Étapes de développement d’un programme en C/C++

1) Un éditeur de texte pour écrire les programmes en C/C++ (fichiers.cpp),

2) Un compilateur qui traduit les programmes de C/C++ en fichiers objets (fichiers.obj),

3) Un éditeur de liens (linker) qui transforme les fichiers objets en programmes exécutables (fichiers.exe),

4) Un débogueur pour la recherche des erreurs dans les programmes.

Utilisation de débogueurs (1)

Il est impensable à l'heure actuelle de développer dans un langage de programmation pour lequel il n'existe pas de **débogueur symbolique**. Celui-ci peut proposer une interface graphique sophistiquée (dans le meilleur des cas) ou être accessible via la ligne de commande.

gdb est un débogueur symbolique, c'est-à-dire un utilitaire **Unix** permettant de contrôler le déroulement de programmes C, C++, Pascal ou Fortran.

gdb permet (entre autres) de mettre des points d'arrêt dans un programme, de visualiser l'état de sa pile d'exécution ou de ses variables, de calculer des expressions, d'appeler interactivement des fonctions, etc.

xxgdb est une interface graphique qui facilite l'utilisation de **gdb** sous X-Window. Un autre débogueur **dbx** peut également être appelé via une interface graphique sous X-Window à l'aide des commandes **debugger** ou **dbxtool** (suivant la version du système d'exploitation).

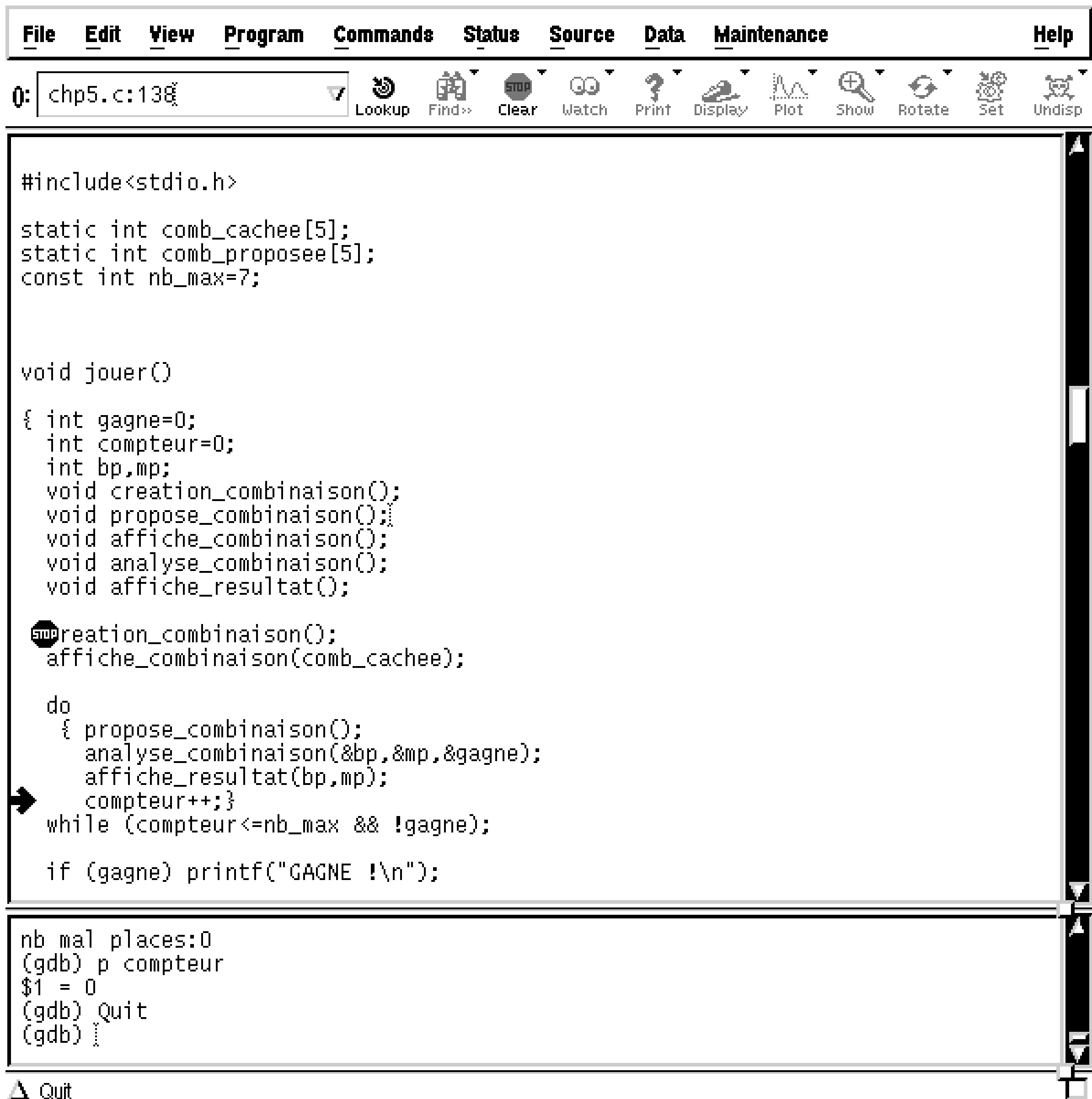
Attention: le débogueur n'est pas toujours indépendant du compilateur. Ainsi, seules les combinaisons suivantes sont toujours valides sur toutes les stations Sun:

compilateur **gcc** + débogueur **gdb** (ou **xxgdb** ou **mxgdb**),

compilateur **cc** + débogueur **dbx** (ou **debugger** ou **dbxtool**).

Utilisation de débogueurs (2)

Les programmes C doivent être compilés avec le compilateur **gcc** avec l'option **-g** pour pouvoir être débogués par **gdb**.



Exemple d'un ***débogueur symbolique*** proposant une interface graphique.

Utilisation de débogueurs (3)

Avec un débogueur, on exécute le programme "pas à pas" et à chaque pas on a accès au contenu des variables.

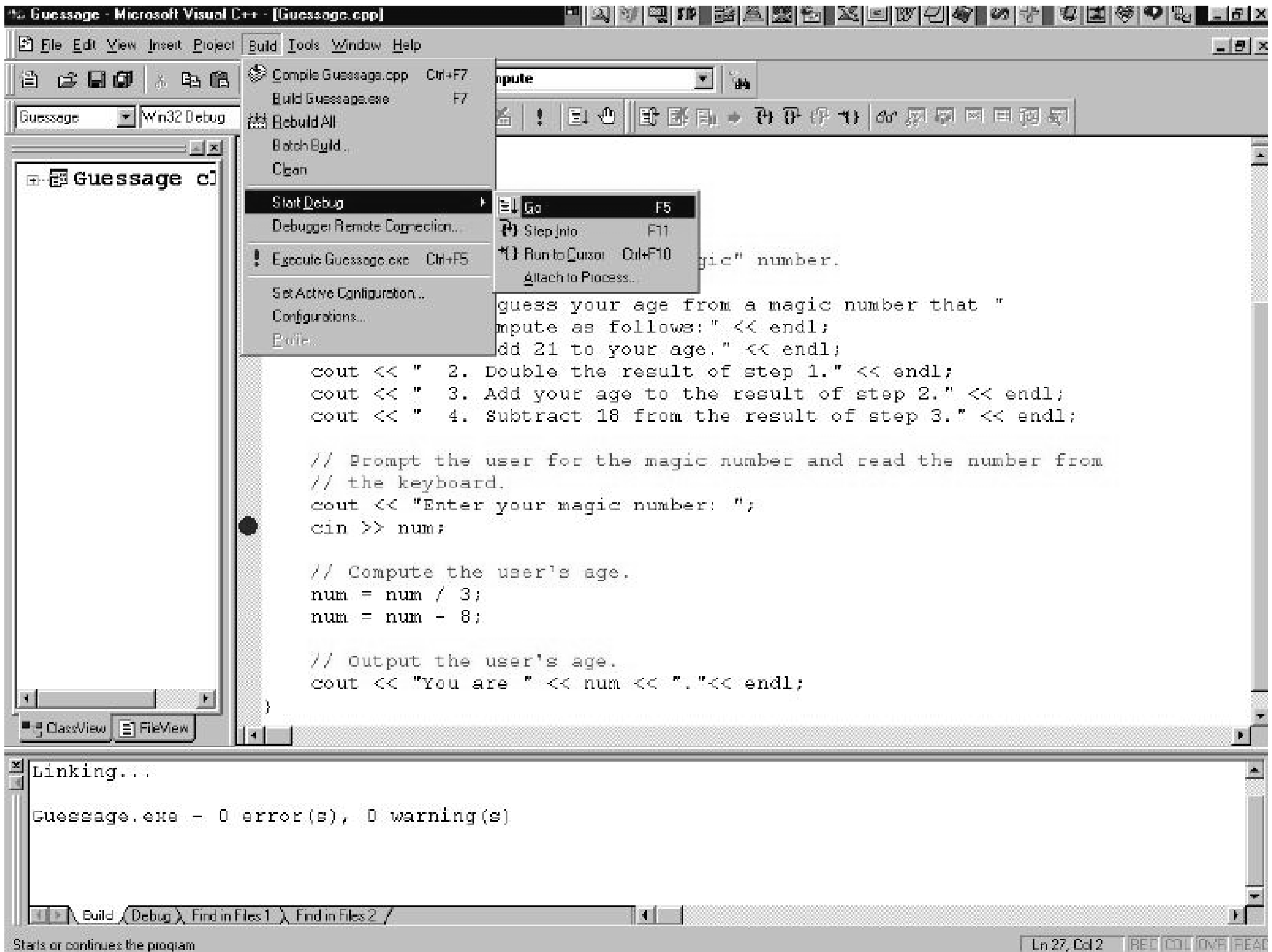
Les commandes les plus utiles d'un débogueur:

- Arrêt sur ligne ou dans une fonction et gestion des points d'arrêt,
- Exécution pas à pas sans entrer dans le code des fonctions,
- Exécution pas à pas en entrant dans le code des fonctions,
- Continuer l'exécution jusqu'au prochain arrêt,
- Lancer une nouvelle exécution,
- Afficher la valeur d'une variable,
- Connaître l'empilement des appels et où l'on se trouve,
- Gérer l'affichage des différents fichiers sources.

Exemple de lancement d'un débogueur:

gcc -g -o myfile myfile.C

Utilisation de débogueurs (4)



Exemple de l'utilisation du débogueur de l'environnement de programmation *Visual C++ 6.0 de Microsoft*.

Le *break point* marqué par un point rouge définit le prochain point d'arrêt du programme.

Compilation séparée (1)

Pour les petits exemples du code étudiés précédemment, le code source a généralement été contenu *dans un seul fichier* .c en C (.C en C++) ou dans un fichier .c et son fichier interface .h.

Pour de plus grosses applications, le code source est généralement *réparti sur plusieurs fichiers* .h et .C et les fichiers sources sont compilés séparément. Il faudra alors être méthodique dans l'organisation des multiples fichiers.

Le code est distribué sur des fichiers .h et .C . Chaque fichier .C possède généralement son fichier .h (ou header) qui est une interface pour l'utilisateur des classes ou des fonctions qu'il implante.

Une fois le code est réparti sur plusieurs fichiers .C et .h , on peut utiliser la **compilation séparée** pour ne recompiler que les fichiers modifiés lors de la dernière modifications (et les fichiers dépendants).

L'utilité de la **compilation séparée** est triple:

- La programmation est modulaire, donc plus compréhensible,
- La séparation en plusieurs fichiers produit des listings plus lisibles,
- La maintenance est plus facile car seule une partie du code est recompilée.

Compilation séparée (2)

Chaque fichier source .C peut être compilé séparément, générant un fichier.o. Le fichier .C contenant le programme principal est alors compilé en utilisant tous les .o (édition de liens) pour générer l'exécutable de l'application.

Supposons par exemple qu'une application est distribuée sur les fichiers **A.C**, **A.h**, **B.C**, **B.h** et **main.C**. On procédera alors de la façon suivante:

$ g++ -c A.C
$ g++ -c B.C

on a maintenant deux fichiers .o: **A.o** et **B.o**. On peut alors obtenir un exécutable nommé **exec**:

$ g++ -o exec A.o B.o main.C

Sous UNIX, on a la notion de "**makefil**e" qui est un fichier de description de commandes de compilation. Un fichier "makefile" étant édité, on exécute la commande "**make**" pour lancer la compilation. Les notions de dépendance de fichiers peuvent être décrites dans le fichier "makefile" de sorte qu'à la compilation (lancée par "make") ne soient compilés que les fichiers nécessaires en fonction des modifications effectuées.

La commande make (1)

L'utilité de la commande **make**:

- la détermination automatique des dépendances,
- la recompilation automatique des fichiers concernés chaque fois qu'une option de compilation ou qu'une dépendance a été modifiée,
- compile les fichiers et crée l'exécutable à l'aide d'une seule commande *make*.

Le schéma de la compilation:

---

*.c *.h (Fichiers source)

•

COMPILATEUR

•

*.o (Fichiers objet)

•

Librairies --> LINKEUR <-- Fichiers objet

•

Fichier exécutable

La commande make (2)

Ce que fait **make**:

- *make* assure la compilation séparée grâce à cc ou gcc,
- *make* utilise des macro-commandes et des variables,
- *make* permet de ne recompiler que le code modifié,
- *make* permet d'utiliser des commandes *shell*, et ainsi d'effectuer une installation d’un logiciel.

Makefile

Le fichier Makefile est un fichier nécessaire à la commande *make*. Un fichier Makefile indique à *make* comment exécuter les instructions nécessaires à l'installation d'un logiciel ou d'une librairie.

Le fichier Makefile doit se trouver dans le répertoire courant lorsqu'on appelle la commande *make*.

Les instructions contenues dans un fichier Makefile obéissent à une syntaxe particulière (*un peu stupide*).

Écriture d’un Makefile

Règles

Les fichiers Makefile sont structurés grâce aux *règles*. Ce sont elles qui définissent ce qui doit être exécuté ou non, et qui permettent de compiler un programme de différentes façons.

Qu'est-ce qu'une règle ?

Une *règle* est une suite d'instructions qui seront exécutées pour construire une *cible*, mais uniquement si des *dépendances* sont plus récentes que l’exécutable. La syntaxe d'une règle est la suivante:

```
cible: dépendances
        commandes
       ...
```

Cible, dépendances et commandes

La **cible** est généralement le nom d'un fichier qui va être généré par les commandes qui vont suivre, ou d’une action gérée par ces mêmes commandes, par exemple *clean* ou *install*.

Les **dépendances** sont les fichiers ou les règles nécessaires à la création de la cible. Les **commandes** sont les suites de commandes *shell* de UNIX qui seront exécutées au moment de la création de la cible.

Exemple d’un Makefile simple

```
# Mon premier Makefile

all:  foobar.o  main.c
	gcc -o main foobar.o main.c

foobar.o:  foobar.c  foobar.h
	gcc -c foobar.c -o foobar.o
```

Si vous avez enregistré l'exemple ci-dessus dans un fichier Makefile, il ne vous reste plus qu'à exécuter *make* dans le même répertoire que celui où vous avez enregistré le fichier.

Make s'exécute tout simplement en lançant la commande:

$ make all

Make va alors interpréter le fichier Makefile et exécuter les commandes contenues dans la règle *all*, une fois que les dépendances *foobar.o* et *main.c* seront vérifiées.

Utilisation des macro-commandes et variables

Il faut considérer les variables de *make* comme des macro-commandes (# *define* en C).

Exemple d’un Makefile un peu plus complexe et commenté:

```
# $(BIN) est le nom du fichier binaire généré
BIN = foo
# $(OBJECTS) sont les objets qui seront générés après la compilation
OBJECTS = main.o  foo.o
# $(CC) est le compilateur utilisé
CC = gcc
# all est la première règle à être exécutée car elle est la première
# dans le fichier Makefile. Notons que les dépendances peuvent être
# remplacées par une variable, ainsi que n'importe quel chaîne de
# caractères des commandes
all: $(OBJECTS)
	$(CC) $(OBJECTS) -o $(BIN)
# ensuite les autres règles
main.o: main.c main.h
	$(CC) -c mainc
foo.o: foo.c foo.h main.h
	$(CC) -c foo.c
```

Conventions d'appellation dans un Makefile (1)

Noms d'exécutables et d'arguments (entre parenthèses les valeurs par défaut):

- AR: programme de maintenance d'archive (*ar*),
- CC: compilateur C (*cc*),
- CXX: compilateur C++ (*c++*),
- RM: commande pour effacer un fichier (*rm*),
- TEX: programme pour créer un fichier TeX dvi,
- ARFLAGS: paramètres à passer au programme de maintenance d'archives (),
- CFLAGS: paramètres à passer au compilateur C (),
- CXXFLAGS: paramètres à passer au compilateur C++ ().

Noms de cibles:

Un utilisateur de *make* peut donner à ses cibles le nom qu'il désire. Mais pour des raisons de lisibilité, on donne toujours un nom standard à ses cibles selon leur comportement. Quelques exemples de cibles standard:

- all: compile tous les fichiers source pour créer l'exécutable principal,
- install: exécute *all*, et copie l'exécutable, les librairies, les datas, et les fichiers d'en-tête s'il y en a dans les répertoires de destination,
- uninstall: détruit les fichiers créés lors de l'installation, mais pas les fichiers du répertoire d'installation (où se trouvent les fichiers source et le Makefile),

Conventions d'appellation dans un Makefile (2)

Noms de cibles (suite):

- clean: détruit tout les fichiers créés par *all*,
- info: génère un fichier *info*,
- dvi: génère un fichier *dvi*,
- dist: crée un fichier *tar* de distribution.

Noms de répertoires de destination (entre parenthèses les valeurs par défaut):

- prefix: racine du répertoire d'installation (/usr/local),
- exec_prefix: racine pour les binaires ($(prefix)),
- bindir: répertoire d'installation des binaires ($(exec_prefix)/bin),
- libdir: répertoire d'installation des librairies ($(exec_prefix)/lib),
- datadir: répertoire d'installation des données statiques pour le programme ($(exec_prefix)/lib),
- statedir: répertoire d'installation des données modifiables par le programme ($(prefix)/lib),
- includedir: répertoire d'installation des en-têtes ($(prefix)/include),
- mandir: répertoire d'installation des fichiers de manuel ($(prefix)/man),
- man*x*dir: répertoire d'installation des fichiers de la section *x* du manuel ($(prefix)/manx),
- infodir: répertoire d'installation des fichiers info ($(prefix)/info),
- srcdir: répertoire d'installation des fichiers source ($(prefix)/src).

Le "Monde" orienté-objets

- Le monde orienté-objets représente les objets de la réalité par des objets programmés dans un langage comme C++ ou Java.

- Comme les objets de la réalité, les objets programmés ont des propriétés et des fonctionnalités possibles/prévues.

- Le langage orienté-objets est basé sur la communication entre objets.

- Les langages orientés-fonctions comme Fortran ou C sont basés sur la suite d'appels de fonctions dans lesquelles on passe les données.

- L'orienté-objets transporte les données avec les fonctionnalités.

- On parle souvent de la validation des données, du contrôle des données et de la vérification des données.

- Un langage orienté-objets met beaucoup plus l'accent sur les données qu'un langage orienté-fonctions.

Premier exemple de programme C++

```
// ce programme lit 2 nombres et affiche en résultat le plus grand des deux
    #include <iostream.h> // au lieu d'inclure le fichier stdio.h

    int max(int, int); //déclaration (prototype), le coprs est défini plus loin
    void ecrire(int n) //déclaration et définition
    { cout << n << "est la plus grande valeur\n";
    }

    int main(void)
    { int val1, val2;
      cout << "Bienvenu! Je suis un super programme\n";
      cout << "Entrez 2 nombres : ";
      cin >> val1 >> val2;
      int valeurMax = max(val1,val2);  // déclaration inhabituelle
      ecrire(valeurMax);
      return 0;
    }
    int max(int i, int j)      //corps de la fonction (définition)
    { if (i > j)
        return i;
      else
        return j;
    }
```

Différences entre le C-ANSI et le C++ (1)

Le C++ est compatible avec les fonctions des bibliothèques du C-ANSI (notamment: **printf**() et **scanf**(), ...).

Cependant, il est préférable d'utiliser la bibliothèque iostream, qui est mieux **adaptée à la *philosophie objets***. Les instructions **cin** >> et **cout** << permettent respectivement de lire sur l'entrée standard ou d'écrire sur la sortie standard.

Les opérateurs >> et << sont **surchargés**: ils acceptent aussi bien des arguments de type entier, réel ou des chaînes de caractères, ...

▸ Emplacement des déclarations des variables

En C-ANSI une déclaration de variable ne peut être faite qu'au début d'un bloc d'instructions. Le langage C++ est **plus souple**:

```
int valeurMax = max(val1,val2);     //voir l’exemple précédent
```

Une déclaration de variable peut se situer n’importe où dans le programme, pourvu qu'elle précède l'utilisation de la variable.

NE PAS EN ABUSER !

La possibilité de pouvoir déclarer une variable en tout point du code peut sembler négligeable, elle prend toute son importance avec objets de types définis par l'utilisateur (les classes).

Différences entre le C-ANSI et le C++ (2)

La **portée** d'une variable s'étend depuis l'endroit ou elle est déclarée jusqu'à la fin du bloc qui la contient.

Exemple:

```
int main(void)
{
        int n;              //déclaration de n
        ...
        cin >> n;           //saisie de sa valeur
        int q = 2*n-1;      //déclaration de q et initialisation
        ...
        for (int i = 0; i < q; i++)
        {
          int j = i*2;
        }
        //ici, la variable j n'existe plus
        cout << i << endl;              //mais i est visible jusqu'à la fin du main()
        ...
        for (int i = 0; i < n; i++)     //Erreur : redéclaration
        for (i = 0; i < 5; i++)         //OK
        ...
}
```

Différences entre le C-ANSI et le C++ (3)

- En C++, un **pointeur** de type void * (pointeur vers un objet de type inconnu) n'est pas converti automatiquement en un pointeur d'un autre type.

void *PTgen;

int i;

int *PTint = &i; //déclaration de Ptint (pointeur d'entier) et initialisation

PTgen = PTint; //légal en C-ANSI et en C++

PTint = PTgen; //légal en C-ANSI uniquement

PTint = (int *)PTgen; //OK en C++

- **Initialisation de tableaux de caractères**

char t[5] = "hello";

- Erreur de compilation en C++ (la dimension insuffisante pour recevoir le caractère \0 de fin de chaîne)
- Le compilateur du C est insensible à cette erreur

Solution - faites confiance au compilateur et écrivez :

char t[] = "hello";

Différences entre le C-ANSI et le C++ (4)

- Les **constantes** de type **char**

En C++, une constante telle que 'a' est de type **char** alors qu'elle était implicitement convertie en **int** en C-ANSI.

sizeof('a') vaut 1 en C++, mais vaut 4 (ou 2) en général en C.

- **Structures et énumérations**

Les constructeurs **struct** et **enum** permettent, en C++, de **définir de nouveaux types** de données utilisables par leur nom (inutile d'utiliser **typedef** comme en C).

- Exemples dans le style C

```
enum couleur {rouge, vert, bleu,...};
enum couleur c1;
typedef enum boolean {false, true} bool;
bool b1, b2;
```

- Style C++

```
enum couleur {rouge, indigo, jaune,...};
couleur c1;
enum boolean {false, true};
boolean b1, b2;
```

Les types définis à l'aide des mots-clés **enum** et **struct** se comportent comme les types de base du langage.

Différences entre le C-ANSI et le C++ (5)

- **utilisation du qualitatif const**.

La norme C-ANSI a introduit ce qualitatif, utilisable également en C++. Dans les deux langages, **const** sert à designer une variable protégée en écriture.

```
const short taille = 10;
taille++;              //Erreur

for (int i = 0; i < taille; i++)         //OK
short *ps = &taille;                     //NON
*ps = 20;   //voici pourquoi l'affectation précédente est illégale

const short *pq = &taille;            //OK
*pq = 20;    //interdit : l'objet pointé est constant
```

L'utilisation de **const** est moins intéressante en C-ANSI qu'en C++:

```
const int size = 100;
char tab[2*size];       //légal en C++, mais illégal en C-ANSI
```

Différences entre le C-ANSI et le C++ (6)

- Une déclaration comme, par exemple **const int** N = 5 est équivalente à :

en C++ : **static const int** N = 5;

en C-ANSI : **extern const int** N = 5;

Ceci empêche en C-ANSI de placer la définition d'une constante dans un fichier d'en-tête (fichier.h).

Problème à l'édition de liens si ce fichier est importé (#include) par plusieurs modules (*.c). N est une variable globale (à tous les modules) définie plusieurs fois.

En C, on utilise plus les directives #define (exemple #define N 5). En C++, les identificateurs **const** peuvent figurer dans les fichiers d'en-tête: pas de conflit lors de l'édition de liens des différents modules. L'attribut #define limite la visibilité d'une variable au module (*.c) dans lequel elle est définie (ou importée par #include).

Différences entre le C-ANSI et le C++ (7)

- **Fonctions inline** (en ligne)

Fonctions dont le code est inséré 'en ligne' par le compilateur. Elles jouent le même rôle en C++ que les macro-commandes en C, mais elles sont d'une utilisation beaucoup plus sûre.

Les macros en C sont parfois source de surprises désagréables

```
#define carre(y)        (y*y)
```

Appel avec carre(a+1) si a = 2, retourne la valeur 5 !!

Code généré après la substitution : a+1*a+1, soit 2*a +1 !!

La macro aurait du être écrite comme suit:

```
#define carre(y)        ((y)*(y))
```

En C++, on utilise de préférence une fonction **inline**

```
inline int carre(int nombre) { return nombre * nombre; }
//Ici on utilise un argument type (nombre)
int i = 2;
int r1 = carre(i+1);
//r1 vaut 9 !
```

Différence avec une fonction 'normale': expansion du code à la compilation. Gain en temps exécution. Cependant, il faut réservez l'instruction **inline** pour les fonctions très courtes et peu complexes.

Entrées/Sorties en C++ avec cin, cout et cerr (1)

Les entrées/sorties en langage C s'effectue par les fonctions ***scanf*** et ***printf*** de la librairie standard du langage C. Il est possible d'utiliser ces fonctions pour effectuer les entrées/sorties de vos programmes en C++, mais cependant les programmeurs C++ préfèrent les entrées/sorties par flux.

Trois flots sont prédéfinis lorsque vous avez inclus le fichier en-tête ***iostream.h***:
cout - qui correspond à la sortie standard,
cin - qui correspond à l'entrée standard,
cerr - qui correspond à la sortie standard d'erreur.

Exemple :

```
#include <iostream.h>
void main()
{
   int i = 123;
   float f = 1234.567;
   char ch[80] = "Bonjour\n", rep;
   cout << "i=" << i << " f=" << f << " ch=" << ch;
   cout << "i = ? ";
   cin >> i; // lecture d'un entier
}
```

Entrées/Sorties en C++ avec cin, cout et cerr (2)

Tout comme pour la fonction ***scanf***, les espaces sont considérés comme des séparateurs entre les données par le flux ***cin***. Notez l'absence de l'opérateur & dans la syntaxe du ***cin***. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

INTERET de ***cin***, ***cout*** et ***cerr***:

> la vitesse d'exécution plus rapide: la fonction ***printf*** doit analyser à l'exécution la chaîne de formatage, tandis qu'avec les flots, la traduction est faite à la compilation.

Exemple:

```
#include <stdio.h>
#include <iostream.h>
void main()
{ int i = 1234;
  double d = 567.89;
  printf("i = %d d = %d !!!!!\n", i, d); // erreur: %lf normalement pour d
  cout << "i= " << i << " d= " << d << "\n";
}
/* Résultats de l'exécution
i= 1234     d= -5243 !!!!!!!
i= 1234     d= 567.89
```

Les manipulateurs

Les manipulateurs sont des éléments qui modifient la façon dont les paramètres sont lus ou écrits dans le flot. Les principaux manipulateurs sont:

Dec	lecture/écriture d'un entier en décimal
Oct	lecture/écriture d'un entier en octal
Hex	lecture/écriture d'un entier en hexadécimal
Endl	insère un saut de ligne et vide les tampons
setw(int n)	affichage de n caractères
setprecision(int n)	affichage de la valeur avec n chiffres avec éventuellement un arrondi de la valeur
setfill(char)	définit le caractère de remplissage
Flush	vide les tampons après écriture

Exemple:

```
#include <iostream.h>
#include <iomanip.h>
void main()
{ int i = 1234;
  float p = 12.3456;
  cout << "|" << setw(8) << setfill('*') << hex << i << "|\n" << "|" <<
  setw(6) << setprecision(4) << p << "|" << endl;
}
// ------------------ résultats de l'exécution ------------------
|*****4d2|
|*12.35|
```