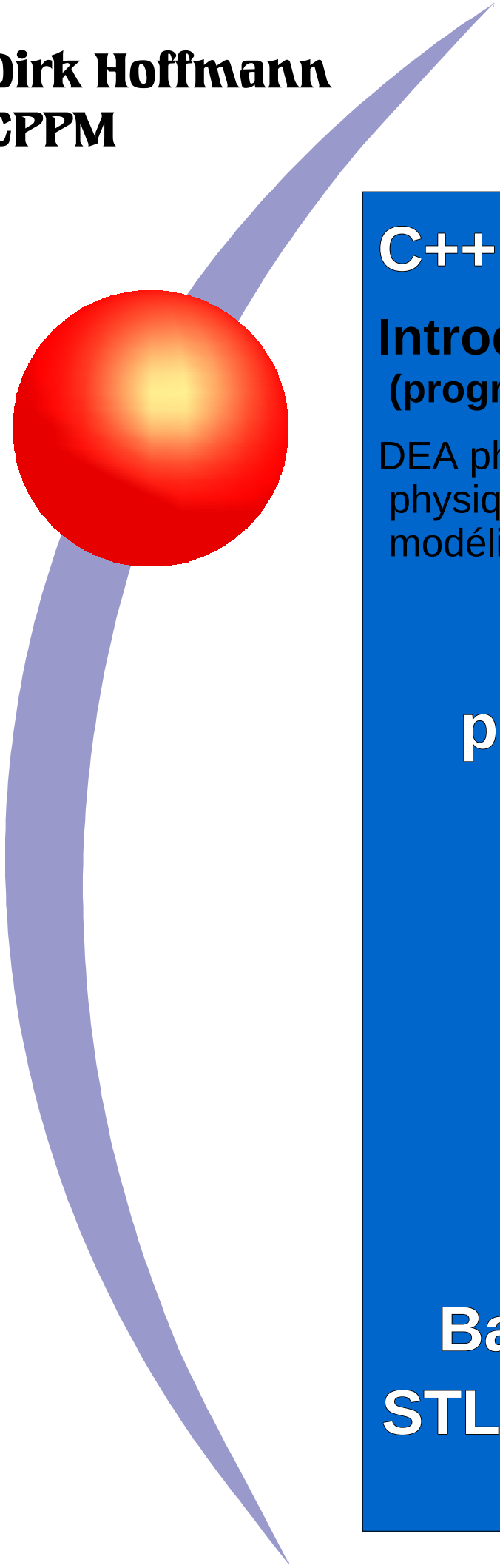
Dirk Hoffmann
CPPM



C++ pour physiciens

Introduction à la POO
(programmation orientée objet)

DEA physique des particules, physique mathématique et modélisation *promotion 2003*

Bases de la programmation

Boucles

Conditions

Variables

Opérateurs

Fonctions

Pointeurs

Classes

Bases de la POO

STL/ROOT/FeynDiag

Java

Fascicule accompagnant le cours de

Programmation OO

pour le DEA de Physique des Particules, Physique Mathématique et Modélisation

Université d'Aix-Marseille, année 2003/2004

J'ai édité ce fascicule rapidement pour documenter le cours, et en particulier parce que certains tableaux n'étaient pas bien sortis sur les polycopies. Une réédition suivra certainement quand je donnerai ce cours la prochaine fois.

Remerciements, livres recommandés

Tout d'abord, je remercie Patrice Payre pour sa confiance et ses indication très utiles concernant ce cours de « C++ pour étudiants en DEA de physique » que j'ai tenu pour la première fois en 2003. Dans les premiers chapitres, j'ai surtout profité de ses notes et du modèle du « BaBar C++ course » de Paul Kunz que j'avais moi-même suivi quelques années auparavant au DESY, à Hambourg. Il était à son tour parti du livre et des exemples du livre de John J. Barton et Lee R. Nackman, d'après mes informations. Il semble donc évident de mentionner ces sources pour une lecture avancée et l'approfondissement des sujets traités.

[1] Paul F. Kunz, **BaBar C++ course**, transparents disponibles sur
`ftp://ftp.slac.stanford.edu/users/pfkb/c++class/`

[2] John J. Barton, Lee R. Nackman, **Scientific and Engineering C++**,
Addison-Wesley, ISBN 0-201-53393-6,
`http://www.awprofessional.com/search`, chercher le terme “barton”

Le marché ne manque aujourd'hui pas de livres sur le langage C++ ou traitant la programmation orienté objet de manière abstraite ou générale. A titre d'exemple, je veux nommer deux autres livres, diamétralement opposé au plan de leur complexité et du niveau en informatique requis du lecteur. La description du C++ par son auteur Bjarne Stroustrup n'est probablement pas abordable pour quelqu'un qui n'a pas déjà une certaine expérience en deux ou trois autres langages de programmation, alors que le deuxième reprend beaucoup de notions de base qui devraient déjà être bien connues à un programmateur d'un autre langage quelconque.

[3] Bjarne Stroustrup, **The C++ Programming language**, Addison-Wesley, ISBN-201-51459-1 (2nd edition, 1990)

[4] Jesse Liberty, **Le langage C++**, ISBN 2-7440-0445-6, Simon & Schuster Macmillan France (1998)

Ce dernier livre a été traduit de l'anglais « Learning C++ in 24 hours », ce qui n'est pas un titre qui parle pour le bon jugement de l'auteur. Néanmoins, il est très explicite et donne beaucoup d'exemples. Et son titre est mieux choisi dans la version française.
Finalement, je mentionne mon site web, `marwww.in2p3.fr/~hoffmann/`, sur lequel j'ajoute régulièrement des documents ou nouvelles concernant mes cours.

Sur la programmation en Java, il y a probablement encore davantage de livres que sur C++ ; et le spectre de leur qualité est par conséquent bien rempli. A titre d'exemple, j'en cite un qui a été très bon dans l'édition suivante :

[5] Roger Cadenhead, Laura Lemay, **Teach yourself Java 2**, Professional Reference Edition, 2nd edition, Sams Publishing, ISBN 0-672-32061-4, `http://www.java21pro.com/`

Curieusement, ce livre a eu des prédécesseur et successeurs assez nombreux, au moins une édition par version de Java ainsi que des saveurs « professionel » et « standard ». Malheureusement, la qualité et bizarrement le contenu et la structure des différentes éditions varient beaucoup. Je me vois donc contraint d'avertir que l'édition actuelle ne correspond éventuellement plus aux critères.

Un cours (tutorial) de Java assez complet a été donné au CERN :

[6] Raul Ramon-Pollàn, **The Java Series**,
actuellement sur `http://ref.cern.ch/CERN/Tutorial/Java/`

Et il faut souligner que la documentation de Java ainsi que les tutorials, tous fournis par Sun (et en général également disponibles en forme de livre) sont excellents! Je recommende fortement à tout programmateur qui s'aventure dans le langage Java, de faire un tour sur `http://java.sun.com`, dans les section « API » (de la version actuelle) et « tutorial ».

Ce cours, et en particulier ce fascicule, ne peuvent constituer la source unique pour devenir programmateur de C++. L'expérience montre que les niveaux d'entrée des étudiants du DEA sont trop dispersés pour pouvoir fournir un document pédagogique unique. Il est indispensable, tout comme dans les autres matières du DEA, que les étudiants choisissent de la lecture supplémentaire, soit dans la liste énoncée plus haut, soit dans une bibliothèque ou une libraire. Le travail individuel et la propre initiative sont nécessaires pour réussir, ainsi que le travail indépendant sur un ordinateur avec compilateur et environnement appropriés. Quelques machines du CPPM sont aussi à la disposition des étudiants pour cela pendant toute la durée du DEA!

Ces remarques sont d'autant plus vraies qu'une sélection très rigoureuse a dû être faite pour tenir dans les 12 heures allouées dans le programme du DEA pour ce cours. Après une introduction générale à la programmation sous Unix en pratique, les bases de la syntaxe du langage C++, la compilation et la production d'un programme exécutable, le sujet de la programmation orienté objet proprement dite ne sera abordé que dans la deuxième moitié du cours. Cette initiation sera suivie de quelques exemples concrets d'utilisation d'environnements de programmation C++. Un survol rapide du langage Java, en tant que super-C++ (C+++) permettra de voir la grande utilité des concepts appris, dans un cadre qui dépasse le langage même du C++, dans quelques exemples d'applications graphiques.

Bases, Syntaxe

Le programme le plus simple en C++ se compose essentiellement de trois lignes (Et il n'a rien à voir avec le programmation orienté objet!) :

```
#include <iostream.h>

main()
{
      cout << "A vos ordres, maître\n";
}
```

Pour l'utiliser, il suffit de mettre ces lignes dans un fichier texte (appelé la source ou le code source), nommé avec un suffix .C (C majuscule), comme trivial.C. Il faut ensuite compiler ce fichier à l'aide du compilateur C++ qui est dénommé CC ou g++ sur la plûpart des systèmes. On trouve ensuite un nouveau fichier a.out dans le répertoire qui représente le fichier *exécutable* et peut ensuite être utilisé comme commande.
Sur une machine appelée marcotte.in2p3.fr, cela peut se transcrire sur l'écran de la manière suivante, en supposant que le répertoire actuel était nouvellement créé ou vide auparavant :

```
user@marcotte > g++ trivial.C
user@marcotte > ls
a.out  trivial.C
user@marcotte > a.out
A vos ordres, maître
```

Nous avons indiqué en **gras**, les commandes saisies par l'utilisateur même.

Le premier pas vers l'apprivoisement de l'ordinateur est fait. Regardons le contenu de cette recette, le code source, en détail : La première ligne est simplement une annonce au compilateur (déclaration) de la bibliothèque qui sera nécessaire pour utiliser la fonction de sortie standard (`cout`) plus loin. La deuxième ligne non-triviale est la déclaration de la fonction principale (`main()`), par laquelle le fil d'exécution du processeur va entrer dans le programme (et ressortir). Cette fonction est délimitée par des accolades (`{}`). Finalement, la seule ligne relevante pour le mini-algorithme de notre programme contient

- un appel à (la fonction de) l'objet de sortie standard[1],
- l'opérateur d'enchaînement d'éléments de sortie (<<) dans ce contexte
- et une chaîne de caractères contenant le texte à afficher à l'écran.

Notons pour les initiés (en langage C, par exemple) que l'utilisation directe des fonctions de système comme `printf()` en plus de `cout` peut poser des problèmes, puisque ce dernier gère la sortie (vers ce même `printf()`, certes) avec ses propres mécanismes et tampons. Des mélanges de sorties peuvent alors arriver au cours du programme, et il est recommandé de choisir tôt dans le développement d'un programme soit l'une, soit l'autre méthode. L'objet `cout` est très puissant à cause de toutes ses options basées sur la (re)définition des opérateurs << et >>. Or, la compréhension précise de ce mécanisme dépasse le cadre de ce cours. Nous allons voir au fur et à mesure quelques aspects importants pour pouvoir nous servir des fonctionnalités essentielles.

La chaîne de caractères est délimitée par des guillemets (`"`). La combinaison `\n` à la fin de la chaîne de caractères provoque un retour à la ligne (`n` de *newline* en anglais pour « retour chariot »).

Boucle while

Après ce premier exemple assez banal, nous allons confier une tâche légèrement plus complexe à notre ordinateur, celle de la répétition d'une ou plusieurs instructions. Considérons pour cela l'exemple suivant.

```
#include <iostream.h>

int main()
{
      int i=10;
      while (i>0) {
            cout << "Combien de fois faut-il "
                        "que je le répète ?\n";
            i--;
            }
      return 0;
}
```

Nous avons un certain nombre d'éléments nouveaux dans cet exemple :

- la définition d'une variable (`i`), son initialisation et son incrémentation,
- une description de boucle,
- la composition d'une chaîne de caractères longue et
- le renvoi d'une valeur de résultat à partir de la fonction principale.

Avant de décrire une tâche répétitive, il convient de trouver une condition de fin de boucle. A présent, nous allons utiliser une variable entière (`int i`, pour *integer* en anglais) qui sera décomptée de 10 à 0. L'instruction `int i=10;` déclare à la fois la variable `i` comme étant une variable qui peut prendre toute valeur entière (dans les limites de la précision autorisée) et l'initialise avec la valeur initiale 10.

La description formelle de la boucle est de la forme

`while (`*CONDITION*`) { `*BOUCLE*` }`,

ce qui se traduit assez précisement en français

« pendant que *CONDITION* (est vraie), alors {*BOUCLE*}! »

Quel est le contenu de *BOUCLE* dans notre cas ? Lors de chaque passage dans la boucle, le programme imprime un texte, de manière similaire au premier exemple. Ensuite, la variable `i` est décrémentée (Sa valeur est réduite d'un.) dans l'instruction-opérateur `i--;`.

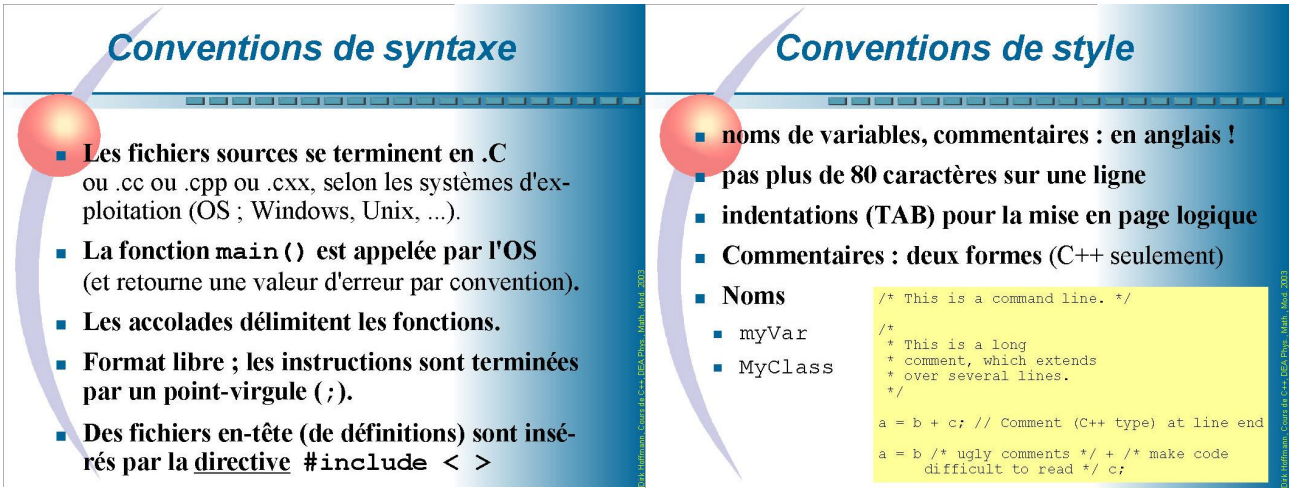
La chaîne de caractères est découpée pour améliorer la lisibilité et surtout pour servir simplement

1 Plus précisément, c'est une instance de la classe de sortie standard. Cela explique pourquoi l'utilisation de l'opérateur << (redéfini!) est permise ici. Il est trop tôt pour faire cette distinction subtile à cet endroit dans le cours.

comme exemple d'une chaîne composée. De manière générale, a syntaxe `"abc" "def"` est strictement identique à `"abcdef"`.

Finalement, l'instruction `return 0;` permet de retourner une valeur de résultat au programme appelant. Dans le cas d'un shell standard, elle peut être vérifiée dans la variable `$?` (par `echo $?` dans les shells `zsh` ou `csh` par exemple). La convention veut que 0 est retourné par un programme qui s'est déroulé comme prévu. Une valeur différente de 0 doit être retournée en cas de problème (erreur de calcul, fichier indispensable pas lisible ou absent, …). Dans un contexte plus complexe ou l'exécution automatique de plusieurs programmes dans un script shell, ce mécanisme de valeur-résultat permet de conclure rapidement et de manière générale, si tous les programmes ont fonctionné de la manière voulue.

Conventions de syntaxe et de style



Accolades

Le placement des accolades `{}` est un grand sujet de discussion et polémique. Je n'y ajouterai pas. A mon avis, un bon programmateur doit

1) choisir une manière qui lui convient et l'utiliser avec conséquence et discipline, mais surtout

2) tenter de comprendre et respecter la convention choisie dans un code qu'il est amené à modifier.

Comme cette deuxième règle est plus fréquemment applicable, elle est la plus importante des deux!

Voici trois manières de placement d'accolades qu'on rencontre le plus souvent :

```
int main() {
      int i=10;
      while (i>0) {
            i--;}
      exit(0);}
```

```
int main()
{
      int i=10;
      while (i>0)
      {
            i--;
      }
      exit(0);
}
```

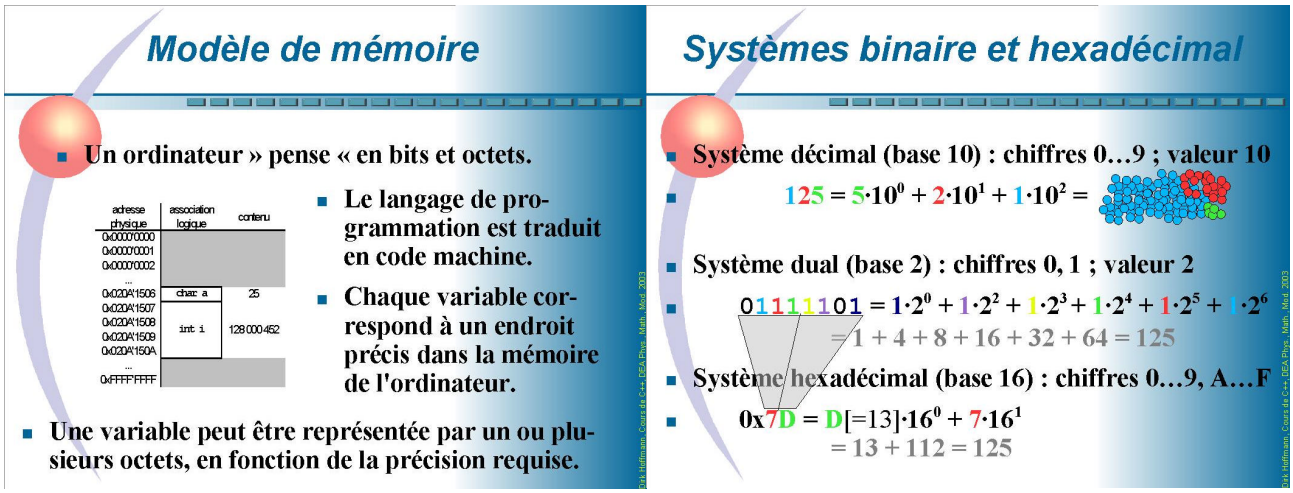
```
int main() {
      int i=10;
      while (i>0) {
            i--;
      }
      exit(0);
}
```

Eléments du langage C++

Variables

- Les noms de variable commencent par une lettre ou le sous-tiret (_) et distinguent majuscules/minuscules. Ils peuvent contenir lettres, sous-tiret et chiffres.
- L'initialisation d'une variable est possible sur la même ligne que sa déclaration.
- La déclaration multiple est mise, mais prête facilement à la confusion.
- Il n'y a pas de variables implicites. Les variables déclarées dans les fichiers en-tête peuvent parfois apparaître comme implicite, mais ne sont en réalité que bien cachées.
- La déclaration avant la première utilisation d'une variable est obligatoire. Cependant, la première instruction exécutable peut précéder une déclaration d'une variable quelconque. D'habitude, la déclaration d'une variable se trouve près de sa première utilisation ce qui facilite la compréhension du code.

```
int i = 3;
float x =  10.5;
i = i+1;
int j = i+128;
```



Types de variables

Nombres entiers

type C++	*taille (bits)*	*domaine numérique*
`long` `long int`	32	$-2^{31}…2^{31}-1$ = 2147483647
`int`	32*	$-2^{31}…2^{31}-1$
`unsigned long` `unsigned long int`	32	$0…2^{32}-1$ = 4294967295
`unsigned int`	32*	$0…2^{32}-1$
`short`	16	$-2^{15}…2^{15}-1$ = 32767
`unsigned short`	16	$0…2^{16}-1$ = 65535
`char`**	8	$-2^{7}…2^{7}-1$ = 127
`unsigned char`	8	$0…2^{8}-1$ = 255
`signed char`	8	$-2^{7}…2^{7}-1$ = 127
`bool`	1	0 ; 1***

* La taille du type int n'est pas définie dans les standards C ou C++. On peut toutefois se baser sur l'inégalité sizeof(`long`)≤sizeof(`int`)≤ sizeof(`short`)>sizeof(`char`).

** Les types `char`, `unsigned char` et `signed char` sont trois types distincts en C++!

*** Le type de variable `bool` est spécial dans le sens qu'il ne prend que deux valeurs et est représenté par un seul bit. Tous les compilateurs C++ ne la comprennent pas, mais elle fait partie du standard ISO à ce jour. Elle se comporte comme le résultat des opérateurs relationnels plus loin dans ce chapitre, où quelques détails supplémentaires seront donnés.

Nombres à virgule flottante

type C++	*taille (bits)*	*domaine numérique*	*précision*
`float`	32	$\pm 1{,}4\cdot10^{-45}…\pm 3{,}38\cdot10^{38}$	7 chiffres déc.
`double` `double float`	64	$\pm 4{,}94\cdot10^{-324}…\pm 1{,}78\cdot10^{308}$	15 chiffres déc.

La taille (et par conséquent la précision) ainsi que le format des nombres à virgule flottante est encore moins bien spécifiée que pour les nombres entiers. On peut toutefois retenir la formule

sizeof(`double`)≤sizeof(`float`).

Un format courant pour les nombres à virgule flottante est celui spécifié par IEEE. Les domaines numériques et la précision approximative sont indiquées pour ce format-là. Comme les nombres sont stockés comme des nombres binaire avec exposant et mantisse binaires, une évaluation exacte de la précision en nombre décimales est difficile à déterminer. Tout calcul numérique critique à cet égard devrait être précédé d'une étude mathématique numérique appropriée, afin d'éviter les surprises ou, pire, la publication de résultats erronés.

Conversion

La conversion entre deux types de variables dans un expression se fait implicitement. Il est donc permis de définir une variable entière par un nombre à virgule flottant, comme dans l'exemple suivant.

```
float x = 10.5;
int i = x;
```

Toutefois, le dépassement de la précision maximale (entre variables d'entiers de types différents)

comme ceci n'est pas défini :

```
      unsigned short surprise = -1;
```

Le résultat peut être +1 ou 65535, selon le type de compilateur!

Limites

Si on envisage d'écrire un code portable, il est sage d'utiliser un fichier standard qui définit les limites de précision dans une implémentation donnée de compilateur. L'exemple suivant utilise les définitions du fichier limits.h pour imprimer les valeurs extrêmes permises pour chaque type d'entier.

```
#include <limits.h>
#include <iostream.h>
int main() {
      cout << "short: " << SHRT_MIN << ".." << SHRT_MAX << "\n";
      cout << "int:   " <<  INT_MIN << ".." <<  INT_MAX << "\n";
      cout << "short: " << LONG_MIN << ".." << LONG_MAX << "\n";
      return 0;
}
```

Un fichier similaire float.h contient des définitions descriptives pour les variables à virgule flottante.

On s'attendra à des changements significatifs lorsque les architectures d'ordinateurs seront basés sur 64 bits par défaut (et de même bien entendu lors du passage aux architectures à 128 bits, si il a lieu un jour).

Opérateurs

Opérateurs arithmétiques

opération	*opérateur en C++*
addition	`x+y`
soustraction	`x-y`
multiplication	`x*y`
division	`x/y`
module	`x%y`
positif unaire	`+x`
négatif unaire	`-x`
postincrément	`x++`
préincrément	`++x`
postdécrément	`x--`
prédécrément	`--x`

L'expression `++x` signifie `x=x+1`, par exemple dans l'expression `y=++x;` qui est équivalente à `y=(x=x+1);`. Le post-incrément a le même effet sur la variable `x`, mais utilise l'ancienne valeur, avant l'incrémentation, dans un expression. De manière pas tout à fait équivalente, on peut expliquer l'effet de `y=x++;` par `y=(int tmp=x, x=x+1, tmp);`. La raison d'implémenter ces deux manières d'incrémentation/décrémentation est d'une part que beaucoup de CPU disposent d'une instruction qui perment d'effectuer l'opération en un minimum de cycles, et d'autre part que cette notation abbrégée simplifie la lecture du code dans beaucoup de cas. En général, un code concis et

univoque est plus clair et compréhensible.

L'exemple suivant permet de se rendre compte du fonctionnement des deux opérateurs d'incrémentation concis, à condition de le faire tourner. (Cette forme d'apprentissage « by trial » est d'ailleurs très utile et souvent plus rapide et plus fiable que les documentations.) Je laisse au lecteur la variation avec les post-incrémentations (`i++;`).

```
#include <iostream>

int main() {
      int i = 1;

      cout << i << ", ";
      cout << (++i) << ", ";
      cout << i << ", ";
      cout << (i++) << ", ";
      cout << i << "\n";

      return(0);
}
```

Opérateurs relationnels

relation	*opérateur en C++*
inférieur	`x<y`
inférieur ou égal	`x<=y`
supérieur	`x<y`
supérieur ou égal	`x>=y`
égal	`x==y`
différent de	`x!=x`

Le résultat est logique (1 bit, booléen, `bool`). Le paragraphe suivant traitera l'équivalence et la conversion numérique–logique plus en détail.

Opérateurs logiques (de Boole)

relation	*opérateur en C++*
faux	`true`
vrai	`false`
inversion	`!x`
conjonction	`x&&y`
disjonction	`x||y`

Les définitions pour `true` et `false` n'existent que si le type `bool` est implémenté, ce qui est le cas pour les compilateurs qui suivent le standard ISO. Autrement, le standard C/C++ définit comme « vraie » toute valeur (entière) différente de zéro, alors que zéro est équivalent à faux. Si le type `bool` est implémenté, une conversion implicite est faite vers `true` et `false`.

Les opérateurs `&&` et `||` (comme beaucoup d'autres opérateurs) évaluent de gauche à droite. De plus, le compilateur optimise, de sorte que dans le code suivant, la division (illégale) par zéro n'aura jamais lieu dans le cas `d==0`, parce que l'expression à droite du `&&` ne sera jamais évaluée.

```
      d && ( x/d < 10.0);
      i || i++;
```

De même, la deuxième ligne n'incrémentera la variable `i` que si elle est égale à zéro. Dans tous les autres cas, elle gardera sa valeur initiale (lors du passage sur ce code).

Opérateurs binaires

opération	*syntaxe C++*	*résultat*
négation, complément	`~x`	11100110
conjonction	`x&y`	00011001
disjonction	`x|y`	00011011
antivalence	`x^y`	00000010
décalage à gauche	`x<<n`	00110010
décalage à droite	`x>>n`	00001101

Les résultats sont donnés à titre d'exemple pour les valeurs supposés de

x = 00011001, y = 00011011 en notation binaire et n=1.

Ces opérations trouveront peu d'application dans la programmation de programmes d'analyse ou de reconstruction. Cependant, elles sont indispensables dans la programmation de systèmes d'acquisition de données (DAQ) ou de déclenchement (trigger), et aussi dans les algorithmes de compression/décompression par exemple.

Opérateurs d'assignation

relation	*syntaxe C++*	*équivalent*
assignation simple	`x = y`	
… avec addition	`x += y`	`x = x + y`
… avec soustraction	`x -= y`	`x = x - y`
… avec multiplication	`x *= y`	`x = x * y`
… avec division	`x /= y`	`x = x / y`
… du reste de division	`x %= y`	`x = x % y`
… de la valeur décalée (d)	`x >>= n`	`x = x << n`
… de la valeur décalée (g)	`x >>= n`	`x = x >> n`
… avec conjonction	`x &= y`	`x = x & y`
… avec disjonction	`x |= y`	`x = x | y`
… avec antivalence	`x ^= y`	`x = x ^ y`

Précédence des opérateurs

Précédence des opérateurs

- Règles pour l'évaluation d'expressions complexes, comme dans la vie quotidienne (3+4·5 ≠ (3+4)·5)
- `z = a*x + b*y + c;` donne exactement ce qu'on a le droit d'espérer.
- précédences gauche/droite et droite/gauche en plus
- paranthèses imposent précédence, comme d'hab
- La liste complète est à apprendre par cœur pour l'expert, mais il est tout à fait permis de rendre son code lisible pour non-experts aussi (paranthèses) !

Précédence des opérateurs

()	évaluation d'une expression
++ --	post-in/décrémentation
sizeof	taille d'objet / type
++ --	pré-in/décrémentation
~ !	complément, négation
+ -	plus / moins unaire
&	addresse (référence)
*	déréférence
new delete	création (allocation) / destruction
()	cast (conversion implicite
* / %	multiplication, division, reste
+ -	addition, soustraction
<< >>	décalage
< <= > >=	comparaison
== !=	égalité
&	et binaire
^	ou exlusif binaire
|	ou inclusif binaire
&&	et logique
||	ou logique
? :	
= *= /= %= += -= <<= >>= &= |= ^=	toutes les assignations

- Pour vous donner une idée, pas plus !

Exécution conditionnelle

Eléments du langage C++ (2)

- Exécution conditionnelle

```
if (current_temperature > max_safe_temperature) {
     cout << "EMERGENCY! Overtemperature - flushing.\n";
     flushWithWater();
     }
```

- Toute expression rendant une valeur numérique est permise.
- Les accolades sont facultatives, si une seule instruction suit.

```
if (x < 0)
     x = -x; // absolute value, indeed.
     y = -y; // error-prone; always done!
```

if – else ; si – sinon

- Code alternatif

```
if (x < 0) {
     y = -x; }
else {
     y = x; }
```

- C(++) n'a pas de contraintes de format (malheureusement pour certains)

```
if (x < 0) {
     y = -x;
} else {
     y = x; }
```

```
if (x < 0) {y = -x;}
else {y = x;}
```

```
if (x < 0)
     {
     y = -x;
     }
else {
     y = x;
     }
```

- Utilisez votre bon sens !

Opérateur conditionnel

- Une autre manière de dire la même chose …

```
if (x < 0) {
     y = -x; }
else {
     y = x; }
```

⇔

```
y = (x < 0) ? -x : x;
```

- Cet opérateur est un bon exemple pour tomber dans un des deux extrêmes :
  - code concis (plus lisible) et élégant
  - code sur corde raide (source d'erreurs futures), à durée de vie limitée (consommateur d'effort et temps) !

Alternatives multiples

- Les choix multiples sont permis :

```
if (x < 0) {
     y = 1/-x; }
else if (x > 0) {
     y = 1/x; }
else {
     y = 0.; }
```

- Ce n'est qu'un autre format de

```
if (x < 0) {
     y = 1/-x; }
else {
     if (x > 0) {
          y = 1/x; }
     else {
          y = 0.; }
     }
```

(qui, lui, est moins évident à lire).

L'instruction case

- Choix multiples (suite et alternative)

```
if (x == 1) {
      /* code for first case */ }
else if (x == 2) {
      /* second case */ }
else if (x == 3) {
      /* third case */ }
else {
      /* all other cases */ }
```

⇔

```
select (x) {
case 1:
      /* code for first case */
      break;
case 2:
      /* second case */
      break;
case 3:
      /* third case */
      break;
default:
      /* all other cases */
      }
```

- `break` obligatoire (à moins que deux cas doivent aboutir au même endroit)

Boucles (suite)

Vérification en fin de boucle

- Au moins un passage dans le code garanti

```
#include <math.h>

#define ACCURACY 10E-5

main() {

      float a = 2., x = 1.4, dx;

      do {
            float nxt_x = (x + a/x) / 2;
            dx = nxt_x - x;
            x = nxt_x;
            } while (fabs(dx) > ACCURACY);

      cout << "Square root of " << a
                  << " is " << x << ".\n";

      exit(0);
}
```

Méthode de Newton (approx. numérique pour racine carrée) :

La suite $x^{(m+1)} = \frac{1}{2}\left(x^{(m)} + \frac{a}{x^{(m)}}\right)$

converge pour $N \to \infty$

vers la solution $x_N \to \sqrt{a}$

Boucle for

- Une formule très générale d'itération

```
for (début; condition; revalidation) {code}
```

⇔

```
début;
while (condition) {
      code;
      revalidation;
      }
```

- Manière très fréquente d'utilisation :

```
for (int i=0; i<=200; i++) {
      /* loop code here */
      }
```

Interruption de boucles

- `break`, `continue`, `goto`
- `continue` va directement à l'itération suivante (s'il y a lieu).
- `break` sort immédiatement de la boucle.
- `goto` est déconseillé !
- `goto` va à n'importe quel endroit dans le scope ({ }) actuel.
- Souvent, il y a de meilleures alternatives pour ces instructions.

Meilleur continue

```
for (int i=0; i<10; i++) {

      /* some code*/

      if (condition_variable > 0)
            continue;

      /* some more code */

      }
```

⇔

```
for (int i=0; i<10; i++) {

      /* some code*/

      if (condition_variable <= 0) {
            /* some more code */
            }

      }
```

`goto` va à n'importe quel endroit dans la *fonction* actuelle!

Alternative de break

```
for (int i=0; i<10; i++) {
     /* some code*/
     if (cond_var > 0)
          break;
     /* some more code */
     }
```

⇔

```
for (int i=0; i<10 && cond_var <=0; i++) {
     /* some code*/
     if (cond_var <= 0) {
          /* some more code */
          }
     }
```

Utilisation de goto

- **NB:**
  Ceci est un example pour vous décourager d'utiliser `goto` !

```
start:
for (int i=0; i<10; i++) {
     /* some code*/
     if (condition1)
          goto other_loop;
     else if (condition2)
          goto end;
     /* some more code */
     }
other_loop:
int j=20;
while (j-- > 0) {
     /* even more */
     if (condition2) goto start:
     }
end:
```

Tableaux

Tableaux (arrays)

```
float x[100]; // allocation de 100 valeurs à virgule flottante
```

- **accéder le premier élément par `x[0]`**
- **accéder le dernier élément par `x[99]`**
- **Initialisation**

```
float
     x[3] = {1.1, 2.2, 3.3},          /* 3-vector */
     y[] = {1.1, 2.2, 3.3, 4.4};      /* four-vector */
```

Le compilateur peut calculer la dimension.

Tableaux multi-dimensionnels

- **… ou matrices**

```
float m[4][4]; /* no initialisation (random values!) */
float m[2][3] = { /* initialisation is row-wise! */
     {1, 2, 3},
     {4, 5, 6}};
```

- **Multiplication ($M = M_1 \cdot M_2$)**

```
float M[3][3], M1[3][3], M2[3][3];

/* Some code for initialisation of matrices M1, M2*/

double sum;
for (int i = 0; i < 3; i++)
     for (int j; j < 3; j++) {
          M[i][j] = 0.0; // clear result element
          for (k=0; k < 3; k++) // contract index
               M[i][j] += M1[i][k] * M2[k][j];
          }
```

Entrées/sorties

Retour à l'I/O (entrée/sortie)

- **`&&` et `||` évaluent de gauche à droite, et le côté droit peut ne jamais être évalué !**
  - Ceci ne donne jamais division par zéro :

```
return d && (x/d < 10.0);
```

  - D'ailleurs, c'est équivalent à :

```
if(d) {
     if (x/d < 10.0)
          return true; }
else
     return false;
```

Que donne `x/0` ?

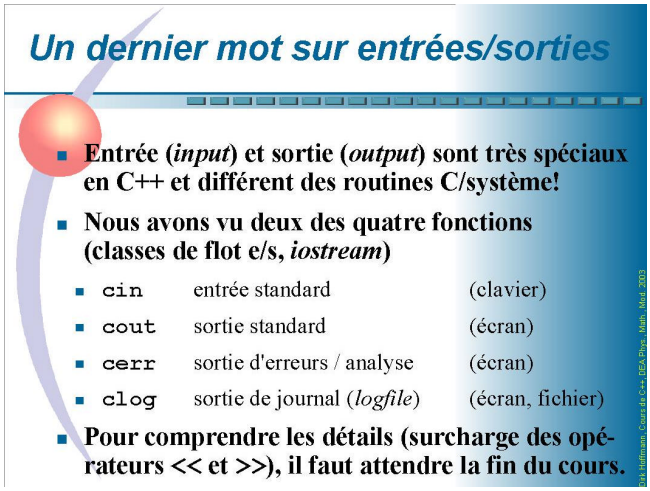
Traitement d'erreurs et sortie

- **La division par zéro génère un signal Unix qui fait avorter le programme. Il s'arrête sur le champ.** *Quel code d'erreur d'ailleurs (`echo $?`) ?*

```
/* Exemple for raising a DIVZERO exception
 * and abort before output is flushed. */

#include <iostream.h>

main() {

     int x = 0, y, n=1000;
     while (n-- > 0)
          cout << "Before we fail, we have"
               << x << ", " << y << ".";
     y = 2/x;
     cout << "This will never be seen.\n";
     exit(0);
}
```

flush
"\n"
endl

Essayez !



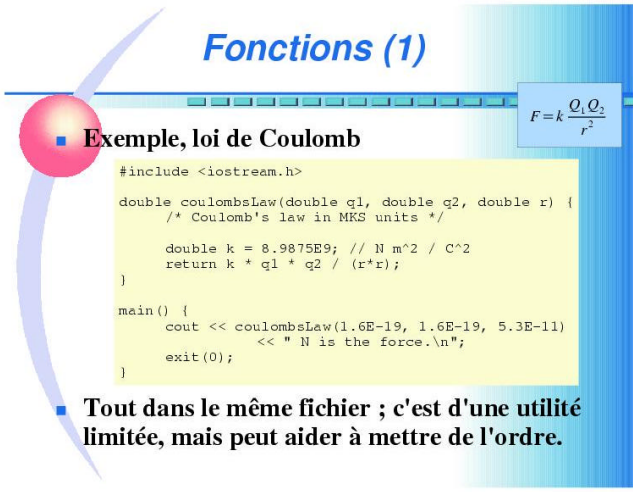
Résumé

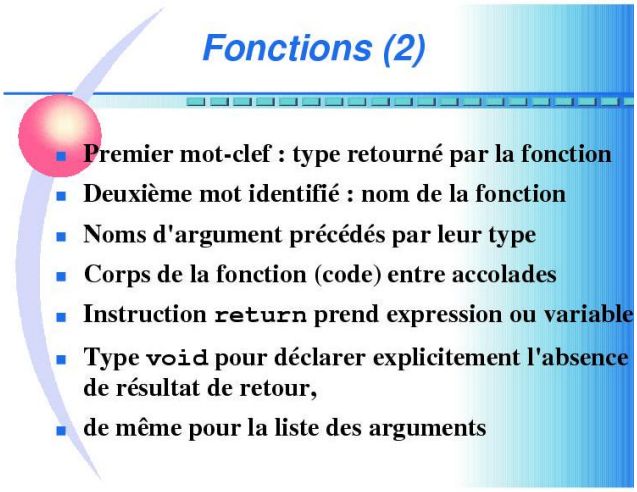
A ce point, les étudiants sont sensés connaître

- la procédure d'édition et compilation d'un programme,
- les éléments de boucle (for, while, do) et d'exécution conditionnelle (if, else),
- les opérateurs
- les fonctions d'entrée et sortie
- les tableaux et matrices (tableaux multi-dimensionnelles)

La partie suivante portera surtout sur deux autres éléments du langage C++ qui sont commun au C, les pointeurs et un approfondissement systématique des fonctions et de leur utilisation.

Fonctions et pointeurs





Librairies et prototypes

```
extern double coulombsLaw(
          double q1, double q2,
          double r){
     double k = 8.9875E9;
     return k * q1 * q2 / (r*r);
}
```

compilateur → coulomb.o

```
#include <iostream.h>
#include <stdlib.h>
int main()
{
     cout << coulombsLaw(1.6E-19,
                1.6E-19, 5.3E-11);
     exit(0);
}
```

vraiment ?

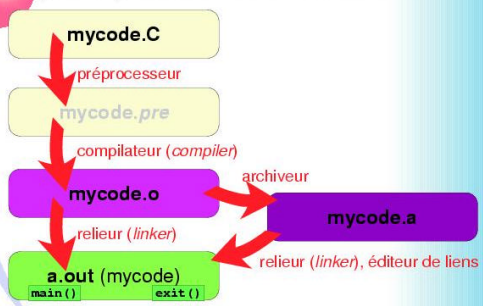
compilateur → main.o

linker (relieur) → a.out (main)

- Séparation des sources pour l'optimisation de la gestion (bibliothèques standard, …)

Source, binaire, bibliothèque, exécutable

- Etapes de production d'un exécutable



L'outil (g)make

- Automatisation de la production par **Makefile**
- Gestion de la recompilation du code selon un schéma :

```
destination: source.C
    →production
    →suite_production
```

- explicitement

```
mycode: mycode.C
    →gcc -o mycode mycode.C
```

- par variables

```
mycode: mycode.C
    →gcc -o $@ $<
```

- règle implicite

```
mycode: mycode.C
```

- source implicite

```
mycode:
```

A vos claviers!

Optimisation de gmake

- Réduction à l'essentiel

```
all: first second third main

second: second.C special.lib

main: main.o coulomb.o
```

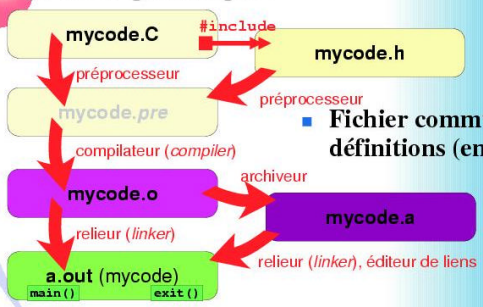
- Génération d'objets par règle implicite (prédéfinie) : `x.o` à partir d'un `x.C`

```
.o.C:
    →gcc -c $<
```

Appliquer sur Coulomb !

Les définitions (en-têtes)

- Qu'est-ce qui manque ?



- Fichier commun des définitions (en-têtes)

Coulomb complet

```
#include <coulomb.h>

extern double coulombsLaw(
          double q1, double q2,
          double r){
     double k = 8.9875E9;
     return k * q1 * q2 / (r*r);
}
```

`coulomb.C` : Implémentation de la fonction

```
extern double coulombsLaw(
          double, double, double);
```

`coulomb.h` : Définitions (communes) permettent de garantir l'accord et usage correct.

```
#include <iostream.h>
#include <coulomb.h>

int main()
{
     cout << coulombsLaw(1.6E-19,
                1.6E-19, 5.3E-11)
          << " N is the force.\n";
     exit(0);
}
```

`main.C` : Code utilisateur

Prototype (signature) de fonction

```
extern double coulombsLaw(double, double, double);
```

- Les noms des variables peuvent être omis.
- C++ vérifie type et nombre des arguments.
- C++ fait la conversion standard, si nécessaire.
- C++ vérifie le type de valeur rendu.
- C++ peut signaler une erreur, si les vérifications échouent.
- C++ permet de forcer l'utilisateur à utiliser la bonne signature pour l'appel d'une fonction.

En-têtes et définitions

- Dans math.h, nous avons les déclarations

```
extern double sqrt(double);
extern double sin(double);
extern double cos(double);
/* and many, many more */
```

- Dans math.C, nous avons les définitions

```
#include <math.h>

double sqrt(double x) {
      /* code */
      return result; }

double sin(double x) {
      /* code */
      return result; }

/* and so on */
```

En-têtes et utilisation

- Dans math.h, nous avons les déclarations

```
extern double sqrt(double);
extern double sin(double);
extern double cos(double);
/* and many, many more */
```

- Dans user.C, nous avons la définition du code utilisateur

```
#include <math.h>

double x, y, z, r;

//

r = sqrt(x*x + y*y + z*z);
```

Directives du préprocesseur

Directives du préprocesseur

- `#include < >` inclut un fichier (système).
- `#include " "` pour fichiers personnels
- `#define VAR xyz` remplace VAR « partout »
- Dans un grand projet, on voudrait éviter les « double-inclusions » (double-définitions!)

```
/* ========== *
 * myfile.h   *
 * ========== */
#ifndef __MYFILE_H__
#define __MYFILE_H__  /* empty! */

     //

#endif/*__MYFILE_H__*/
```

- Tous les commentaires (`/* */`) disparaissent d'ailleurs dans l'étape pré-processeur !

Eléments de C++ : variables globales

- La portée d'une variable se limite par défaut au *scope* ({}), dans lequel elle est déclarée, une fonction, une boucle, un bloc.
- Mais il est possible de la rendre globale (partagée avec tous les objets liés au programme).

```
extern double aNum;

int foo() {
      cout << aNum << "\n";
      exit(0);
}
```

- C'est une manière dangereuse et déconseillée de partager des données, contraire à la POO.

fonctions locales (static)

- Les fonctions déclarées deviennent par défaut globales ; cela peut être indésirable.

```
static double exp_random(double mu) {
      return -mu * log(random());
}

void simulation1() {
      double x1 = exp_random(2.1);
      //
      double x2 = exp_random(1.4);
      //
      return;
}
```

- Le mot-clé `static` signale la portée limitée de la déclaration. Elle est seulement connue à l'intérieur du fichier.

Variables statiques (static)

- Pour les variables, la signification est tout autre!
- Voyez :

```
#include <iostream.h>
#include <stdlib.h>

int counter() {
      static int count = 0;
      return count++;
}

main(){
      int i;
      i = counter();
      count << i << ", ";
      i = counter ();
      exit(0);
}
```

- Les objets statiques retiennent leur valeurs après retour de la fonction.
- Initialisation au chargement du code
- A utiliser avec modération, mais très utile pour encapsulation.

… à ne pas confondre avec const

- Nous avons vu

```
int i = 100, j;
float x = 3.14;
```

- Une déclaration `const`

```
const float e = 2.71828;
const float half_pi = 3.14159/2;
```

  - Une variable `const` ne peut être changée, une fois initialisée.
  - Le compilateur nous en empêche (au lieu de nous laisser faire des bêtises).

```
const float pi = 3.1415;
pi = 3.0; // act of congress
```

- Ceci est obsolète (macro), mais maintenu pour compatibilité.

```
#define M_PI 3.1415;
```

Transitivité des arguments formels

```
void f(int i, float x, float* a) {
      i = 100;
      x = 101.0;
      a[0] = 0.0;
      }

/* --- */

int j = 1;
int k = 2;
float y[] = {3.0, 4.0, 5.0};
f(j, k, y);
```

- Quelle est la valeur de `j` après l'appel de `f()` ?
- C/C++ passent les arguments par valeur (byVal).
- `i`, `x`, `a` sont des arguments formels et valables qu'à l'intérieur de la portée (*scope*) de la fonction `f()`.

Arguments par défaut

- On peut spécifier des arguments par défaut.

```
#include <math.h>

extern double log_of(double x, double base = M_E);
      /* M_E is in math.h */
```

- peut être utilisé comme

```
#include <math.h>
#include "logof.h"

x = log_of(y);      // base e
z = log_of(y, 10); // base 10
```

- Après le premier argument par défaut utilisé, tous les suivants doivent l'être aussi.
- Défaut doit être visible pour appelant.

Récursion

- Une fonction peut s'appeler elle-même.

```
int stirling (int n, int k) {
      if (n<k)
            return 0;
      if (k==0 && n>0)
            return 0;
      if (n==k)
            return 1;

      return k * stirling(n-1, k) + stirling(n-1, k-1);
}
```

- Tout facteur binomial peut se calculer par le biais de deux facteurs « précédents » ($k'<k$, $n'<n$).

$$\binom{n}{k} = k\binom{n-1}{k} + \binom{n-1}{k-1}$$

Implémenter !

Argument const de fonction

```
void f (int n, float x, const float *a) {
      i = 100;
      x = 101.0 * a[0];    // OK
      a[0] = 0.0;          // WRONG!

      return;}

/* --- */

int j=1, k=2;
float y[] = {3.0, 4.0, 5.0};
f(j, k, y);
```

- Un argument const indique à l'utilisateur d'une fonction que ses données ne seront changées en aucun cas.
  - Le compilateur surveille !
  - Ceci est un premier aspect « OO », client/serveur, …

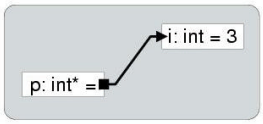
Pointeurs

Les (affreux) pointeurs

- Un pointeur indique l'endroit où se trouve un autre objet
- Déclarations : `int* p; int *q;` Attention : `int* p, r; int *q, *s;`
- Attribuer une valeur :

```
int i=3;
int *p = &i;
```

- lire `&` « adresse de » (dans la mémoire)



Résiliation de pointeur

- `*p` utilise l'objet référencé par le pointeur
- `*p` peut être utilisé des deux côtés du signe =
- Si `p==0`, on l'appelle un pointeur *nul(l)*.

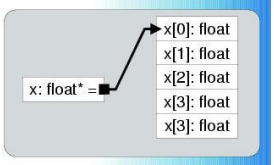
```
#include <iostream.h>
#include <stdlib.h>

main() {
      int* p;
      int j = 4;
      p = &j;

      cout << *p << "\n";

      exit(0);
}
```

- L'utilisation d'un pointeur nul déclenche un core dump, comme toute utilisation d'un pointeur « illégal ».

Vérifiez !

Pointeurs et tableaux

- Nous avions vu : `float x[5];`
- Notre modèle mémoire :
- Que représente `x` ? Pointeur vers le premier élément.
- `*x` ⇔ `x[0]`, `x` ⇔ `&x[0]`
- Les éléments du tableaux peuvent être accédés indifféremment des deux manières.
- Mais `x` est une étiquette vers un tableau, pas un objet pointeur.

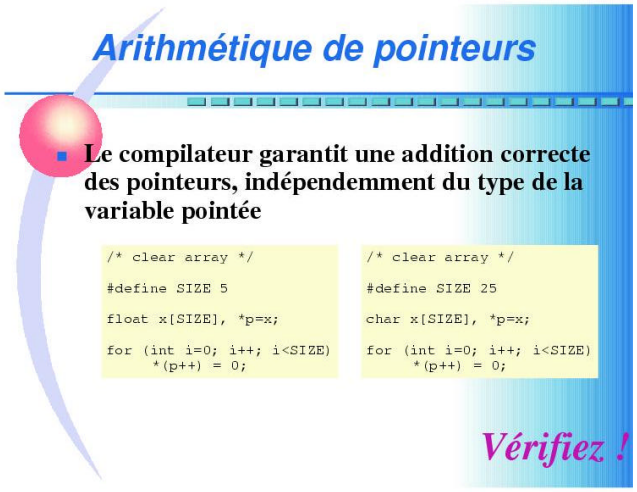


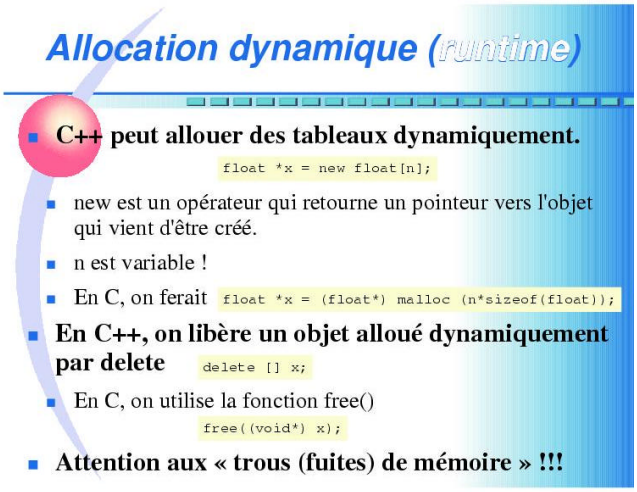
Arithmétique de pointeurs

- Un pointeur peut pointer vers l'élément d'un tableau.

```
float x[5];
float *y = &x[0];
float *z = x
```

- `y` est un pointeur vers `x[0]`, et `z` également.
- `y+1` est un pointeur vers `x[1]`, donc `*(y+1)` et `x[1]` accèdent le même objet.
- y[1] est un raccourci pour *(y+1)
- Addition, soustraction et comparaison sont permis sur les pointeurs.





Résumé

A la fin de cette deuxième grande partie du cours, nous avons traité les éléments de base qui sont communs à presque tous les langages de programmation sous une forme ou une autre. Effectivement, la plûpart des éléments traités sont des éléments plus anciens du langage C. Avec ces éléments, le lecteur est désormais capable de résoudre des problèmes simples de mathématiques numérique ou de programmer des tâches répétitives.

Il nous reste à ce point d'introduire les idées et concepts de la programmation orienté objet (POO), ce qui commencera au prochain paragraphe. Pourtant, il serait exagéré d'espérer de former des développeurs en C++/POO au bout d'une semaine. Après une introduction des principes et la syntaxe de base, le cours portera donc surtout sur l'utilisation efficace de classes fournies, ce qui présente après tout même une grande partie des programmeurs professionnels. Un des buts de ce cours est de faire comprendre le message qu'un code efficacement réutilisé vaut souvent mieux qu'un code réinventé et pas (encore) complet.

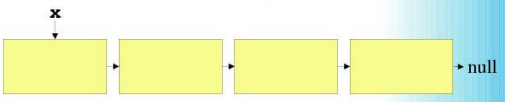
Un langage qui permet de profiter extensivement d'une anthologie de librairies plus que complète est Java qui sera présenté brièvement dans le dernier chapitre avec quelques exemples de base, autant pour sa technologie originale de byte-code, bâtard entre langage compilé et interprété, et sa proximité à la syntaxe du C++.

Encapsulation, objets, classes, méthodes

Idées et concepts de la programmation orientée objet

Programmation Orienté Objet (1)

- **Nous avons vu la manière procédurale de résoudre (implémenter) un problème donné.**
- **Exemple : une liste (de points, de traces)**

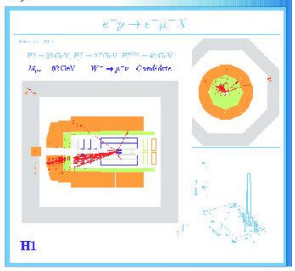


- **Il nous faut des méthodes pour**
  - initialiser, ajouter, enlever
  - répérer le 1er élément, le suivant, …

Les structures C

- **Nous passons rapidement sur ce constituant classique du C, parce que nous en verrons bientôt de meilleures (en C++).**

```
struct trace {
      float theta;
      float phi;
      float radius;
      float dca;
      float zvtx; }

typedef t_trace struct trace;

struct trace t1, t2, t3;

/* init */
t1.theta =0;
t1.phi = 0;
// ...
```



Programmation Orienté Objet (2)

- **Et si on mettait toutes les fonction requises dans le même fichier (avec .h associé) ?**
  - Ça s'appelle modularisation (pas de C++ nécessaire).
- **Et si je cachais même la struct dans mon code, invisible pour l'utilisateur ?**
  - Je dois prévoir des méthodes d'acces (l'interface), pourquoi pas plus confortables aussi (px, py, pz) ?
  - Ça s'appelle abstraction (pas de C++ nécessaire).

Programmation Orienté Objet (3)

- **Finalement, je peux modéliser chaque aspect de mon problème de cette manière (modularisation, abstraction), en classes ou types d'objets, et passer les aspect communs entre différents types par héritage.**
  - C'est l'idée de base de l'orienté objet, et un langage approprié est indispensable pour le faire.
  - Et pour les autres aspects, le compilateur C++ évite bien des erreurs et facilite le contrôle de la discipline de programmation (protection, encapsulation, ...).

Modèles de données OO

- **ADT (*abstract data type*), type abstrait**
  - définit un type et un ensemble d'opérations
  - l'interface est seul moyen d'accès (encapsulation)
- **GDT (*generic data type*) type générique**
  - un type abstrait qui agit sur d'autres types de variables, tout en gardant la même signification semantique, p. e. :
    - une liste de nombres entiers
    - une liste de traces
- **Une instance d'un ADT est un variant spécifique, défini par un paramètre lors de la création.**

Exemple de spécifications

- **Un type abstrait pour GenInteger (ent. pos.) est :**
  - Données
    - séquences de chiffres, nommée *N*
  - Opérations
    - constructeur
    - destructeur
    - add(), nouveau GenInteger qui égale *N*+*k*
    - sub(), de même, *N*–*k*
    - set(), définir la valeur de *N*

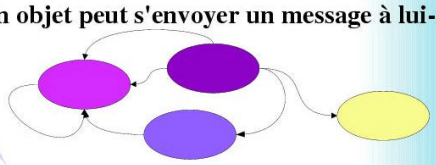
Vocabulaire OO

- **En POO, nous appelons**
  - Implémentation d'un type abstrait : une représentation particulière, utilisant certains langage et notation
  - Classe le type abstrait qui définit les attributs (structure de variables) et méthodes (opérations)
  - Objet l'instance d'une classe, identifiable de manière univoque par son nom (et *scope*)
- **Plusieurs notations sont possible pour manipuler des objets :**

```
i.set(1); // generic notation

i=1;      // more intuitive
```

Programmation Orienté Objet

- **Un programme est une collection dynamique d'objets qui échangent des messages.**
- **Un message est la requête à un objet d'appeler une de ses méthodes.**
- **Un objet peut refuser l'exécution de la méthode.**
- **Un objet peut s'envoyer un message à lui-même.**



Relations OO

- **Admettant que j'ai deux classes**

```
class Point{
      attributes:
            int x, y;
      methods:
            setX(int newX);
            setY(int newY);
      }
```

```
class Circle{
      attributes:
            int x, y;
            int radius;
      methods:
            setX(int newX);
            setY(int newY);
            setRadius(int newR);
      }
```

- **Elles ont en commun des données (x,y), et aussi quelques méthodes (setX, setY).**
- **`Circle` est « une sorte de » `Point`. (is_a)**

Relations OO

- **Une classe peut être composée d'une autre classe.**

```
class Logo{
      attributes:
            Circle a_circle;
            Triangle a_triangle;
      methods:
            set(Point where);
      }
```

- **On dit : `Logo` has_a `Circle`.**

Héritage : dérivation de classes

- **Héritage (*Inheritance*) signifie qu'une classe hérite toutes les propriétés d'une autre classe.**

```
class Circle inherits from Point {
      attributes:
            int radius;
      methods:
            setRadius(int newR);
      }
```

…tout le reste est « hérité ».

- **`Circle` peut utiliser les méthodes de `Point` de manière transparente.**

```
Circle myCircle;

myCircle.setX(2);
myCircle.setRadious(12);
```

Héritage (2)

- **Si nous définissons comment déplacer un `Point`**

```
class Point {
      //
      methods:
            moveX(int deltaX) {
                  x += deltaX; }
      //
      }
```

- **nous savons immédiatement déplacer un `Circle`.**

```
Circle myCircle;

myCircle.move(10);
```

- **La gestion du code devient plus efficace!**
- **Les « héritiers » sont appelés sous-classes.**

Classes abstraites

- **Une classe abstraite**
  - force ses sous-classes à redéfinir certaines propriétés
  - et aide donc à la structuration des classes.
  - Ne peut être instanciée !

```
abstract class Drawable {
      attributes:
      methods:
            paint();
      }
```

```
class Point : Drawable {
     attributes:
          int x, y;
     methods:
          setX(int newX);
          setY(int newY);
          print(); // won't work otherwise!
     }
```

Implémentation d'une classe

Classe simpliste

- **Vecteur en 3 dimensions**
  (~hoffmann/1ereClasse.txt)

```
class Vector3D {
    public:
        Vector3D();
        Vector3D(double x, double y, double z);
        void print();
    private:
        double x, y, z;
    };

Vector3D::Vector3D() {
    x=1;
    y=1;
    z=1;}

Vector3D::Vector3D(double iniX, double iniY, double iniZ) {
    x = iniX;
    y = iniY;
    z = iniZ;
    }

void Vector3D::print() {
    cout << "Vector3D = (" << x << ", " << y << ", " << z << ");
    }

main() {

    Vector3D* v1 = new Vector3D(1, 2, 3);
    Vector3D  v2;

    v1->print();
    cout << endl;
    v2.print();
    cout << endl;

    delete v1;

    exit(0);
    }
```

A vos claviers!

Syntaxe pour une classe C++

- **Déclaration :**

```
class laClasse {
     private:
          données;
     public:
          méthodes(en_général);
};
```

- **Implémentation :**
  - **`laClasse::élément`** pour désigner le contenu
- **Utilisation :**
  - **`laClasse.élément`** ou **`laClasse.méthode()`** pour accéder le contenu

Instanciement en C++

- **Instance implicite (et son nettoyage)**

```
laClasse c;
```

- **Création explicite (nécessite nettoyage explicite!)**

```
laClasse *c;
c = new laClasse();

delete c;
```

Extensions de la classe

- **Rayon, orientation (angles)**
- **Addition, soustraction, rotation**
- ***autres suggestions ?***

A vos claviers!

Surcharge (redéfinition) d'opérateur

Surcharge d'opérateur

- **Considérons un opérateur comme une fonction**

```
double add(double a, double b) {
      return a+b; }
 //
double x, y, z;
z = x + y;
z = add(x, y);
```

- **En C++, nous avons**

```
z = x.operator+(y);
```

  - **`add(❑,❑)`** agit sur deux opérands, ainsi que **❑ + ❑**.
  - Opérateurs : plus concis, plus lisible

```
z = add(mul(a,x), mul(b,y);
z = a*x + b*y;
```

Surcharge d'opérateur

- **Nous redéfinissons un opérateur en C++ ainsi**

```
class Vector3D {
      public:
      double Vector3D& operator* (const Vector3D&);
      }
```

  - Le nom est **`operator`**, suivi par le symbole.

```
double Vector3D::operator* (const Vector3D& v){
      double result = x * v.x + y * v.y + z * v.z;
      return result;
      }
```

  - Il faut utiliser des références (pas encore vu!) pour ce type de méthodes!

Deuxième projet sérieux

- **Une classe « nombre entier illimité »**
  - contenant toutes les opérations permises sur
- **Vous voulez probablement utiliser quelque chose comme**

```
unsigned char n* = new char[10];

delete [] n;
```

- **Et vous avez jusqu'à vendredi matin.**

Fin du troisième cours

- **Qu'est-ce que nous avons appris ?**
  - Idées et concepts de la programmation OO
  - Première classe
- **Qu'est-ce qui suivra ?**
  - Exemples de classes, et surtout de librairies (génération de graphes de Feynman, CLHEP, …)
  - Vue sur Java | Vue sur les détails de C++

La programmation orienté objet, Finesses et exemples du C++

… à la fin du troisième cours

- **Qu'est-ce que nous avons appris ?**
  - Idées et concepts de la programmation OO
  - Première classe et gros problème
- **Qu'est-ce qui suivra (aujourd'hui) ?**
  - Notre problème et rappels de base sur les classes
  - L'outil ROOT
  - CLHEP, STL
  - La bibliothèque FeynDiagram
  - Vue sur Java

Ajout sur l'accès des éléments

- **Il existe une équivalence utile pour accéder les éléments d'une `class` (ou `struct`) dont on tient le pointeur :**

`(*X).y ⇔ X->y`

```
class dummy {
      //
      public:
            int a, b, c;
            void function();
      };

dummy d1;
dummy* d2 = new dummy;

d1.function();
(*d2).function();

/* now the new one: */
d2->function();
```

Le problème (revu) …

- **Une classe « nombre entier illimité »**
  - contenant les opérations **+** et *****, ainsi qu'un moyen de définir une valeur (**=**) et d'imprimer le résultat.
- **Vous voulez probablement utiliser quelque chose comme** («allocation dynamique»)

```
unsigned char n* = new char[10];
delete [] n;
```

- **Et je fais grâce de la redéfinition des opérateurs (parce que vous voudriez les implémenter de vous même, je suppose ;-)**
- **Et vous avez jusqu'à …**

Exemple simpliste : le vide (actif), sous-classe, héritage

Création de (classe) vide

- Démonstration de l'instanciement, du constructeur et du déstructeur (exemple pédagogique) :

```
class VideActif {
      private:
            int numero_;
      public:
            VideActif(const int id):numero_(id) { // constructor!
                  cout << "Création VideActif"<<numero_<<"\n"; }
            ~VideActif() {            // destructor!!
                  cout << "Déstruction VideActif"<<numero_<<"\n"; }

main() {
      VideActif v(1);
      {
            VideActif w(2);
            v.numero();
            w.numero();
      }
      v.numero();
      exit(0);    // essayez de remplacer par "return"!
      }
```

A vos claviers!

Sous-classe, héritage

- **`DemiPlein` est dérivé de `VideActif` et hérite par défaut toutes ses propriétés .**

```
class DemiPlein {
      private:
            int ident_;
      public:
            DemiPlein(const int id): VideActif(), ident_(id) {
                  ident_ = id;
                  cout << "Création VideActif"<<numero_<<"\n"; }
            ~VideActif() {            // destructor!!
                  cout << "Déstruction VideActif"<<numero_<<"\n"; }
      };

//

      DemiPlein d(3);
      DemiPlein p* = new DemiPlein(4);
      delete p;
```

Compris ?

Héritage à l'envers ?

- **Quand le carré est un rectangle, ou pas.**
- Implémentation :

```
class Rectangle {
      // ...
      void setLength(float);
      void setHeight(float);
      // ...
      float length, height;
      };
```

- Que serait un carré par rapport à cette classe ?
  - En géométrie, les carrés sont un sous-ensemble des rectangles.
  - En programmation, nous avons besoin d'un paramètre en moins !
- **Quand je dois enlever des paramètres, mon hiérarchie (d'héritage) n'est pas bonne !**

Héritage multiple

- **L'héritage multiple (deux classes modèles) est autorisé**

```
class Rectangle : public Square, public Drawable {
      // ...
      };
```

- **Comme les conflits (entre variables ou fonctions similaires ou égales) sont probables, réfléchissez (et réfléchissez encore), si la relation souhaitée est vraiment «is_a» et non pas «has_a».**

Initialisation d'environnements particuliers – ini

Outre les chargement et installation explicites des paquets de logiciels comme ROOT et Java, beaucoup de sites offrent soit la mise en disposition forcée (pour tous les utilisateurs), soit une activation facile (au choix de chaque utilisateur) des différents environnements pré-installés (et donc testés). Cette dernière possibilité se présente une partie des machines du CPPM, sous forme de l'outil ini. Pour pouvoir l'utiliser, il faut dans un premier temps

- soit exécuter le fichier `/marscotte/dea/cshrc` par la commande
  ```
  source /marscotte/dea/cshrc
  ```
- soit provoquer l'exécution automatique lors du login en copiant ce fichier par
  ```
  cp /marscotte/dea/cshrc ~/.cshrc
  ```
  dans la liste des commandes de login personnel.

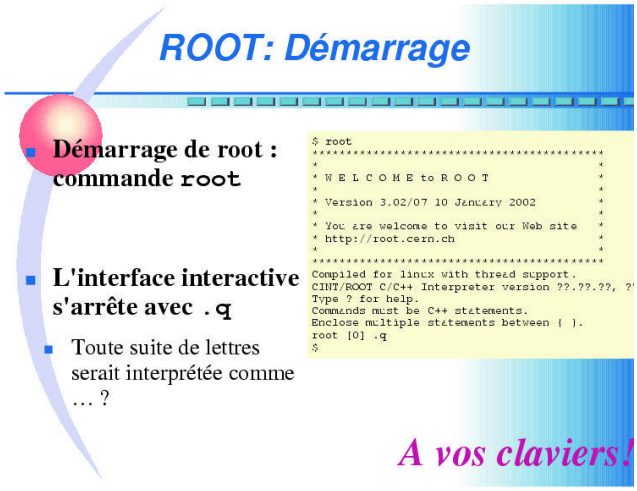
Il est ensuite possible d'appeler la commande `ini` dans un shell (ce qui affichera par défaut toutes les options disponibles), ou bien d'initialiser dans le cadre de ce cours les deux environnements

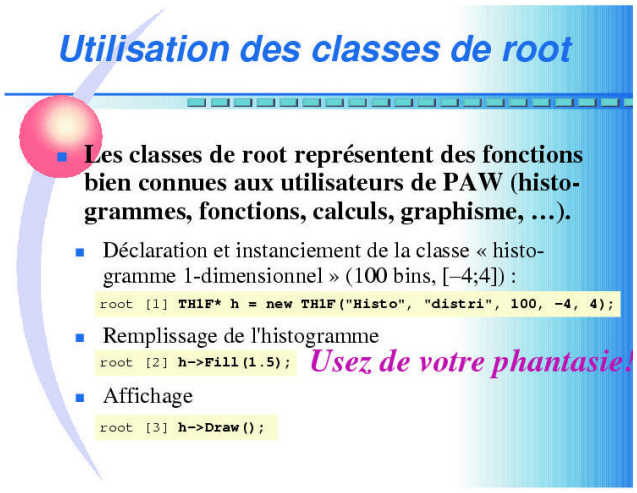
- ROOT avec `ini root` et
- Java avec `ini jdk4`.

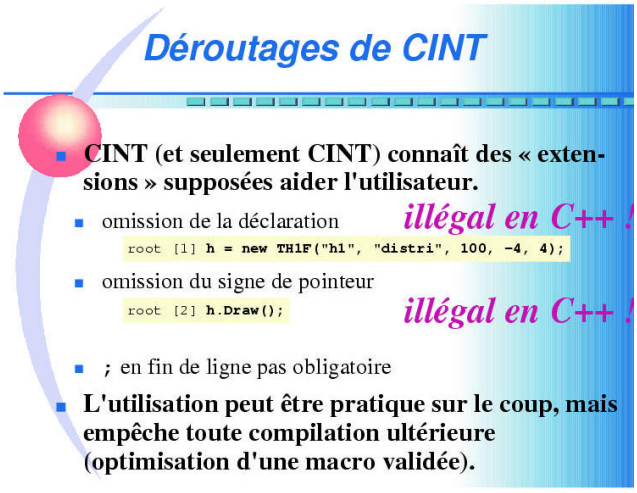
Exemple d'application : L'environnement ROOT

Un groupe de programmateur a développé l'outil ROOT pour l'analyse et la visualisation de données au CERN, qui est gratuitement mis à la disposition des chercheurs en physique des particules et quelques autres domaines. Il est programmé en C++ et l'utilisateur contrôle le fonctionnement interactif à travers un langage quasi-C++, l'interpréteur CINT. Outre la compilation de séquences interactives répéptées, ce logiciel et modularisé et toutes ses librairies peuvent être réutilisées dans des logiciels individuels et indépendants. Comme ROOT est à ce jour devenu un outil standard d'analyse dans de nombreuses expériences comme H1, DØ, BaBar et au LHC à côté de JAS (Java Analysis Studio), la probabilité pour les étudiants est grande d'être ammenés un jour ou un autre à l'utiliser.

La commande ini évoquée plus haut s'occupera d'ajuster les variables $PATH, $ROOTSYS et $LD_LIBRARY_PATH correctement, afin de pouvoir utiliser l'environnement ROOT pré-installé. Le site de téléchargement (et de documentation d'ailleurs) serait `http://root.cern.ch`.







Macros et compilation ROOT

- **Plusieurs lignes peuvent être exécutées en mode script (commande `.x`) :**
  - Si mymacro.C contient

```
{
      TH1F* h = new TH1F("Histo", "distri", 100, -4, 4);

      for (int=0; i<50; i++) {
            h->Fill(i/60);
            }

      h->Draw();
}
```

  - la commande `root [1] .x mymacro.C` produira l'image attendue (ou presque ?).

Corrigez l'erreur !

Macros ROOT (suite)

- **Avec ces quelques outils en main, on arrive presque à fournir un travail scientifique complet** *(en ce qui concerne l'aspect visualisation)*.

```
      for (int=0; i<50; i++) {
            float x = (float)i*i/50000;
            h->Fill(x);
            h->Fill(1-x);
            }

      h->Draw();
```

Jouez !

Un exemple presque inutile

- **Essayez**

```
root [1] TpaveLabel hello(0.2, 0.4, 0.8, 0.6, "A vos ordres");
root [2] hello.Draw();
```

A vos claviers!

- **Vous avez vu la différence ?**

Vraiment ?

Macros compilées

- **Si le code se conforme au standard,**

```
int mymacro()
{
      TH1F* h = new TH1F("Histo", "distri", 100, -4, 4);

      for (int=0; i<50; i++) {
            h->Fill(i/60);
            }

      h->Draw();

      return 0;
}
```

- **il peut être «intégré» ou pré-compilé (performance!)**

```
root [1] .L mymacro.C
root [2] mymacro()
```

```
root [1] .X mymacro.C
root [2] mymacro()
```

Vous avez vu la valeur de retour ?

Générateurs aléatoires

- **ROOT contient également des classes pour générer des nombres aléatoires**

```
TH1F* h = new TH1F("Histo", "distri", 100, -4, 4);

gRandom->SetSeed();

for (int=0; i<50; i++) {
      h->Fill(gRandom->Gaus(-1, 1.5));
      }

h->Draw();
```

Et ainsi de suite …

Modèles de types – templates et la STL

Un élément très puissant du langage C++ est le template (modèle, schéma). L'idée derrière ce mécanisme avec une syntaxe qui peut sembler abstraite et compliquée est de pouvoir écrire du code qui sera réutilisable non seulement pour un type de variable (voire même un type de classe) donné, mais pour toute classe, à condition qu'elle contienne les opérations utilisées dans le modèle de code. Pour cela, il n'est même pas nécessaire que de futurs candidats de classe soient défini avant l'écriture du code-modèle qu'on réutilise avec elles.

Standard Template Library

- « Don't (re)invent the wheel, use STL. »
- La STL est implémenté dans beaucoup de distributions C++ aujourd'hui et fait objet d'un standard ISO.
- Elle contient de nombreux méthodes et classes « prêt à porter », (conteneurs, algorithmes, maniement de chaînes de caractères.
- Utilisées (donc testées) par un grand nombre de programmateurs. C'est un produit mûr.
- Nous allons voir une très petite partie de cette panoplie, les conteneurs.

Modèles de classe

- Calculer le carré d'une variable

```
double sqr(double x) {return x*x ;}
```

- Calculer le carré (self-produit) de *quelque chose*

```
template<class T>
     T sqr(T x) {return x*x ;}
```

- Nous pouvons ainsi faire

```
int i=1;
float f=3.1;
Hep3Vector v(1,1,1);

cout << sqr(i) << "\n";
cout << sqr(f) << "\n";
cout << sqr(v) << "\n";
```

- Mais que signifie $\vec{v}\cdot\vec{v}$ au juste ?
- Donne-t-il un vecteur ? *Attention!*

Essayez quand même !

Exemple : STL vector

- Un modèle de « vecteur » est implémenté sous le nom `vector`.
- Création (instance)

```
vector<T> coll;
```

- Ajout d'éléments

```
coll.push_back(T);
```

- Taille

```
coll.push_back(T);
```

- Accès *(comment ??)*

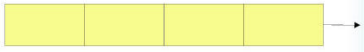
```
coll[i];
```

```
#include <iostream.h>
#include <vector.h>

main() {
      vector<int> coll;

      for (int i=0; i<=6; i++)
            coll.push_back(i*i);

      for (int i=0; i<coll.size(); i++)
            cout << coll[i] << ' ';

      cout << "\n";

      exit(0);
}
```

A vos claviers!

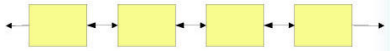
Modèles de collection

- Vecteur



- Deque (« dèque », *double ended queue*)

- Liste



Facile ?

- Rappelez-vous de la propriété de la récursion.

```
#include <iostream.h>

int factorial(int x) {

      // ... ?
}


main() {

      for (int=0; i<20; i++)
            cout << i << "! = " << factorial(i) << "\n";

      return 0;
}
```

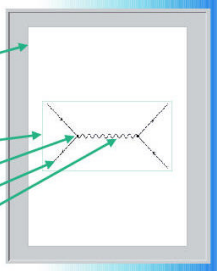
A vos claviers!

En considérant ces possibilités, les étudiants vont commencer à voir de nombreuses améliorations et surtout simplifications de code possibles, dans les problèmes posés ultérieurement (nombres premiers et définition d'une classe de nombre à précision quasi-illimitée). Elles sont à implémenter en exercice, en dehors du cours.

Dessin des graphes de Feynman – FeynDiagram

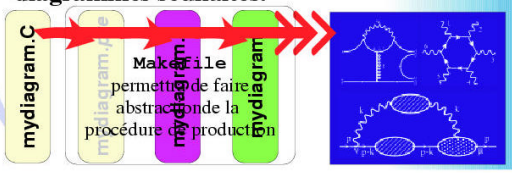
Description d'un graphe en OO

- Interaction e^- e^+ par photon au premier ordre
- Une instance de `page`
- contient une ou plusieurs instances de `FeynDiagram`
- qui contiennent des instances
  - `vertex_dot`
  - `line_plain`
  - `line_wiggle`



FeynDiagram (fine diagram)

- Une bibliothèque C++ a été écrite par Bill Dimm (et al.) pour dessiner des diagrammes de Feynman (`http://feyndiagram.com`).
- Un fichier source C++ est compilé et utilisé pour produire des fichiers PostScript avec les diagrammes souhaités.



La compilation

- Il faut indiquer l'endroit des fichiers en-tête
  - -I/marcotte/dea/FD/include
- et l'endroit et le nom de la librairie compilée
  - -L/marcotte/dea/FD/lib -lFD

```
g++ -I/marcotte/dea/FD/include -L/marcotte/dea/FD/lib -lFD mydiag.C
```

- Et la sortie ?

A vos claviers !

```
a.out > mydiag.ps
gv mydiag.ps
```

- visualisation avec Ghostview (`gv`)

Description du diagramme en C++

- Des classes supplémentaires permettent d'organiser la géométrie (`xy` pour points 2D).
- Appel final à `pg.output()`

```
#include <FeynDiagram/fd.h>

main() {
      page pg;
      page::prologdir="/marcotte/dea/FD/Prolog/";

      FeynDiagram fd(pg);

      xy e1(-10,5), e2(-10,-5), e3(10,5), e4(10,-5);
      vertex_dot v1(fd,-5,0), v2(fd,5,0);

      line_plain f1(fd,e1,v1);
      line_plain f2(fd,v1,e2);
      line_wiggle photon(fd,v1,v2);
      line_plain f3(fd,v2,e3), f4(fd,e4,v2);

      pg.output();
      return 0;
}
```

A vos claviers !

Fin de la partie C++

Nous sommes arrivés à la fin de la partie « C++ » du cours. Désormais, le lecteur connaîtra :

- les bases de la programmation, et en particulier les paradigmes de la programmation « orienté objet »,
- les mécanismes de compilation d'un ou plusieurs fichiers source afin de produire un fichier exécutable correspondant qui peut
  - résoudre un problème mathématique numérique simple à l'aide d'un logiciel informatique,
- les langages C++ et dans une certaine mesure aussi le C, suffisamment pour
  - pouvoir s'en servir en tant qu'utilisateur et
  - pouvoir modifier les programmes fournies, après les avoir compris,
- les environnements et librairies
  - ROOT
  - FeynDiagram et
  - Standard Template Library (STL)
- pour pouvoir s'en servir de manière extensive et extensible par étude de la documentation.

Ce cours est complété par une introduction brève à Java en tant que langage très similaire du C++, au point de pouvoir confondre la plupart de leur syntaxe. En vue du champ d'application croissant de Java dans tous les domaines, il semble indispensable de faire cette excursion à ce jeune cousin du C++ dans le cadre d'une introduction générale à la programmation OO.

Introduction à Java – classes, compilation, machine virtuelle

La machine virtuelle

- **Java est un langage « bâtard » entre les binaires compilés (`a.out`) et les langages interprétés (shell).**
  - Une machine virtuelle est nécessaire pour l'exécution du byte code (`*.class`).
  - Elle garantit une portabilité maximale.

Numéros de version

- **Sun a crée une grande confusion avec son choix (ou plutôt sa non-décision) de numérotation.**
  - Java 1.0 était la premiere version, centrée sur le WWW.
  - Java 1.1 était la première grande amélioration en 1997
  - Java 2 comprenait la réécriture de parties importantes du code Java, en particulier l'interface graphique, et ajoutait beaucoup d'éléments utiles comme l'accès aux bases de données. **La version correspondante du SDK est 1.2!**
  - Java 1.3 (ou « Java 2 avec SDK 1.3 » officiellement), sortie en 2000, et Java 1.4 depuis 2003, sont les version actuelles, et le développement a lieu sur 1.4β2.

Pour pouvoir utiliser l'environnement Java (JDK, Java Development Kit) sur les machines du CPPM, on peut (outre le téléchargement et l'installation de la version la plus récente) utiliser l'outil ini présenté dans le chapitre précédent : `ini jdk4` initialise la version la plus récente de la famille JDK 1.4.x.On rappelle que `ini` tout court donne la liste complète des options installées.

Retour aux bases

- **Tout compilation résulte en une classe.**

```
class Trivial {

     public static void main(String[]) {

          System.out.println("A vos ordres, maître");

          System.exit(0);
          }
}
```

- **`javac` est la commande de compilation**
- **`java` charge la machine virtuelle.**

Retour aux bases

- **Toute compilation résulte en une classe.**

```
class Trivial {

     public static void main(String[] Argv) {

          System.out.println("A vos ordres, maître");

          System.exit(0);
          }
}
```

- **`javac` est la commande de compilation.**
  `javac Trivial.java`
- **`java` charge la machine virtuelle.**
  `java Trivial A B C` *A vos claviers!*
  (pas `Trival.class` !) execute
  `Trivial.main({"A","B","C")`

Types de variables

- **Quelques vieux connus**
  - ***`byte`*** 8 bits
  - **`short`** 16 bits
  - **`int`** 32 bits
  - **`long`** 64 bits
  - **`float`** 32 bits
  - **`double`** 64 bits, les deux conformes IEEE 754
  - Mais ***`char`*** est 16 bits (Unicode)!
- **Garantis dans n'importe quelle JVM !**

Constantes littéraires

`123`	**`int`**
`1234l, 1234L`	**`long`**
`073`	**octal**
`0xABC`	**hexadécimal**
`5.7777`	**`double`**
`2.56F, 3.14f`	**`float`**
`true, false`	***`boolean`***

Opérateurs

- De vieux connus

```
+ - * / % =
+= -= *= /= ++ --
== != > < <= >=
& && | || ^ >> << ~
<<= >>= &= |= ^=
```

- font tous ce que vous espérez.
- <<< et >>> complètent (décalage et remplissage par 0).
- Précédence comme en C/C++ !

Pointeurs

- Y a pas.

– au moins pas visible pour l'utilisateur.

- Tout passage d'argument « BY_REF »
- La collection des objets qui ne sont plus utilisés, se fait automatiquement! *(garbage collection)*
- La déclaration d'une variable implique sa création automatique seulement pour les variables génériques.

Avantage de constructeurs multiples

- Déclaration et utilisation des fonctions de librairie font une (`import`), suite au byte-code!

```
import java.util.Date;

class DateTester {

     public static void main(String[] Argv) {
          Date d1, d2, d3;

          d1 = new Date;
          System.out.println("Date 1: " + d1);

          d2 = new Date(71, 7, 1, 7, 30);
          System.out.println("Date 2: " + d2);

          d3 = new Date("April 3 1993 3:24 PM");
          System.out.println("Date 3: " + d3);
          }
}
```

A vos claviers!

Définition d'une classe

- Equivalent au C++

```
class myClass {

     public static void main(String[] Argv) {
          System.out.println("This is an autonomous class.");
          }

     private void reservedFunction() {
          //
          }

     private int secretData;
}
```

- Commentaires : comme C++, `/*  */` `//`

Chaînes de caractères

- Beaucoup plus de facilités (et internationalisation !)

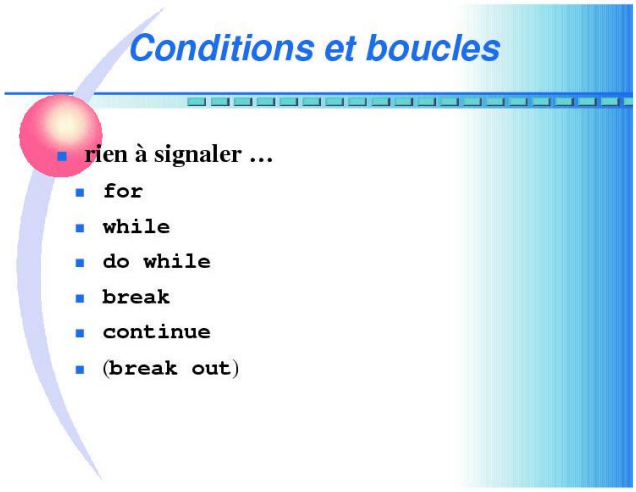
```
String s = "Initialisé";
int i = 42;

System.out.println("1ère partie " + "2ème partie " + i);
```

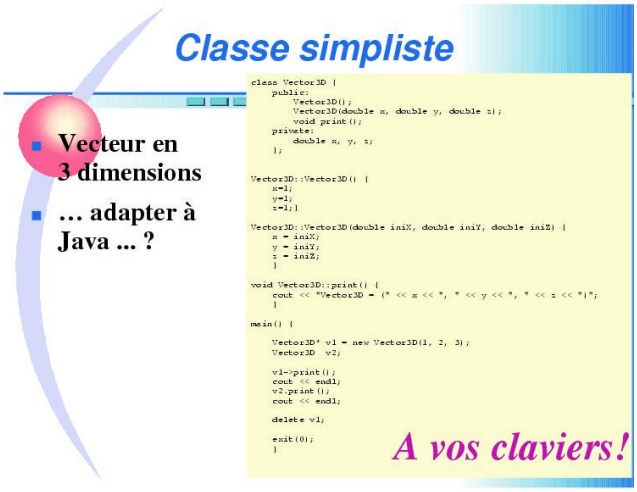
- Conversion implicite facile (recherche de méthode `toString()`)
- Un `String` n'est plus un tableaux de `char`!

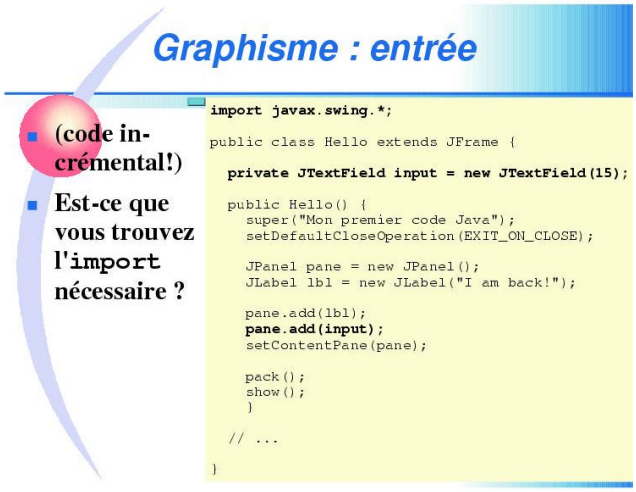
Tableaux

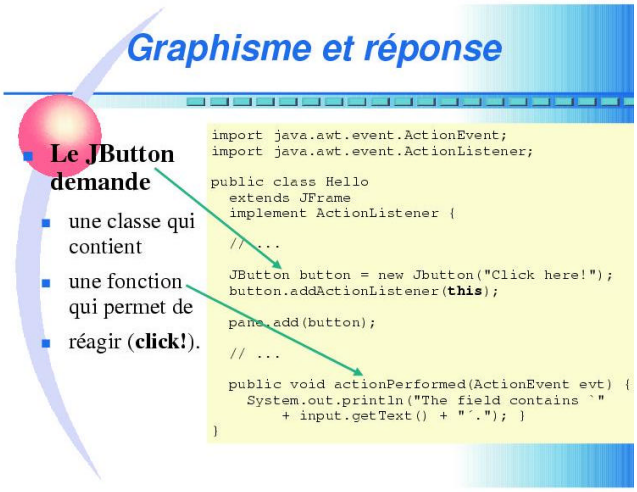
- Comme en C++, quelques relâches syntactiques

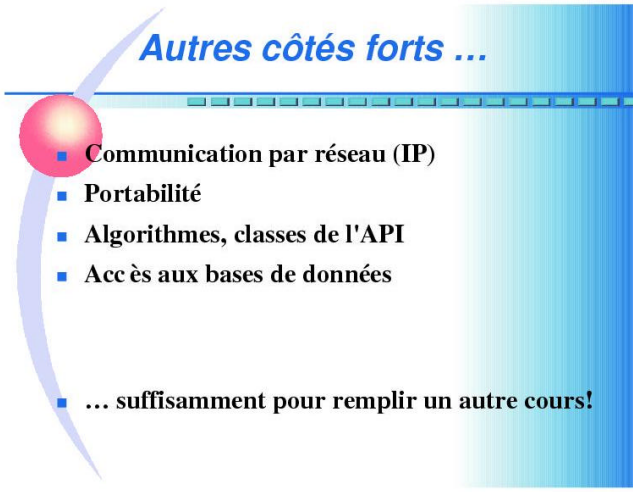
```
int[] temp = new int[99];
int other_temp[] = new int[999];

String[] names = new String[10];
```











Résumé

Arrivés à la fin de la partie sur Java, il convient de faire un petit point. Nous avons vu que Java est un langage OO, facile à utiliser, et qui ressemble beaucoup au C++ du point de vue syntactique. Nous avons vu quelques exemples simples et les blocs élémentaires pour écrire un programme utile.

Pour continuer l'apprentissage Java, il reste à voir une multitude de classes et leur méthodes dans les bibliothèques Java. Une bonne documentation (appelée communement des « bibles » à cause de l'épaisseur typique des descriptions de bibliothèques) est en général indispensable pour le développement efficace en Java.