surdéfini pour les `int`, les `double` et les `char*` .

3. Compléter ce programme pour vérifier que l'opérateur `>>` est bien associatif à gauche.

5.2 Copier un fichier

5.2.1 `cin` → `cout`

1. Ecrire un programme C++ (`cp1.C`) qui copie l'entrée standard `cin` sur la sortie standard `cout`.

2. Comment utiliser le programme précédent, sans le modifier, pour qu'il copie un fichier quelconque sur un autre fichier ?

5.2.2 `source` → `destination`

1. Ecrire un programme C++ (`cp2.C`) qui copie un fichier `source` sur un fichier `destination`.

2. Modifier le programme précédent pour que les noms des fichiers `source` et `destination` soient passés dans la ligne de commande.

5.3 iostream package

5.3.1 Synopsis

Purpose :

buffering, formatting and input/output

Include files :

```
#include <iostream.h>
    class streambuf ;
    class ios ;
    class istream :  virtual public ios ;
    class ostream :  virtual public ios ;
    class iostream :  public istream, public ostream ;
    class istream_withassign :  public istream ;
    class ostream_withassign :  public ostream ;
    class iostream_withassign :  public iostream ;
    extern istream_withassign cin ;
    extern ostream_withassign cout ;
    extern ostream_withassign cerr ;
    extern ostream_withassign clog ;
#include <fstream.h>
    class filebuf :  public streambuf ;
    class fstream :  public iostream ;
    class ifstream :  public istream ;
    class ofstream :  public ostream ;
#include <strstream.h>
    class strstreambuf :  public streambuf ;
    class istrstream :  public istream ;
    class ostrstream :  public ostream ;
#include <stdiostream.h>
    class stdiobuf :  public streambuf ;
    class stdiostream :  public ios ;
```

Description :

The C++ `iostream` package declared in `iostream.h` and other header files consists primarily of a collection of classes.

The `iostream` package consists of several core classes, which provide the basic functionality for I/O conversion and buffering, and several specialized classes derived from the core classes.

5.3.2 Core classes

The core of the `iostream` package comprises the following classes :

`streambuf`

This is the base class for buffers. It supports insertion (also known as storing or putting) and extraction (also known as fetching or getting) of characters. Most members are inlined for efficiency.

`ios`

This class contains state variables that are common to the various stream classes, for example, error states and formatting states.

`istream`

This class supports formatted and unformatted conversion from sequences of characters fetched from streambufs.

`ostream`

This class supports formatted and unformated conversion to sequences of characters stored into streambufs.

`iostream`

This class combines istream and ostream. It is intended for situations in which bidirectional operations (inserting into and extracting from a single sequence of characters) are desired.

`istream_withassign, ostream_withassign, iostream_withassign`

These classes add assignment operators and a constructor with no operands to the corresponding class without assignment. The predefined streams (see below) cin, cout, cerr, and clog, are objects of these classes.

5.3.3 Predefined streams

The following streams are predefined :

`cin`

The standard input (file descriptor 0).

`cout`

The standard output (file descriptor 1).

`cerr`

Standard error (file descriptor 2).

Output through this stream is unit-buffered, which means that characters are flushed after each inserter operation.

`clog`

This stream is also directed to file descriptor 2, but unlike `cerr` its output is buffered.

`cin`, `cerr`, and `clog` are tied to `cout` so that any use of these will cause `cout` to be flushed.

5.3.4 Classes derived from `streambuf`

Classes derived from `streambuf` define the details of how characters are produced or consumed. The available buffer classes are :

`filebuf`

This buffer class supports I/O through file descriptors. Members support opening, closing, and seeking. Common uses do not require the program to manipulate file descriptors.

`stdiobuf`

This buffer class supports I/O through stdio `FILE` structs. It is intended for use when mixing C and C++ code. New code should prefer to use `filebuf`s.

`strstreambuf`

This buffer class stores and fetches characters from arrays of bytes in memory (i.e., strings).

5.3.5 Classes derived from `istream`, `ostream`, and `iostream`

Classes derived from `istream`, `ostream`, and `iostream` specialize the core classes for use with particular kinds of `streambuf`s. These classes are :

`ifstream, ofstream, fstream`

These classes support formatted I/O to and from files. They use a filebuf to do the I/O. Common operations (such as opening and closing) can be done directly on streams without explicit mention of `filebuf`s.

`istrstream, ostrstream`

These classes support "in core" formatting. They use a `strstreambuf`.

`stdiostream`

This class specializes iostream for stdio `FILE`s.

5.3.6 class ios

```
class ios {
public:
      enum io_state { goodbit=0, eofbit, failbit, badbit };
      enum open_mode { in, out, ate, app, trunc, nocreate, noreplace };
      enum seek_dir { beg, cur, end };
      /* flags for controlling format */
      enum {
            skipws=01, left=02, right=04, internal=010,
            dec=020, oct=040, hex=0100,
            showbase=0200, showpoint=0400, uppercase=01000, showpos=02000,
            scientific=04000, fixed=010000, unitbuf=020000, stdio=040000
      };
      static const long basefield;   /* dec|oct|hex */
      static const long adjustfield; /* left|right|internal */
      static const long floatfield;  /* scientific|fixed */
public:
      ios(streambuf*);
      int bad();
      static longbitalloc();
      void clear(int state =0);
      int eof();
      int fail();
      char fill();
      char fill(char);
      long flags();
      long flags(long);
      int good();
      long& iword(int);
      int operator!();
      operator void*();
      int precision();
      int precision(int);
      streambuf* rdbuf();
      void*& pword(int);
      int rdstate();
      long setf(long setbits, long field);
      long setf(long);
      static void sync_with_stdio();
      ostream* tie();
      ostream* tie(ostream*);
      long unsetf(long);
      int width();
      int width(int);
      static int xalloc();
protected:
      ios();
      int init(streambuf*);
private:
      ios(ios&);
      void operator=(ios&);
};
```

```
ios& dec(ios&) ;
ios& hex(ios&) ;
ios& oct(ios&) ;
```

5.3.7 class ostream

```
class ostream :  public ios {
public:
      ostream(streambuf*);
      ostream& flush();
      int opfx();
      ostream& put(char);
      ostream& seekp(streampos);
      ostream& seekp(streamoff, seek_dir);
      streampos tellp();
      ostream& write(const char* ptr, int n);
      ostream& write(const unsigned char* ptr, int n);
      ostream& operator<<(const char*);
      ostream& operator<<(char);
      ostream& operator<<(short);
      ostream& operator<<(int);
      ostream& operator<<(long);
      ostream& operator<<(float);
      ostream& operator<<(double);
      ostream& operator<<(unsigned char);
      ostream& operator<<(unsigned short);
      ostream& operator<<(unsigned int);
      ostream& operator<<(unsigned long);
      ostream& operator<<(void*);
      ostream& operator<<(streambuf*);
      ostream& operator<<(ostream& (*)(ostream&));
      ostream& operator<<(ios& (*)(ios&));
};
```

```
class ostream_withassign :  public ostream {
      ostream_withassign();
      istream& operator=(istream&);
      istream& operator=(streambuf*);
};
```

```
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;
ostream& endl(ostream&) ;
ostream& ends(ostream&) ;
ostream& flush(ostream&) ;
```

5.3.8 class istream

```
class istream :  public ios {
public:
      istream(streambuf*);
      int gcount();
      istream& get(char* ptr, int len, char delim='\n');
      istream& get(unsigned char* ptr,int len, char delim='\n');
      istream& get(unsigned char&);
      istream& get(char&);
      istream& get(streambuf& sb, char delim ='\n');
      int get();
      istream& getline(char* ptr, int len, char delim='\n');
      istream& getline(unsigned char* ptr, int len, char delim='\n');
      istream& ignore(int len=1,int delim=EOF);
      int ipfx(int need=0);
      int peek();
      istream& putback(char);
      istream& read(char* s, int n);
      istream& read(unsigned char* s, int n);
      istream& seekg(streampos);
      istream& seekg(streamoff, seek_dir);
      int sync();
      streampos tellg();
      istream& operator>>(char*);
      istream& operator>>(char&);
      istream& operator>>(short&);
      istream& operator>>(int&);
      istream& operator>>(long&);
      istream& operator>>(float&);
      istream& operator>>(double&);
      istream& operator>>(unsigned char*);
      istream& operator>>(unsigned char&);
      istream& operator>>(unsigned short&);
      istream& operator>>(unsigned int&);
      istream& operator>>(unsigned long&);
      istream& operator>>(streambuf*);
      istream& operator>>(istream& (*)(istream&));
      istream& operator>>(ios& (*)(ios&));
};
```

```
class istream_withassign :  public istream {
      istream_withassign();
      istream& operator=(istream&);
      istream& operator=(streambuf*);
};
```

```
extern istream_withassign cin;
istream& ws(istream&);
```

T.P. 6

Un petit analyseur lexical

6.1 Lexèmes

Un lexème constitue une unité lexicale d'un langage donné (un mot du langage).
Nous considérons ici un langage arithmétique formé de réels, de fonctions (suite de caractères alphabétiques) et des 4 opérateurs `+`, `-`, `*` et `/`. Les "mots" de ce langage (`REEL`, `OPERATEUR`, `FONCTION`) peuvent être séparés par un ou plusieurs espaces (au sens de la macro `isspace` défini dans le fichier standard `ctype.h`)
Le fichier `lexeme.h` ci–dessous précise les notations et les définitions utilisées par la suite.

```
// lexeme.h

#ifndef LEXEME_H
#define LEXEME_H

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

const int MAXCARLEX = 80;

enum TypeLexeme {FIN,REEL,OPERATEUR,FONCTION,INCONNU};

struct Lexeme {
      TypeLexeme type;
      union {
            double reel;
            char chaine[MAXCARLEX];
      } valeur;
};

Lexeme scanner(istream& = cin);

#endif
```

6.1.1 Représentation des lexèmes

1. Proposer une formulation équivalente du type énuméré `TypeLexeme` (voir `lexeme.h`).
2. Quelle différence principale existe–t–il entre une `union` et une `struct` ?
3. Comment est défini ici un lexème (expliciter la structure `Lexeme`; voir `lexeme.h`) ?

6.1.2 Lecture d'un lexème

On considère les déclarations suivantes :

```
Lexeme lex, *plex;
```

1. Comment peut–on savoir si la variable `lex`, supposée déjà initialisée, représente un réel ?
2. Comment accède–t–on à la valeur numérique d'un réel à partir de la variable `lex`, supposée représentant un réel ?
3. Comment accède–t–on à la valeur numérique d'un réel à partir du pointeur `plex`, supposé représentant un réel ?

6.1.3 Ecriture d'un lexème

On considère les déclarations suivantes :

```
Lexeme lex, *plex;
```

1. Initialiser la variable `lex` pour qu'elle représente un réel de valeur `3.14` .
2. Initialiser le pointeur `plex` pour qu'il représente un réel de valeur `3.14` .

6.2 Analyse lexicale

6.2.1 Analyseur lexical

Définir la fonction `Lexeme scanner(istream& in);` qui effectue l'analyse lexicale des caractères lus sur le flot d'entrée `in` (par défaut `cin`) selon le langage arithmétique défini en section 6.1 page 43, et retourne le `Lexeme` reconnu.

6.2.2 Evaluation de fonctions

Utiliser la fonction `Lexeme scanner(istream&);` dans un programme qui, pour chaque lexème reconnu, affiche sa valeur. Dans le cas d'une fonction, on appliquera cette fonction à la valeur `3.14` (voir fonction `trouverFonction` du T.P. 3).

T.P. 7

Mes premières classes

7.1 Conventions d'écriture

Pour chaque classe C++, on séparera son interface de son implémentation.

- L'interface sera écrite dans un fichier en–tête d'extension `.h` .
  - Le nom de la classe commence par une lettre majuscule.
  - Les parties `public`, `protected` et `private` sont spécifiées dans cet ordre.
  - Les noms des attributs et fonctions protégés et privés sont précédés du caractère souligné (_) .
- L'implémentation sera écrite dans un fichier d'extension `.C` .
  - Le code source sera suffisamment aéré et indenté.
  - Les éventuelles fonctions `inline` seront implémentées dans un fichier particulier d'extension `.Ci` (fichier à inclure dans le fichier `.h`).

7.1.1 Interface d'une classe

On considère le fichier `exemple.h` qui contient l'interface de la classe `Exemple` (voir section 7.3.1 page 47).

1. Rappeler le rôle des directives

   ```
   #ifndef EXEMPLE_H
   #define EXEMPLE_H
   ...#endif
   ```

2. Est–il nécessaire d'inclure `iostream.h` dans le fichier `exemple.h` ?

3. Que signifie l'expression `(int n = 0)` en argument du constructeur de la classe `Exemple` ?

4. Quel est le rôle du mot réservé `const` placé en fin d'un prototype de fonction ?

```
int observateur(void) const;
```

5. Quelle(s) différence(s) existe–t–il entre un passage par référence constante (`const Exemple& exemple`) et un passage par valeur (`Exemple exemple`) ?

7.1.2 Implémentation d'une classe

On considère le fichier `exemple.C` qui contient l'implémentation des fonctions (non `inline`) de la classe `Exemple` (voir section 7.3.3 page 49).

1. Pourquoi le fichier de l'interface de la classe (`exemple.h`) doit–il être systématiquement inclut au début du fichier de l'implémentation (`exemple.C`) de cette même classe ?

2. Que fait la fonction membre `void manipulateur(const int& n);` ?

3. Redéfinir l'opérateur d'injection `<<` en supposant qu'il n'a pas été déclaré `friend` .

7.1.3 Fonctions `inline`

On considère le fichier `exemple.Ci` qui contient l'implémentation des fonctions `inline` de la classe `Exemple` (voir section 7.3.2 page 48).

1. Quelles différences existe–t–il entre une fonction `inline` et une macro–instruction équivalente ?

2. Pourquoi le fichier de fonctions `inline` (`exemple.Ci`) doit–il être systématiquement inclut dans le fichier de l'interface de la classe (`exemple.h`) ?

3. Quel est le rôle de l'expression : `__attribut(n)` dans la définition du constructeur de la classe `Exemple` ?

7.2 Une classe `Point`

Un point du plan sera considéré ici comme un doublet de 2 réels (abscisse, ordonnée). On pourra extraire l'abscisse x et l'ordonnée y d'un point, et connaître la distance d'un point (x_1, y_1) à un autre point (x_2, y_2). Enfin, un point pourra être déplacé de $(\delta x, \delta y)$, et affiché sous la forme (x, y)

7.2.1 Du TAD à l'interface de la classe

1. Définir formellement le TAD `Point` en s'inspirant de la description informelle précédente.

2. Définir l'interface C++ de la classe `Point` correspondant au TAD `Point` .

7.2.2 L'implémentation de la classe

1. Définir les fonctions prévues dans l'interface de la classe `Point` .
2. Quelles fonctions pourrait–on définir `inline` ?

7.2.3 L'utilisation de la classe

1. Ecrire un programme (fichier `tp7.C`) qui teste le bon fonctionnement de la classe `Point` .
2. Inclure le programme précédent dans le fichier `exemple.C` de telle manière qu'il ne soit compilé que si la macro `TESTPOINT` est définie.

7.3 Les fichiers de la classe `Exemple`

7.3.1 Fichier `exemple.h`

```
// exemple.h

#ifndef EXEMPLE_H
#define EXEMPLE_H

#include <iostream.h>

class Exemple {
public:
      Exemple(int n = 0);
      int observateur(void) const;
      void manipulateur(cont int& n);
      friend ostream& operator<<(ostream& flot, const Exemple& exemple);
private:
      int _attribut;
};

#include "exemple.Ci"                                    // fonctions inline

#endif
```

7.3.2 Fichier `exemple.Ci`

```
// exemple.Ci

//------------------------------------------------------ Exemple exemple(n)
Exemple::Exemple(int n) :  _attribut(n) { }

//------------------------------------------------ n = exemple.observateur()
int
Exemple::observateur(void) const {
     return _attribut;
}
```

7.3.3 Fichier exemple.C

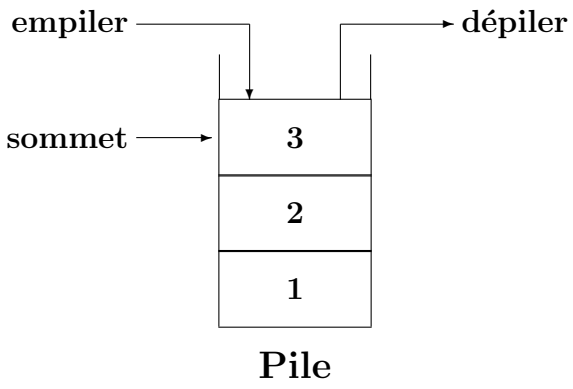
```
// exemple.C

#include "exemple.h"

//------------------------------------------ rappel des fonctions inline
Exemple::Exemple(int n);
int Exemple::observateur(void) const;

//-------------------------------------------- exemple.manipulateur(n)
void
Exemple::manipulateur(cont int& n) {
     int tmp = (n > 0) ?  n :  -n;
     for(int i = 0; i < tmp; i++) {
          _attribut += i;
     }
}

//------------------------------------------------------ flot << exemple
ostream&
operator<<(ostream& flot, const Exemple& exemple) {
     flot << '(' << exemple._attribut << ')';
     return flot;
}
```

T.P. 8

Mes premières piles

8.1 Du TAD au TCD

Une pile est une structure LIFO *(Last In, First Out)* illustrée sur la figure ci–dessous.



Pile

8.1.1 Le TAD `Pile`

1. Citer des exemples de piles rencontrées dans la vie courante.
2. Citer des exemples d'utilisation de piles en Informatique.
3. Proposer une définition du TAD `Pile`.
   (a) Observateurs
   (b) Manipulateurs
   (c) Préconditions associées
   (d) Axiomes
4. Vérifier que ce TAD ne suppose rien quant au type des éléments stockés dans une pile.

8.1.2 Interface publique d'une PILE

On veut définir une classe C++ PILE qui stocke des éléments de type TYPE.
Dans cette section, on ne s'intéresse qu'à la partie publique de l'interface de la classe PILE (fichier PILE.h).

1. Déclarer les prototypes des fonctions C++ associées aux services du TAD Pile.
2. Déclarer les prototypes d'un constructeur par défaut, d'un constructeur par recopie et du destructeur de la classe PILE.
3. Déclarer les prototypes de l'affectation, de l'égalité et de la non égalité.
4. Déclarer les prototypes des opérateurs >> et << pour qu'ils correspondent respectivement aux opérations depiler et empiler.
5. Déclarer le prototype de l'opérateur d'injection (<<) d'une pile dans un flot.

8.1.3 Choix d'une représentation physique

Dans cette section, on ne s'intéresse qu'à la partie privée de l'interface de la classe PILE.

1. Proposer différentes manières de stocker physiquement les éléments d'une pile. Discuter les avantages et les inconvénients de ces différentes représentations.
2. Déclarer les attributs privés nécessaires à la représentation d'une pile sous la forme d'un tableau statique de MAX éléments maximum.

8.1.4 Paramétrage de la classe PILE

1. Ecrire l'interface de la classe RPile (fichier rpile.h) qui correspond à une PILE de réels (double) implémentée sous la forme d'un tableau statique (on utilisera tel quel le fichier PILE.h).
2. Ecrire l'interface de la classe EPile (fichier epile.h) qui correspond à une PILE d'entiers (int) implémentée sous la forme d'un tableau statique (on utilisera tel quel le fichier PILE.h).

8.2 Implémentation en C++

Dans cette section, on choisit de représenter une pile par un tableau statique de MAX éléments de type TYPE.

8.2.1 La classe PILE

1. Définir les différentes fonctions de la classe PILE dans le fichier PILE.C .
2. Ecrire les implémentations des classes RPile (rpile.C) et EPile (epile.C) (on utilisera tel quel le fichier PILE.C).

8.2.2 Test de la classe PILE

On considère le programme de test suivant.

```
#ifdef TESTPILE

int main(void) {
      TYPE x, y, z, t;
      PILE p1, p2;

      cout << "Entrer 4 elements" << endl;
      cin >> x >> y >> z >> t;

      p1 << x << y ;
      p1.empiler(z);
      p1.empiler(t);
      p2 = p1;
      cout << p2 << endl;
      p1 >> x >> y ;

      PILE p3(p1);
      cout << p3 << endl;
      p1 >> z >> t ;
      cout << x << ' ' << y << ' ' << z << ' ' << t << endl;
      while(!p2.vide()) {
            cout << p2.depiler() << endl;
      }
      return 0;
}

#endif
```

1. Tester ce programme avec une pile de réels.
2. Tester ce programme avec une pile d'entiers.

T.P. 9

Ma première calculette à pile

On veut réaliser une calculette à pile à 4 registres qui soit capable d'évaluer les 4 opérateurs arithmétiques ainsi que les fonctions mathématiques usuelles : `acos`, `asin`, `atan`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `log`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`.
Un exemple d'utilisation de cette calculette est donné ci–dessous.

```
$ rcalc
?  1.2
= 1.2
?  5.6
= 5.6
?  +
5.6 + 1.2 = 6.8
?  sin
sin(6.8) = 0.494113
?  asin
asin(0.494113) = 0.516815
?  3.14159
= 3.14159
?  2
= 2
?  *
2 * 3.14159 = 6.28318
?  +
6.28318 + 0.516815 = 6.79999
?  flor
flor(6.79999) = ?
?  floor
floor(6.79999) = 6
⋮
$
```

Pour réaliser cette calculette on réutilisera :

- l'analyseur lexical du T.P. 6

```
Lexeme scanner(istream& in);
```

- l'évaluateur de fonctions mathématiques du T.P. 3

```
double (*trouverFonction(char* nom, Fonction* tab))(double);
```

- la classe `RPile` du T.P. 8.

La calculette elle–même ne sera pas vue ici comme une classe mais comme une fonction de test des 3 T.P. considérés (T.P. 3, 6 et 8).

9.1 Le fichier `Makefile`

La préparation de ce travail passe par l'écriture d'un fichier de règles pour la commande `make` (voir T.P. 4).

9.1.1 Compilation de l'analyseur lexical

1. Ecrire la règle `make` permettant d'obtenir le fichier `lexeme.o` dans le répertoire de travail `tp/src/tp9` sans déplacer le fichier source correspondant (`tp6/lexeme.C`) .
2. Vérifier le bon fonctionnement de cette règle par la commande

```
make lexeme.o
```

9.1.2 Compilation de la `RPile`

1. Ecrire la règle `make` permettant d'obtenir le fichier `rpile.o` dans le répertoire de travail `tp/src/tp9` sans déplacer le fichier source correspondant (`tp8/rpile.C`) .
2. Vérifier le bon fonctionnement de cette règle par la commande

```
make rpile.o
```

9.1.3 Compilation de la calculette

La fonction test qui correspond à la calculette sera définie dans un fichier `rcalc.C` dont le squelette est donné ci–dessous.

1. Ecrire la règle `make` permettant d'obtenir le fichier `rcalc.o` dans le répertoire de travail.
2. Vérifier le bon fonctionnement de cette règle par la commande

```
make rcalc.o
```

```
// rcalc.C

#include "rcalc.h"

int main(void) {
     RPile pile;
     Lexeme lexeme;
     double x, y, z, t;
     double (*f)(double);

     do {
     ⋮
     } while(lexeme.type != FIN);
     return 0;
}
```

9.1.4 Edition de liens

1. Ecrire la règle `make` permettant d'obtenir le fichier exécutable `rcalc` dans le répertoire de travail.

2. Vérifier le bon fonctionnement de cette règle par la commande

   `make rcalc`

3. Faire en sorte que le fichier `rcalc` soit obtenu simplement par la commande

   `make`

9.2 Réalisation de la calculette

9.2.1 Traitement des nombres

1. Compléter la boucle d'évaluation de la calculette (voir fichier `rcalc.C`) pour traiter la lecture d'un nombre.

   ```
   ?  1.2
   = 1.2
   ```

2. Vérifier le bon fonctionnement de ce traitement par la commande `make ; rcalc` .

9.2.2 Traitement des fonctions

1. Compléter la boucle d'évaluation de la calculette (voir fichier `rcalc.C`) pour traiter la lecture d'une fonction.

   ```
   ?  6.8
   = 6.8
   ?  sin
   sin(6.8) = 0.494113
   ```

2. Vérifier le bon fonctionnement de ce traitement par la commande `make ; rcalc` .

9.2.3 Traitement des opérateurs

1. Compléter la boucle d'évaluation de la calculette (voir fichier `rcalc.C`) pour traiter la lecture d'un opérateur.

```
?  3.14159
= 3.14159
?  2
= 2
?  *
2 * 3.14159 = 6.28318
```

2. Vérifier le bon fonctionnement de ce traitement par la commande `make ; rcalc` .

9.2.4 Autres traitements

1. Compléter la boucle d'évaluation de la calculette (voir fichier `rcalc.C`) pour traiter la lecture d'un lexème inconnu .

2. Vérifier le bon fonctionnement de ce traitement par la commande `make ; rcalc` .

T.P. 10

Les opérateurs en C++

10.1 Les chaînes de caractères

Définir la classe `String` qui est une représentation des chaînes de caractères.

```
class String {
public:
// Allocators
      String(void);
      String(char c, size_t n = 1);
      String(const char* str);
      String(const String& str);
      String& operator=(const char* str);
      String& operator=(const String& str);
      ~String(void);
// Inspectors
      bool empty(void) const;
      size_t size(void) const;
      operator const char*(void) const;
      const char& operator[](size_t n) const;
      bool operator==(const String& str) const;
      bool operator!=(const String& str) const;
      bool operator<(const String& str) const;
// Input/output
      friend ostream& operator<<(ostream& stream, const String& str);
      friend istream& operator>>(istream& stream, String& str);
// Modifiers
      char& operator[](size_t n);
      void setSize(size_t n);
      String& operator+=(const String& str);
      friend String operator+(const String& str1, const String& str2);
private:
// Attributes
      size_t _size;
      char* _str;
};
```

10.2 Les nombres rationnels

Définir la classe `Rational` qui représente les nombres rationnels.

```
class Rational {
public:
// Allocators
     Rational(void);
     Rational(int numerator, int denominator = 1);
     Rational(const Rational& r);
     ~Rational(void);
// Inspectors
     int numerator(void) const;
     int denominator(void) const;
     bool operator==(const Rational& r) const;
     bool operator!=(const Rational& r) const;
     friend bool operator<(const Rational& r1, const Rational& r2);
     friend bool operator>(const Rational& r1, const Rational& r2);
     friend bool operator<=(const Rational& r1, const Rational& r2);
     friend bool operator>=(const Rational& r1, const Rational& r2);
// Modifiers
     void setNumerator(int numerator);
     void setDenominator(int denominator);
     void simplify(void);
     Rational& operator=(const Rational& r);
     Rational& operator+=(const Rational& r);
     Rational& operator-=(const Rational& r);
     Rational& operator*=(const Rational& r);
     Rational& operator/=(const Rational& r);
     operator double(void);
// Arithmetic
     Rational& operator+(void);
     Rational& operator-(void);
     friend Rational operator+(const Rational& r1, const Rational& r2);
     friend Rational operator-(const Rational& r1, const Rational& r2);
     friend Rational operator*(const Rational& r1, const Rational& r2);
     friend Rational operator/(const Rational& r1, const Rational& r2);
// Streams operations
     friend ostream& operator<<(ostream& stream, const Rational& r);
     friend istream& operator>>(istream& stream, Rational& r);
private:
     int _numerator;
     int _denominator;
};
```

T.P. 11

La généricité en C++

11.1 Fonctions génériques

11.1.1 Prototypes

Déclarer les prototypes des fonctions génériques suivantes :

1. la fonction `max` qui détermine le maximum de deux nombres;
2. la fonction `max` qui détermine le maximum d'un tableau de n nombres.

11.1.2 Définitions

1. Définir les deux fonctions `max` dont les prototypes ont été déclarés dans la question précédente.
2. Quelle contrainte doit–on imposer sur le type générique des fonctions `max` ?

11.1.3 Instanciations

1. Ecrire un programme qui calcule le maximum de 2 entiers et le maximum d'un tableau de réels.
2. Expliquer ce que doit faire le compilateur pour compiler ce programme.

11.1.4 Compilation

La procédure de compilation de fonctions et de classes génériques dépend encore fortement du compilateur utilisé.
Nous considérons ici le cas du compilateur `IBM AIX XL C++ Compiler/6000`.

1. Ecrire les 3 fichiers suivants :
   (a) le fichier `tp/include/tp10.1.h` contient les prototypes de la section 11.1.1;

(b) le fichier `tp/include/tp10.1.c` (`.c` et non `.C`) contient les définitions de la section 11.1.2;

(c) le fichier `tp/src/tp10/tp10.1.C` (`.C` et non `.c`) contient le programme de la section 11.1.3.

2. Vérifier que la commande

```
xlC -I../../include tp10.1.C
```

suffit à compiler le programme du fichier `tp10.1.C` .

3. Vérifier que le compilateur a créé un nouveau répertoire (`tempinc`) et consulter les différents fichiers de ce répertoire.

11.2 Classes génériques

On reprend l'exemple de la classe `Point` du T.P. 7 mais ici, un point du plan sera considéré comme un doublet de 2 nombres (soit entiers, soit réels).
On pourra extraire l'abscisse x et l'ordonnée y d'un point. Enfin, un point pourra être déplacé de $(\delta x, \delta y)$, et affiché sous la forme (x, y)

11.2.1 Interface

1. Définir formellement le TAD `Point` en s'inspirant de la description informelle précédente.

2. Définir l'interface C++ (fichier `gpoint.h`) de la classe générique `GPoint` correspondant au TAD `Point` .

11.2.2 Implémentation

1. Définir les fonctions prévues dans l'interface de la classe générique `GPoint` (fichier `gpoint.C`).

2. Que fait réellement le compilateur lors de la compilation du fichier `gpoint.C` ?

```
xlC -c -I../../include gpoint.C
```

11.2.3 Instanciations

1. Ecrire un programme (fichier `tp10.2.C`) qui teste le bon fonctionnement de la classe `GPoint` .

2. Inclure le programme précédent dans le fichier `gpoint.C` de telle manière qu'il ne soit compilé que si la macro `TESTGPOINT` est définie.

T.P. 12

Des piles et des tableaux

12.1 La classe `SPile`

1. Définir (`spile.C`) la classe générique `template <class T,int N> class SPile;` qui correspond à la classe `TPile` précédente, mais où le type des éléments et la taille maximum du tableau des éléments sont maintenant génériques.

   L'interface de la classe `SPile` (`Pile` Statique) est donnée ci–dessous (`spile.h`).

   Remarque : la compilation d'une classe générique nécessite un traitement particulier qui est décrit par la règle "`make`" suivante (fichier `Makefile.tp3`):

   ```
   .C.o:  $*.h $*.C
          cp $*.C $(INCDIR)/$*.c
          $(CCC) $(CCFLAGS) $(CPPFLAGS) $*.C
   ```

   On examinera les différents fichiers créés lors de la compilation d'une classe générique.

2. Ecrire un programme (`scalc.C`) qui utilise une `SPile` pour réaliser une calculette à pile (voir T.P. 9 page 55).

12.2 La classe `DPile`

1. Définir (`dpile.C`) la classe générique `template <class TYPE> DPile` qui correspond à une pile dont les éléments sont stockés dans un tableau dynamique.

   L'interface de la classe `DPile` (`Pile` Dynamique) est donnée en annexe (`dpile.h`).

2. Ecrire un programme (`dcalc.C`) qui utilise une `DPile` pour réaliser une calculette à pile (voir T.P. 9 page 55).

12.3 Interface de la classe SPile

```
// spile.h

#ifndef SPILE_H
#define SPILE_H

#include <iostream.h>

#ifdef PILE
#undef PILE
#endif

#define PILE SPile

template <class TYPE, int MAX>
class PILE {
public:
     #include "spile.public.h"
private:
     #include "spile.private.h"
};

#endif
```

12.3.1 Interface publique

```
// spile.public.h

     // constructeurs/destructeur
     PILE(void);
     PILE(const PILE<TYPE,MAX>&);
     ~PILE(void);

     // operateurs
     PILE<TYPE,MAX>& operator=(const PILE<TYPE,MAX>&);
     int operator==(const PILE<TYPE,MAX>&) const ;
     int operator!=(const PILE<TYPE,MAX>&) const ;

     // TAD
     int vide(void) const ;
     int pleine(void) const ;
     TYPE sommet(void) const ;
     PILE<TYPE,MAX>& operator>>(TYPE&);                          // depiler
     TYPE depiler(void);
     PILE<TYPE,MAX>& operator<<(const TYPE&);                    // empiler
     void empiler(const TYPE&);

     // entrees/sorties
     friend ostream& operator<<(ostream&, const PILE<TYPE,MAX>&);
```

12.3.2 Interface privée

```
// spile.private.h

     // attributs
     int _taille;
     TYPE _pile[MAX];

     // utilitaires
     void _copier(const PILE<TYPE,MAX>&);
     void _detruire(void);
```

12.4 Interface de la classe DPile

```
// dpile.h

#ifndef DPILE_H
#define DPILE_H

#include <iostream.h>

#ifdef PILE
#undef PILE
#endif

#define PILE DPile

template <class TYPE>
class PILE {
public:
     #include "dpile.public.h"
private:
     #include "dpile.private.h
};

#endif
```

12.4.1 Interface privée

```
// dpile.private.h

     // attributs
     int _taille;
     int _capacite;
     TYPE* _pile;

     // utilitaires
     void _copier(const PILE<TYPE>&);
     void _detruire(void);
```

12.4.2 Interface publique

```
// dpile.public.h

     // constructeurs/destructeur
     PILE(void);
     PILE(const PILE<TYPE>&);
     ~PILE(void);

     // operateurs
     PILE<TYPE>& operator=(const PILE<TYPE>&);
     int operator==(const PILE<TYPE>&) const ;
     int operator!=(const PILE<TYPE>&) const ;

     // TAD
     int vide(void) const ;
     int pleine(void) const ;
     TYPE sommet(void) const ;
     PILE<TYPE>& operator>>(TYPE&);                              // depiler
     TYPE depiler(void);
     PILE<TYPE>& operator<<(const TYPE&);                        // empiler
     void empiler(const TYPE&);

     // entrees/sorties
     friend ostream& operator<<(ostream&, const PILE<TYPE>&);
```

T.P. 13

Des piles et des listes

Les éléments d'une `SPile` ou d'une `DPile` (voir T.P. 12 page 63) sont stockés dans un tableau statique `TYPE _pile[MAX]` ou dans un tableau dynamique `TYPE* _pile`.
Dans ce T.P., on étudiera une autre manière de stocker les éléments d'une pile : dans une liste chaînée.

13.1 La classe `LPile`

1. Définir (`lpile.C`) la classe générique `template <class TYPE> LPile` qui correspond à une pile dont les éléments sont stockés dans une liste chaînée.

   On utilisera la structure générique `template <class TYPE>`
   ```
   struct Noeud {
        TYPE _valeur;
        Noeud<TYPE>* _suivant;
   };
   ```
   pour représenter un élément d'une liste chaînée (voir `lpile.h`).

2. Ecrire un programme (`lcalc.C`) qui utilise une `LPile` pour réaliser une calculette à pile.

13.2 Interface de la classe `LPile`

```
// lpile.h

#ifndef LPILE_H
#define LPILE_H

#include <iostream.h>

#ifdef PILE
#undef PILE
#endif

#define PILE LPile

template <class TYPE>
struct Noeud {
      TYPE _valeur;
      Noeud<TYPE>* _suivant;
};

template <class TYPE>
class PILE {
public:
      #include "dpile.public.h"
private:
      #include "lpile.private.h
};

#endif
```

13.2.1 Interface publique

L'interface publique d'une `LPile` est identique à celle d'une `DPile` (voir T.P. 12 page 63).

13.2.2 Interface privée

```
// lpile.private.h

     // attributs
     int _taille;
     Noeud<TYPE>* _pile;

     // utilitaires
     void _copier(const PILE<TYPE>&);
     void _detruire(void);
```

T.P. 14

L'héritage en C++

L'héritage est un des principes de base de la programmation par objets. Il permet de structurer une application aussi bien d'un point de vue conception que d'un point de vue implémentation.

14.1 Modes d'héritage

Jusqu'à présent, une classe C++ définissait 2 niveaux de protection (`private` et `public`). L'héritage introduit un niveau intermédiaire (`protected`); ce niveau qualifie des membres qui ne sont accessibles que par la classe elle–même et par ses classes dérivées. Le tableau ci–dessous résume les modes d'héritage entre une classe dérivée et sa classe de base.

`class Base`	`class Derivee : private Base`	`class Derivee : protected Base`	`class Derivee : public Base`
`private:`	... *inaccessible* ...	... *inaccessible* ...	... *inaccessible* ...
`protected:`	..... *privé*	..... *protégé*	..... *protégé* .....
`public:`	..... *privé*	..... *protégé*	..... *public* .....

On considère la classe `Base` définie en annexe 14.4.1 page 72. Cette classe définie simplement (!?) un tableau `_c` de `_b` entiers tels que `_c[i] = i + _a` pour `0 <= i < _b`. L'opérateur `operator()` a été redéfini pour calculer la somme des éléments du tableau `_c`.

1. Expliquer pourquoi le fait que _a soit un attribut public de la classe `Base` n'est pas un choix judicieux du point de vue de l'utilisateur de cette classe.

2. On considère les classes dérivées `D1` (section 14.4.2 page 74), `D2` (section 14.4.3 page 75) et `D3` (section 14.4.4 page 75).

   Définir les constructeurs et destructeurs de ces classes.

14.2 Liaisons dynamiques et classes abstraites

On considère le fichier `dyna.C` de l'annexe 14.4.5 page 76.

1. L'exécution de ce programme conduit au résultat ci–dessous.

```
$ a.out
1 2 3 2
A1::~A1() A2::~A2() A1::~A1() A3::~A3()
A1::~A1() A4::~A4() A2::~A2() A1::~A1()
```

   Comment modifier les déclarations des classes A_i pour que l'exécution du même programme conduise au résultat suivant ?

```
$ a.out
1 2 3 8
A1::~A1() A2::~A2() A1::~A1() A3::~A3()
A1::~A1() A4::~A4() A2::~A2() A1::~A1()
```

2. On considère la classe abstraite `A` définie ci–dessous.

```
class A {
public:
      ~A(void) { cerr << "A::~A()" << endl; }
      virtual int operator()(void) { return f(); };
      virtual int f(void) = 0;
};
```

   On suppose maintenant que la classe `A1` du fichier `dyna.C` hérite publiquement de la classe `A`. On considère la fonction `main` ci–dessous.

```
void main(void) {
      A** t = new A*[4];
      t[0] = new A1; t[1] = new A2; t[2] = new A3; t[3] = new A4;
      for(int i = 0; i < 4; i++) cout << (*t[i])() << endl;
      for(int j = 0; j < 4; j++) delete t[j];
      delete t;
}
```

   L'exécution de ce programme conduit maintemant au résultat suivant.

```
$ a.out
1 2 3 8
A::~A() A::~A() A::~A() A::~A()
```

   Comment modifier la déclaration de la classe A pour que l'exécution du même programme conduise au résultat suivant ?

```
$ a.out
1 2 3 8
A1::~A1() A::~A()
A2::~A2() A1::~A1() A::~A()
A3::~A3() A1::~A1() A::~A()
A4::~A4() A2::~A2() A1::~A1() A::~A()
```

14.3 Les tours de Hanoï

Une tour de Hanï est une sorte de pile dans laquelle les éléments ne peuvent être empilés que dans un ordre décroissant.

1. Définir la classe `Hanoi<N>` comme une classe dérivée de la classe `SPile<int,N>` (voir section 14.4.6 page 77).

```
template <int N> class Hanoi :  public SPile<int,N>;
```

2. Vérifier que le programme ci–dessous fonctionne correctement.

```
#include "hanoi.h"

template <int N>
void hanoi(int n, Hanoi<N>& d, Hanoi<N>& inter, Hanoi<N>& a) {
      assert(n <= N);
      if(n > 0) {
            hanoi(n-1,d,a,inter);
            a.empiler(d.depiler());
            hanoi(n-1,inter,d,a);
      }
}

int main(void) {
      Hanoi<15> depart, arrivee, intermediaire;
      for(int i = 15; i > 0; i--) depart.empiler(i);
      cout << depart << ' ' << intermediaire << ' ' << arrivee << endl;
      hanoi(15, depart, intermediaire, arrivee);
      cout << depart << ' ' << intermediaire << ' ' << arrivee << endl;
      return 0;
}
```

14.4 Annexes

14.4.1 La classe Base

```
// base.h

#ifndef BASE_H
#define BASE_H

#include <iostream.h>

class Base {
public:
      int _a;
      Base(int a = 1, int b = 2);
      Base(const Base& b);
      Base& operator=(const Base& b);
       Base(void);
      const int* getc(void) const;
      int getb(void) const;
      void setb(int b);
      int operator()(void) const;
      friend int operator==(const Base& b1, const Base& b2);
      friend int operator!=(const Base& b1, const Base& b2);
      friend ostream& operator<<(ostream& s, const Base& b);
protected:
      int _b;
private:
      int* _c;
      void _destroy(void);
      void _copy(const Base& b);
};

#endif
```

```
// base.C (1/3)

#include "base.h"

Base::Base(int a, int b) :  _a(a), _b(b), _c(new int[b]) {
      cerr << "Base::Base(a,b)" << endl;
      for(int i = 0; i < _b; i++) _c[i] = _a + i;
}

Base::Base(const Base& b) :  _a(b._a), _b(b._b), _c(new int[b._b]) {
      cerr << "Base::Base(base)" << endl;
      _copy(b);
}
```

```
// base.C (2/3)

Base& Base::operator=(const Base& b) {
      cerr << "Base::operator=(base)" << endl;
      if(this != &b) {
            _destroy();
            _copy(b);
      }
      return *this;
}

Base::~Base(void) {
      cerr << "Base:: Base()" << endl;
      _destroy();
}

const int* Base::getc(void) const { return _c; }

int Base::getb(void) const { return _b; }

void Base::setb(int b) {
      cerr << "Base::setb(b)" << endl;
      if(b > _b) {
            int* newc = new int[b];
            for(int i = 0; i < b; i++) newc[i] = _a;
            _destroy();
            _c = newc;
      }
      _b = b;
}

int Base::operator()(void) const {
      cerr << "Base::operator()" << endl;
      int somme = 0;
      if(_b > 0) {
            somme = _c[0];
            for(int i = 1; i < _b; i++) somme += _c[i];
      }
      return somme;
}

int operator==(const Base& b1, const Base& b2) {
      cerr << "b1 == b2" << endl;
      int retour = 1;
      if(b1._a != b2._a) retour = 0;
      if(b1._b != b2._b) retour = 0;
      for(int i = 0; i < b1._b; i++)
            if(b1._c[i] != b2._c[i]) retour = 0;
      return retour;
}
```

```
// base.C (3/3)

int operator!=(const Base& b1, const Base& b2) { return !(b1 == b2); }

ostream& operator<<(ostream& s, const Base& b) {
      cerr << "ostream << base" << endl;
      s << '(';
      if(b._b > 0) {
            for(int i = 0; i < b._b - 1; i++) s << b._c[i] << ',';
            s << b._c[b._b - 1];
      }
      return s << ')';
}

void Base::_destroy(void) { if(_c) delete [] _c; }

void Base::_copy(const Base& b) {
      _a = b._a; _b = b._b;
      _c = new int[_b];
      for(int i = 0; i < _b; i++) _c[i] = b._c[i];
}
```

14.4.2 La classe D1

```
// d1.h

#ifndef D1_H
#define D1_H

#include "base.h"

class D1 :  public Base {
private:
      int _d;
public:
      D1(int a = 1 , int b = 2, int d = 4);
      D1(const D1& d);
      D1& operator=(const D1& d);
      ~D1(void);
      int getd(void) const { return _d; }
      void setd(int d) { _d = d; }
      int operator()(void) {
            cerr << "D1::operator()" << endl;
            return Base::operator()() * _d;
      }
};
```

14.4.3 La classe D2

```
// d2.h

class D2 :  protected Base {
private:
      int _d;
public:
      D2(int a = 1 , int b = 2, int d = 4);
      D2(const D2& d);
      D2& operator=(const D2& d);
      ~D2(void);
      int getd(void) const { return _d; }
      void setd(int d) { _d = d; }
      int operator()(void) {
            cerr << "D2::operator()" << endl;
            return Base::operator()() * _d;
      }
};
```

14.4.4 La classe D3

```
// d3.h

class D3 :  private Base {
private:
      int _d;
public:
      D3(int a = 1 , int b = 2, int d = 4);
      D3(const D3& d);
      D3& operator=(const D3& d);
      ~D3(void);
      int getd(void) const { return _d; }
      void setd(int d) { _d = d; }
      int operator()(void) {
            cerr << "D3::operator()" << endl;
            return Base::operator()() * _d;
      }
};
```

14.4.5 Le fichier `dyna.C`

```
// dyna.C
#include <iostream.h>

class A1 {
private:
      int* _i;
public:
      A1(int i = 1) :  _i(new int(i)) {}       virtual ~A1(void) {
            cerr << "A1::~A1()" << endl;
            if(_i) delete _i;
      }
      int f(void) { return *_i; }
};

class A2 :  public A1 {
public:
      A2(int i = 2) :  A1(i) {}
      virtual ~A2(void) { cerr << "A2::~A2()" << endl; }
};

class A3 :  public A1 {
public:
      A3(int i = 3) :  A1(i) {}
      virtual ~A3(void) { cerr << "A3::~A3()" << endl; }
};

class A4 :  public A2 {
private:
      int* _i;
public:
      A4(int i = 4) :  A2(), _i(new int(i)) {}
      virtual ~A4(void) {
            cerr << "A4::~A4()" << endl;
            if(_i) delete _i;
      }
      int f(void) { return A2::f() * (*_i); }
};

void main(void) {
      A1* t[4];
      t[0] = new A1; t[1] = new A2; t[2] = new A3; t[3] = new A4;
      for(int i = 0; i < 4; i++) cout << t[i]->f() << endl;
      for(int j = 0; j < 4; j++) delete t[j];
      delete t;
}
```

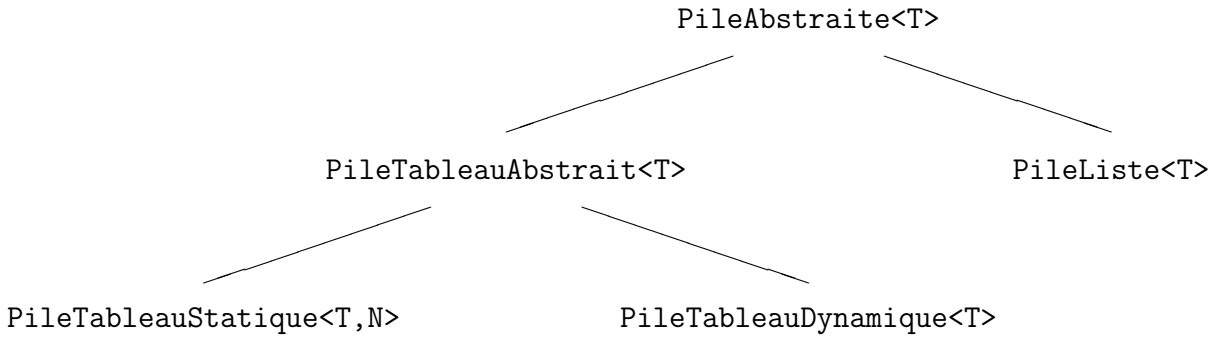
14.4.6 La classe SPile

```
// spile.h
#ifndef SPILE_H
#define SPILE_H

#include <assert.h>
#include <iostream.h>

template <class T, int N>
class SPile {
protected:
      int _taille;
      T _t[N];
      virtual void _detruire(void) {}
      virtual void _copier(const SPile<T,N>& p) {
            for(int i = 0; i < _taille; i++) _t[i] = p._t[i];
      }
public:
      SPile(void) :  _taille(0) {}
      SPile(const SPile<T,N>& p) :  _taille(p._taille) { _copier(p); }
      SPile<T,N>& operator=(const SPile<T,N>& p) {
            if(this != &p) {
                  _detruire();
                  _copier(p);
            }
            return *this;
      }       virtual ~SPile(void) { _detruire(); }
      virtual int estVide(void) const { return _taille == 0; }
      virtual int estPleine(void) const { return _taille == N; }
      virtual T sommet(void) const {
            assert(!estVide());
            return _t[_taille - 1];
      }
      virtual T depiler(void) {
            assert(!estVide());
            return _t[_taille-- - 1];
      }
      virtual void empiler(const T& t) {
            assert(!estPleine());
            _t[_taille++] = t;
      }
};

#endif
```

T.P. 15

Une famille de piles

Nous considérons ici une famille de piles hiérarchisées selon l'arborescence suivante.



- La classe `PileAbstraite<T>` factorise les services que doit rendre une pile quelle que soit son implémentation : c'est une classe abstraite.
- La classe `PileTableauAbstrait<T>` est également une classe abstraite qui représente les piles implémentées sous la forme d'un tableau. A ce niveau de l'arborescence, le choix n'est pas encore fait entre une représentation sous forme de tableau statique ou sous forme de tableau dynamique. Ce sont les classes concrètes `PileTableauStatique<T,N>` et `PileTableauDynamique<T>` qui effectuent ce choix.
- La classe `PileTableauStatique<T,N>` représente les piles implémentées à l'aide d'un tableau statique de capacité `N`.
- La classe `PileTableauDynamique<T>` représente les piles implémentées à l'aide d'un tableau dynamique.
- La classe `PileListe<T>` représente les piles implémentées à l'aide d'une liste chaînée.

1. Définir les interfaces de ces 5 classes génériques dans un même fichier `pile.h` .
2. Définir les implémentations de ces 5 classes dans un même fichier `pile.C` .
3. Montrer sur un exemple simple l'intérêt de l'existence de la classe abstraite `PileAbstraite<T>`.

T.P. 16

Les difficultés du C++

16.1 Constructeurs

1. On considère les classes `A` et `B` définies ci–dessous.

```
class A {
public:
      class B {
      public:
            int attribut;
      };
};
```

   Déclarer un objet `b` de la classe `B`.

2. On considère les classes `A` et `B` ci–dessous.

```
class A {
private:
      int _n;
public:
      A(int n = 0) { _n = n; cout << "A::A(int)" << endl; }
      A& operator=(int n) {
            _n = n;
            cout << "A::operator=(int)" << endl;
            return *this;
      }
      friend ostream& operator<<(ostream& s, const A& a)
                                                { return s << a._n << endl; }
};

class B {
private:
      A _a;
public:
      B(void) { _a = 5; }
      friend ostream& operator<<(ostream& s, const B& b)
                                                { return s << b._a << endl; }
};
```

(a) Qu'affiche le programme suivant `void main(void) { B b; cout << b << endl;}` ?

(b) Comment peut–on optimiser la classe `B`, sans la modifier fondamentalement, et sans modifier la classe `A`, pour que l'affichage du programme ci–dessus soit simplifié ?

16.2 Destructeurs

1. Pourquoi déclarer un destructeur comme une fonction virtuelle (`virtual ~Classe(void);`) ?

2. Pourquoi ne faut–il jamais déclarer un destructeur comme une fonction virtuelle pure (`virtual ~Classe(void) = 0;`) ?

16.3 Opérateurs

On considère la classe générique `Tableau<TYPE>` dont l'interface est donnée en Annexe 16.6.1 page 85.

1. La définition de l'opérateur d'affectation ressemble au code suivant :

```
Tableau<TYPE>& Tableau<TYPE>::operator=(const Tableau<TYPE>& t) {
     if(this != &t) {
          _detruire();
          _copier(t);
     }
     return *this;
}
```

   Quel est le rôle du test `if(this != &t)` ?

2. Pourquoi l'opérateur d'indexation (`operator[]`) est–il défini de deux manières différentes ?

3. Pourquoi les deux opérateurs d'injection (`<<`) retournent–ils une référence ?

4. Est–il nécessaire de déclarer `friend` l'opérateur d'injection dans un flot ? Pourquoi ?

16.4 Généricité

1. On considère la classe générique `A<T>` de l'annexe 16.6.2 page 86 (fichier `a.h`), la classe `B` de l'annexe 16.6.2 page 86 (fichier `b.h`), et le programme principal de l'annexe 16.6.2 page 86 (fichier `ab.C`).

   Dans ce programme, l'utilisation des classes `A<T>` et `B` n'est pas correcte. Une même erreur provoque d'ailleurs des comportements différents selon le compilateur !

- Le compilateur `xlC` d'IBM provoque une erreur de compilation dans le fichier `a.h`
- Une erreur d'exécution apparaît lorsque ce programme a été compilé "avec succès" par le compilateur `Borland C++`.

(a) Quelle est l'erreur commise ?

(b) Dans le cas du compilateur `xlC` d'IBM, sur quelle ligne du fichier `a.h` l'erreur est–elle indiquée ?

2. On considère les classes C et `D<T>` ci–dessous.

```
class C {
public:
     int c;
};

template <class T>
class D {
public:
     D(const T& t = 0) {_t = t;}
     const T& t(void) const {return _t;}
private:
     int _t;
};

void main (void) {
     D<int> d;                                          // declaration 1
     D<C> dc;                                           // declaration 2
     D<D<int> > dd;                                     // declaration 3
}
```

L'une des 3 déclarations de la fonction `main` provoque une erreur de compilation. Laquelle et pourquoi ?

16.5 Héritage

1. On considère les classes C et D de l'annexe 16.6.3 page 87.
   Pour chacune des instructions suivantes, indiquer si elle est autorisée et justifier, selon le cas, l'erreur ou l'affichage obtenu.

   (a) `C* c = new D; c->f();`

   (b) `C* c = new D; c->f(2);`

   (c) `C* c = new D; c->f("bonjour");`

   (d) `D d; d.f();`

   (e) `D d; d.f(2);`

(f) `D d; d.f("bonjour");`

2. On considère les classes `E` et `F` de l'annexe 16.6.4 page 87.

   Indiquer, pour chacune des instructions de la fonction `main()` du fichier `ef.C`, les affichages obtenus.

3. On considère les classes `E` et `F` ci–dessous.

```
class E {
public:
     int f(void) = 0;
};

class F : public E {
public:
     F(int n = 0) { _f = n; }
     int f(void) { return _f; }
private:
     int _f;
};
```

   La compilation de ces classes provoque une erreur. Laquelle et pourquoi ?

4. On considère les classes `Figure` et `Cercle` ci–dessous.

```
class Figure {
public:
     virtual void afficher(void) = 0;
     Figure(void) { this->afficher(); }
     virtual ~Figure(void) {}
};

class Cercle :  public Figure {
public:
     virtual void afficher(void) { cout << "Cercle" << endl; }
     Cercle(void) :  Figure() {}
     virtual ~Cercle(void) {}
};
```

   Le programme suivant provoque une erreur à l'exécution. Pourquoi ?

```
void main(void) {
     Cercle* c = new Cercle;
     c->afficher();
}
```

16.6 Annexes

16.6.1 Fichier `tableau.h`

```
#ifndef TABLEAU_H
#define TABLEAU_H

template <class TYPE>
class Tableau {
public:
// constructeurs/destructeur
      Tableau(void);
      Tableau(const Tableau<TYPE>& t);
      virtual ~Tableau(void);
// operateurs
      Tableau<TYPE>& operator=(const Tableau<TYPE>& t);
      const TYPE& operator[](int n) const;
      TYPE& operator[](int n);
      int operator==(const Tableau<TYPE>& t) const;
      int operator!=(const Tableau<TYPE>& t) const;
      Tableau<TYPE>& operator+=(const Tableau<TYPE>& t);
      Tableau<TYPE> operator+(const Tableau<TYPE>& t);
      Tableau<TYPE>& operator<<(const TYPE& elem);
      friend ostream& operator<<(ostream& s, const Tableau<TYPE>& t);
// TAD
      int taille(void) const;
      int capacite(void) const;
      int vide(void) const;
      int plein(void) const;
      void ajouter(const TYPE& elem);              // ajouter en fin du tableau
      TYPE enlever(int n);
      void vider(void);
private:
      #include "tableau.private.h"
};

#endif
```

16.6.2 Classes A<T> et B

Classe générique A<T>

```
// a.h

template <class T>
class A {
public:
      A(const T& t = T()) {_t = t;}
      A(const A<T>& a) {_t = a._t;}
      A<T>& operator=(const A<T>& a) {_t = a._t; return *this;}
      int operator==(const A<T>& a) const {return (_t == a._t);}
private:
      T _t;
};
```

Classe B

```
// b.h

class B {
public:
      B(int i = 0) {_i = i;}
      B(const B& b) {_i = b._i;}
      B& operator=(const B& b) {_i = b._i; return *this;}
private:
      int _i;
};
```

Utilisation des classes A<T> et B

```
// ab.C

#include <iostream.h>
#include "a.h"
#include "b.h"

void main(void) {
      B b(2);
      A<B> a1(b), a2(a1);
      if(a1 == a2) cout << "a1 == a2" << endl;
      else cout << "a1 != a2" << endl;
}
```

16.6.3 Classes C et D

```
// cd.h

#include <iostream.h>

class C {
public:
     virtual void f(void) {cout << "classe C" << endl;}
};

class D : public C {
public:
     void f(int i) {cout << "classe D : " << i << endl;}
     void f(const char* s) {cout << "classe D : " << s << endl;}
};
```

16.6.4 Classes E et F

```
// ef.C

#include <iostream.h>

class E {
public:
     E(void) {fct();}
     virtual ~E(void) {cout << "E::~E()" << endl;}
     virtual void fct(void) {cout << "E::E()" << endl;}
};

class F : public E {
public:
     F(void) {fct();}
     virtual ~F(void) {cout << "F::~F()" << endl;}
     void fct(void) {cout << "F::F()" << endl;}
};

void main(void) {
     E* e1 = new E; E* e2 = new F; F* f1 = new F;
     delete e1; delete e2; delete f1;
}
```

T.P. 17

Standard Template Library

17.1 Itérateurs

17.1.1 Définition et utilisation

1. Qu'est–ce qu'un itérateur dans STL ?

2. Quels sont les principaux types d'itérateurs ?

3. Comment accède–t–on aux itérateurs d'une liste (`list<T>`) ? plus généralement d'un conteneur ?

17.1.2 Classe `IstreamIterator<T>`

Définir la classe `IstreamIterator<T>` (voir interface en annexe 17.5.1 page 92) qui associe un itérateur à un flot d'entrée.

```
// exemple d'utilisation
// on passe le nom d'un fichier d'entiers sur la ligne de commande
// et on copie les entiers dans un vecteur
char fileName[80];
::strcpy(fileName,argv[1]);
ifstream input(fileName);
vector<int> v;
copy(IstreamIterator<int>(input),IstreamIterator<int>(),v.begin());
```

17.2 Séquences

17.2.1 `vector<T>`, `deque<T>` et `list<T>`

1. Préciser les conditions optimales d'utilisation des types `vector<T>`, `deque<T>` et `list<T>`.

2. Ecrire une fonction `void cp(vector<T> v,list<T> l)` qui copie les éléments du vecteur `v` dans la liste `l`.

3. Généraliser la fonction précédente pour copier les éléments d'une séquence (`vector<T>`, `deque<T>` ou `list<T>`) dans une autre séquence (`vector<T>`, `deque<T>` ou `list<T>`).

17.2.2 Encore une pile !

1. Définir une classe `VPile<T>` à l'aide de la classe `vector<T>`.
2. Généraliser la classe `VPile<T>` pour qu'elle puisse utiliser une séquence quelconque (`vector<T>`, `deque<T>` ou `list<T>`).
3. Qu'est–ce qu'un adaptateur dans STL ?

17.3 Tables associatives

17.3.1 Ensembles et tas

1. Quelles sont les différences entre un `set` et un `multiset` ?
2. On considère le fichier `words` qui contient des mots anglais (1 mot par ligne).
   (a) Utiliser un `istream_iterator<String,ptrdiff_t>` pour copier les mots du fichier `words` dans un `vector<String>` et dans un `set<String,less<String>>`.
   (b) Combien y a–t–il de mots dans le fichier `words` ?
   (c) Comparer les temps de recherche du mot `timeout` entre le `vector` et le `set`. On effectuera 1000 recherches successives pour comparer les temps.

17.3.2 Dictionnaires et encyclopédies

1. Quelles sont les différences entre un `map` et un `multimap` ?
2. On considère le fichier `words` qui contient des mots anglais (1 mot par ligne).
   (a) Utiliser un `istream_iterator<String,ptrdiff_t>` pour insérer les mots du fichier `words` dans un `multimap<int,String,less<int>>` dont la clé est le nombre de lettre du mot.
   (b) Afficher pour chaque nombre de lettres, le nombre de mots correspondant.
   (c) Quels sont les mots les plus longs ?

17.4 Anagrammes

On considère le fichier `words` qui contient des mots anglais (1 mot par ligne). Ce fichier sera le dictionnaire de référence pour trouver des anagrammes.
Un anagramme est un mot obtenu par permutation des lettres d'un autre mot (exemples : chine, niche, chien).

17.4.1 Trouver les anagrammes d'un mot

Ecrire un programme qui affiche tous les anagrammes d'un mot à partir d'un dictionnaire de référence.

```
$ anagramme words
taille du dictionnaire :  23788 mots
Entrer un mot :  gonar
> argon groan organ
...
$
```

1. Utiliser un `istream_iterator<String,ptrdiff_t>` pour copier les mots du fichier `words` dans un `vector<String>` (`vector<String> dico;`).
2. Afficher la taille du dictionnaire.
3. Lire un mot au clavier (`cin`) et le stocker dans un `vector<char>` (`vector<char> word;`).
4. Utiliser les fonctions STL `sort`, `binary_search` et `next_permutation` pour trouver et afficher les anagrammes du mot entré au clavier.

17.4.2 Dictionnaire d'anagrammes

Ecrire un programme qui affiche tous les anagrammes d'un mot à partir d'un dictionnaire d'anagrammes.

1. Utiliser un `istream_iterator<String,ptrdiff_t>` pour copier les mots du fichier `words` dans un `multimap<String,String,less<String>>` dont la clé est le mot dont les lettres ont été triées par ordre alphabétique. Ainsi, les mots `argon`, `groan` et `organ` se retrouvent sous la même clé `agnor`.
2. Utiliser le `multimap` pour trouver tous les anagrammes d'un mot.

17.5 Annexes

17.5.1 La classe IstreamIterator<T>

```
// IstreamIterator.h

template <class T>
class IstreamIterator :  public input_iterator<T,ptrdiff_t> {
public:
      IstreamIterator(void);
      IstreamIterator(istream& inputStream);
      const T& operator*(void) const;
      IstreamIterator& operator++(void);
      IstreamIterator operator++(int);
      bool operator==(const IstreamIterator& iterator) const;
      bool operator!=(const IstreamIterator& iterator) const;
protected:
      istream* _stream;
      T _value;
      bool _end;
      void _read(void);
};
```

T.P. 18

La vie artificielle des “ bugs ”

18.1 Cahier des charges

Des “ bugs ” vivent dans un plat contenant des bactéries, qui sont la nourriture de base des “ bugs ”.
Les “ bugs ” se déplacent de façon aléatoire dans le plat, mangeant les bactéries qu'ils rencontrent au cours de leurs pérégrinations.
Les bactéries leur fournissent de l'énergie, ainsi les “ bugs ” peuvent continuer à se déplacer, et quand ils sont devenus adultes et qu'ils ont assez d'énergie, ils se reproduisent. Au cours de la reproduction, un “ bug ” donne naissance sur place à 2 “ bugs ”, puis meurt et devient une bactérie.
S'ils ne se reproduisent pas, les “ bugs ” meurent de famine (énergie nulle) et deviennent des bactéries.

1. Proposer un diagramme OMT pour préciser ce cahier des charges.

18.2 Implémentation

1. Définir des classes C++ pour simuler la vie des “ bugs ” décrite précédemment.

   Les positions des bugs et des bactéries seront repérées par des points (voir classe `Point` section 18.3.1 page 94) sur une grille 2D rectangulaire.

   On pourra utiliser la classe `multimap` de la bibliothèque STL pour stocker les bugs et les bactéries, et les retrouver facilement par leurs positions.

   Un exemple de fonction `main` est donné en annexe 18.3.2 page 96. Cet exemple permet l'affichage de la grille à l'aide de l'utilitaire `gnuplot`.

   ```
   $ bug | gnuplot
   ```

2. Il y a maintenant plusieurs types de bugs :

   (a) les “ RandomBug ” qui sont les bugs de base,

   (b) les “ CrazyBug ” qui se déplacent en zig-zag comme les fous sur un échiquier,

(c) les "HorseBug" qui se déplacent comme les cavaliers sur un échiquier,

(d) les "SnifferBug" qui regardent où est la nourriture avant de choisir leur direction de déplacement.

Modifier le programme précédent pour tenir compte de ces nouveaux types de bugs.

3. Il y a maintenant plusieurs types de bactéries :

(a) les "LazyBacteria" qui sont les bactéries de base,

(b) les "RandomBacteria" qui se déplacent de manière aléatoire à la manière des "RandomBug",

(c) les "PreyBacteria" qui se déplacent en s'éloignant des bugs.

Modifier le programme précédent pour tenir compte de ces nouveaux types de bactéries.

18.3 Annexes

18.3.1 La classe `Point`

Le fichier `point.h`

```
class Point {
public:
// Allocators
     Point(int x = 0, int y = 0);
     Point(const Point& p);
     Point& operator=(const Point& p);
     virtual  Point(void);
// Comparisons
     bool operator==(const Point& p) const;
     bool operator!=(const Point& p) const;
     bool operator<(const Point& p) const;
     bool operator<=(const Point& p) const;
     bool operator>=(const Point& p) const;
     bool operator>(const Point& p) const;
// Inspectors
     int x(void) const;
     int y(void) const;
// Modifiers
     void setx(int x);
     void sety(int y);
// Input/Output
     friend ostream& operator<<(ostream& stream, const Point& p);
private:
// Attributes
     int _x;
     int _y;
};
```

Le fichier `point.C`

```
Point::Point(int x, int y) :  _x(x), _y(y) {}

Point::Point(const Point& p) :  _x(p._x), _y(p._y) {}

Point& Point::operator=(const Point& p) {
     if(this != &p) { _x = p._x; _y = p._y; }
     return *this;
}

Point:: Point(void) {}

bool Point::operator==(const Point& p) const
     { return (x() == p.x()) && (y() == p.y()); }

bool Point::operator!=(const Point& p) const { return !(*this == p); }

bool Point::operator<(const Point& p) const {
     if(x() < p.x()) return true;
     if(x() == p.x()) return (y() < p.y());
     return false;
}

bool Point::operator<=(const Point& p) const { return !(p < *this); }

bool Point::operator>=(const Point& p) const { return !(*this < p); }

bool Point::operator>(const Point& p) const { return (p < *this); }

int Point::x(void) const { return _x; }

int Point::y(void) const { return _y; }

void Point::setx(int x) { _x = x; }

void Point::sety(int y) { _y = y; }

ostream& operator<<(ostream& stream, const Point& location) {
     return stream << '(' << location.x() << ',' << location.y() << ')';
}
```

18.3.2 La fonction `main`

```
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <fstream.h>

// affichage pour "gnuplot"
void gnuplot(Site* site, const char* className,
                                          const char* prefixe = "/tmp/") {
     char fileName[80] = "";
     if(site) {
          ::strcat(fileName,prefixe); ::strcat(fileName,className);
          ofstream file; file.open(fileName,ios::out);
          multimap< Point, Agent*, less<Point> >::iterator i;
          i = site->population().begin();
          while(i != site->population().end()) {
               if(!::strcmp((*i).second->className(),className)) {
                    file << (*i).first.x() << ' '
                         << (*i).first.y() << endl;
               }
               i++;
          }
          file.close();
     }
}

// la vie artificielle des bugs
int main(void) {
     // initialisation du generateur de nombres aleatoires
     srand(time(0));
     // creation du site
     Site site("Culture biologique");
     // creation des agents
     for(int i = 0; i < 50; i++) {
          site.add(new Bug(),Point(rand()%site.xmax(),rand()%site.ymax()));
          site.add(new Bacteria(),
                           Point(rand()%site.xmax(),rand()%site.ymax()));
     }
     // simulation
     while(site.population().size() > 0 &&
                                       site.population().size() < 1000) {
          site.cleanup();
          cerr << site.population().size() << ' ';
          gnuplot(&site,"Bug"); gnuplot(&site,"Bacteria");
          cout << "plot [0:10][0:10] \"/tmp/Bug\" with points, "
               << "\"/tmp/Bacteria\" with points" << endl;
          site.live();
     }

     return 0;
}
```

Table des Matières

Avant–propos **1**

1 Rappels UNIX **3**
1.1 Le système UNIX de travail . . . 3
1.1.1 Organisation des TP . . . 3
1.1.2 Connexion/Déconnexion . . . 3
1.1.3 Environnement de travail . . . 3
1.2 Organisation des répertoires . . . 4
1.2.1 Création de l'arborescence de travail . . . 4
1.2.2 La variable `PATH` . . . 4
1.3 Corrigés des TP . . . 5
1.3.1 Recherche d'un corrigé . . . 5
1.3.2 Script de recherche . . . 5

2 La compilation en C++ **7**
2.1 Le compilateur C++ . . . 7
2.1.1 Définition d'un compilateur . . . 7
2.1.2 La commande UNIX . . . 7
2.1.3 Les principales options du compilateur C++ . . . 7
2.2 La chaîne de traitement . . . 8
2.2.1 L'édition du code source . . . 8
2.2.2 Le préprocesseur . . . 8
2.2.3 La compilation . . . 8
2.2.4 L'édition de liens . . . 8
2.2.5 L'exécution du programme . . . 9
2.3 Les erreurs de compilation . . . 9
2.3.1 Erreurs liées au préprocesseur . . . 9
2.3.2 Erreurs liées au compilateur . . . 9
2.3.3 Erreurs liées à l'éditeur de liens . . . 9

3 Compilation conditionnelle, compilation séparée **11**
3.1 La compilation conditionnelle . . . 11
3.1.1 Manipulation de fonctions . . . 11
3.1.2 Test d'un fichier . . . 12
3.2 La compilation séparée . . . 12

3.2.1 Edition de liens . . . 12
3.2.2 Utilisation d'une librairie . . . 12
3.3 Fichiers de travail . . . 12
3.3.1 Fichier `fonction.h` . . . 12
3.3.2 Fichier `fonction.C` . . . 13

4 La commande make **15**
4.1 Le manuel `UNIX` . . . 15
4.1.1 `$ man make` . . . 15
4.1.2 *"The* `make` *command"* . . . 15
4.2 Evaluation de fonctions . . . 16
4.2.1 Un "make" explicite . . . 16
4.2.2 Un "make" implicite . . . 16
4.3 The `make` command . . . 17
4.3.1 Command overview . . . 17
4.3.2 Synopsis . . . 19
4.3.3 Description file . . . 20
4.3.4 Macros . . . 22
4.3.5 Suffix Rules . . . 24
4.3.6 Flags . . . 30
4.3.7 Example . . . 31

5 Entrées/sorties en C++ **35**
5.1 Les classes d'entrées/sorties . . . 35
5.1.1 Le graphe d'héritage . . . 35
5.1.2 Injection dans un flot . . . 35
5.1.3 Extraction d'un flot . . . 35
5.2 Copier un fichier . . . 35
5.2.1 `cin` → `cout` . . . 35
5.2.2 `source` → `destination` . . . 36
5.3 `iostream` package . . . 37
5.3.1 Synopsis . . . 37
5.3.2 Core classes . . . 38
5.3.3 Predefined streams . . . 38
5.3.4 Classes derived from `streambuf` . . . 39
5.3.5 Classes derived from `istream`, `ostream`, and `iostream` . . . 39
5.3.6 `class ios` . . . 40
5.3.7 `class ostream` . . . 41
5.3.8 `class istream` . . . 42

6 Un petit analyseur lexical **43**
6.1 Lexèmes . . . 43
6.1.1 Représentation des lexèmes . . . 44
6.1.2 Lecture d'un lexème . . . 44
6.1.3 Ecriture d'un lexème . . . 44

6.2 Analyse lexicale 44
6.2.1 Analyseur lexical 44
6.2.2 Evaluation de fonctions 44

7 Mes premières classes 45
7.1 Conventions d'écriture 45
7.1.1 Interface d'une classe 45
7.1.2 Implémentation d'une classe 46
7.1.3 Fonctions `inline` 46
7.2 Une classe `Point` 46
7.2.1 Du TAD à l'interface de la classe 46
7.2.2 L'implémentation de la classe 47
7.2.3 L'utilisation de la classe 47
7.3 Les fichiers de la classe `Exemple` 47
7.3.1 Fichier `exemple.h` 47
7.3.2 Fichier `exemple.Ci` 48
7.3.3 Fichier `exemple.C` 49

8 Mes premières piles 51
8.1 Du TAD au TCD 51
8.1.1 Le TAD `Pile` 51
8.1.2 Interface publique d'une `PILE` 52
8.1.3 Choix d'une représentation physique 52
8.1.4 Paramétrage de la classe `PILE` 52
8.2 Implémentation en C++ 52
8.2.1 La classe `PILE` 52
8.2.2 Test de la classe `PILE` 53

9 Ma première calculette à pile 55
9.1 Le fichier `Makefile` 56
9.1.1 Compilation de l'analyseur lexical 56
9.1.2 Compilation de la `RPile` 56
9.1.3 Compilation de la calculette 56
9.1.4 Edition de liens 57
9.2 Réalisation de la calculette 57
9.2.1 Traitement des nombres 57
9.2.2 Traitement des fonctions 57
9.2.3 Traitement des opérateurs 58
9.2.4 Autres traitements 58

10 Les opérateurs en C++ 59
10.1 Les chaînes de caractères 59
10.2 Les nombres rationnels 60

11 La généricité en C++ **61**
11.1 Fonctions génériques . . . 61
11.1.1 Prototypes . . . 61
11.1.2 Définitions . . . 61
11.1.3 Instanciations . . . 61
11.1.4 Compilation . . . 61
11.2 Classes génériques . . . 62
11.2.1 Interface . . . 62
11.2.2 Implémentation . . . 62
11.2.3 Instanciations . . . 62

12 Des piles et des tableaux **63**
12.1 La classe `SPile` . . . 63
12.2 La classe `DPile` . . . 63
12.3 Interface de la classe `SPile` . . . 64
12.3.1 Interface publique . . . 64
12.3.2 Interface privée . . . 65
12.4 Interface de la classe `DPile` . . . 65
12.4.1 Interface privée . . . 65
12.4.2 Interface publique . . . 66

13 Des piles et des listes **67**
13.1 La classe `LPile` . . . 67
13.2 Interface de la classe `LPile` . . . 68
13.2.1 Interface publique . . . 68
13.2.2 Interface privée . . . 68

14 L'héritage en C++ **69**
14.1 Modes d'héritage . . . 69
14.2 Liaisons dynamiques et classes abstraites . . . 70
14.3 Les tours de Hanoï . . . 71
14.4 Annexes . . . 72
14.4.1 La classe `Base` . . . 72
14.4.2 La classe `D1` . . . 74
14.4.3 La classe `D2` . . . 75
14.4.4 La classe `D3` . . . 75
14.4.5 Le fichier `dyna.C` . . . 76
14.4.6 La classe `SPile` . . . 77

15 Une famille de piles **79**

16 Les difficultés du C++ **81**
16.1 Constructeurs . . . 81
16.2 Destructeurs . . . 82
16.3 Opérateurs . . . 82
16.4 Généricité . . . 82

16.5 Héritage 83
16.6 Annexes 85
16.6.1 Fichier `tableau.h` 85
16.6.2 Classes `A<T>` et `B` 86
16.6.3 Classes `C` et `D` 87
16.6.4 Classes `E` et `F` 87

17 Standard Template Library **89**
17.1 Itérateurs 89
17.1.1 Définition et utilisation 89
17.1.2 Classe `IstreamIterator<T>` 89
17.2 Séquences 89
17.2.1 `vector<T>`, `deque<T>` et `list<T>` 89
17.2.2 Encore une pile ! 90
17.3 Tables associatives 90
17.3.1 Ensembles et tas 90
17.3.2 Dictionnaires et encyclopédies 90
17.4 Anagrammes 91
17.4.1 Trouver les anagrammes d'un mot 91
17.4.2 Dictionnaire d'anagrammes 91
17.5 Annexes 92
17.5.1 La classe `IstreamIterator<T>` 92

18 La vie artificielle des "bugs" **93**
18.1 Cahier des charges 93
18.2 Implémentation 93
18.3 Annexes 94
18.3.1 La classe `Point` 94
18.3.2 La fonction `main` 96