SCIENCES SORBONNE UNIVERSITÉ

Programmation Système Concurrente et Répartie
Master 1 Informatique – MU4IN400 – PSCR

Cours 1 : Introduction au C++

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Organisation

- 2h cours, 2h TD, 2h TME par semaine
- Examen réparti 1 (novembre) sur machines
- Examen réparti 2 (janvier) sur feuille
- Répartition : 10% note de TME, 30% Exam 1, 60% Exam 2.

- Attention : refonte des supports, nouvelle version de l'UE
- Passage au C++ !

- Emplois du temps : https://cal.ufr-info-p6.jussieu.fr/master/

Plan des Séances

Objectif : principes de programmation d'applications concurrentes et réparties

1. Introduction C++, modèle mémoire, modèle objet
2. Opérateurs, classes, surcharge
3. Conteneurs, Itérateurs, lib standard
4. Programmation concurrente : threads, mutex
5. Synchronisations : conditions, partage mémoire
6. Multi-processus : fork, exec, signal
7. Communications interprocessus (IPC) : shm, sem, pipe, …
8. Communications distantes : Sockets
9. Protocoles de communication : Sérialisation, protobuf
10. Parallélisme grain fin : lock-free, work steal

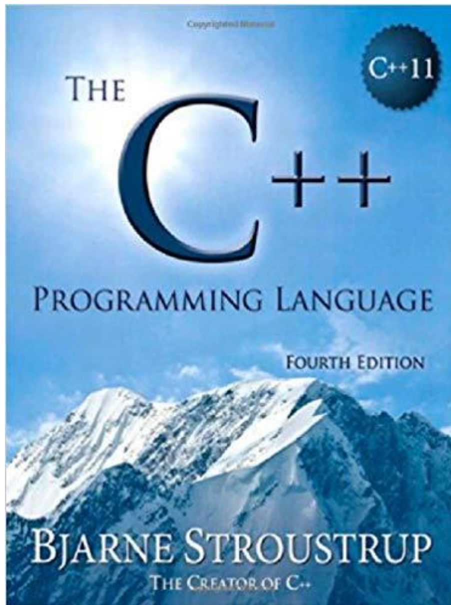
Ouverture : introduction à MPI, CUDA

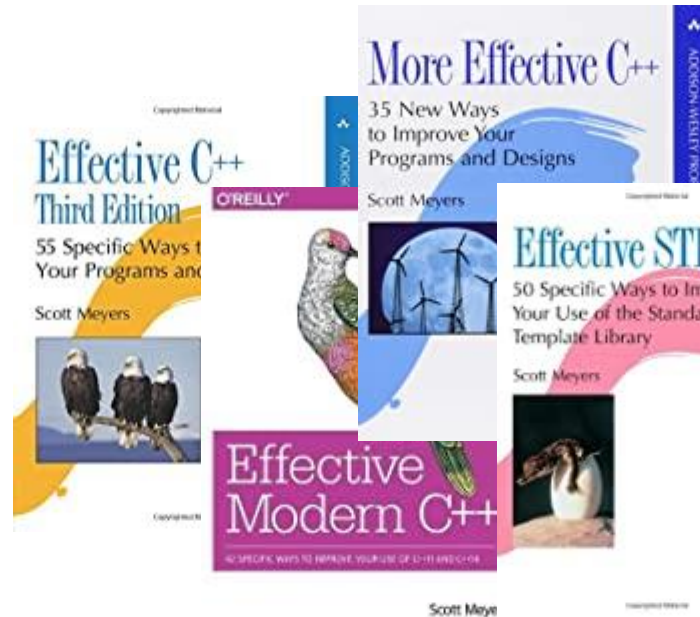
Références

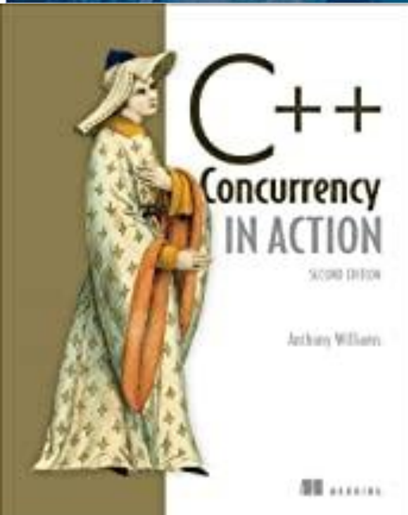
- Hypothèse / pré-requis :
  - Niveau intermédiaire en C,
  - Niveau confirmé en Java (POBJ, PRC en L3 ?)
- Références web :
  - http://www.cplusplus.com/ (tutos, docs de références)
  - https://cppreference.com/ (des parties traduites, des parties en français)
  - Une version en ligne du man : https://man.cx/
  - StackOverflow : https://stackoverflow.com/
- Références biblio utiles :
  - Stroustrup, « The C++ Programming Language », 4th Ed (C++11)
  - Gamma, Helm, Vlissides, Johnson, « Design Patterns »
  - Meyers, « Effective XXX » XXX=C++, STL, modern C++
- Cours en partie basé sur les supports de
  - Denis Poitrenaud (P5), et Souheib Baarir (P10) sur le c++
  - L. Arantes, P.Sens (P6) sur Posix

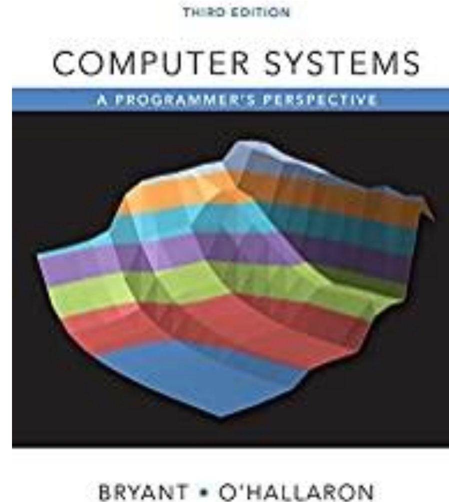
Références : Livres

Ne pas hésiter à lire !









Langage C++

C++ vs C

- C++ est un sur-ensemble du C98 standard
  - Types de données : char, int, long, float, double …+ unsigned
  - Tableaux, struct, pointeurs
  - Structures de contrôle : if/then/else, for (i=0 ; i<N; i++), while, do/while, switch/case
  - Fonctions, paramètres typés, sémantique mémoire stack/heap
- En plus on trouve :
  - Namespaces, visibilités
  - Type référence, type const
  - Classes, instances, orientation objet (héritage)
  - Polymorphisme très riche
  - Redéfinition d'opérateurs : +, *, &&, =, …
  - Généricité via templates
  - Librairie standard riche, librairies C++ efficaces
  - Lambda, inférence de type

C++ vs Java

- C++ partage avec Java
  - Les concepts d'orienté objet : classe, instance, héritage
  - Beaucoup de la syntaxe
- C++ offre en plus/moins
  - Pas de garbage collector, pas de classe parente Object
  - Accès fin à la mémoire / Gestion mémoire plus difficile
  - Généricité via la substitution vs « erased types »
- C++ offre en plus
  - Interaction immédiate avec les API noyau, devices, matériel
  - Efficacité mémoire accrue (~x10), performances
- Java offre en plus
  - Réflexion, introspection, dynamicité
  - Modèle sémantique uniforme et simple
  - Lib standard et non standard très étendue

Le Langage C++

- Sur-ensemble du C : gestion fine de la mémoire, du matériel
- Langage compilé : très efficace, pas de runtime
- Langage orienté objet : structuration des applications, variabilité
- Langage moderne en rapide évolution : C++11, 14, 17 et bientôt 20
- Fort support industriel : intel, microsoft, jeux…
- S'interface bien avec des langages front-end come Python
- Langage très versatile et puissant

Mais

- Langage relativement complexe, beaucoup de concepts

- Inutile de maîtriser tout le langage pour l'UE, deux séances pour apprendre la syntaxe et l'environnement + concepts plus avancés abordés au fil des séances

Du C au C++

Compilation, Link, Exécutable, Librairie…

C++ : un sur-ensemble du C

CPoint.h

```
#pragma once
struct Cpoint {
    double x;
    double y;
};

void init(CPoint* p, double x, double y);
double distOri(CPoint *p);
```

CPoint.cpp

```
#include "CPoint.h"
#include <cmath>

void init(CPoint *p, double x, double y)
{
        p->x = x;
        p->y = y;
}

double distOri(CPoint *p)
{
        return sqrt(p->x * p->x + p->y * p->y);
}
```

main.cpp

```
#include "CPoint.h"
#include <cstdio>

int main() {
        CPoint p1;
        scanf("%ld %ld", &p1.x, &p1.y);
        double d = distOri(&p1);
        printf("Distance :%ld\n", d);
        return 0;
}
```

Un programme C « propre » :

Déclarations (.h)

Implantation (.c/.cpp)

Utilisation => voir les déclarations

Compilation : PréProcesseur

- Source divisés en :
  - Header (.h, .hh, .hpp) : déclarations de types, de variables et de fonctions
  - Source (.cpp, .cc) : corps des fonctions, définition des variables statiques
- On utilise #include pour inclure un source
  - Gestion faite par le préprocesseur cpp
  - Headers standards : <string>, <vector>, <iostream>
  - Headers personnels : « MyClass.h », « util/Utility.h»
- Attention aux double include

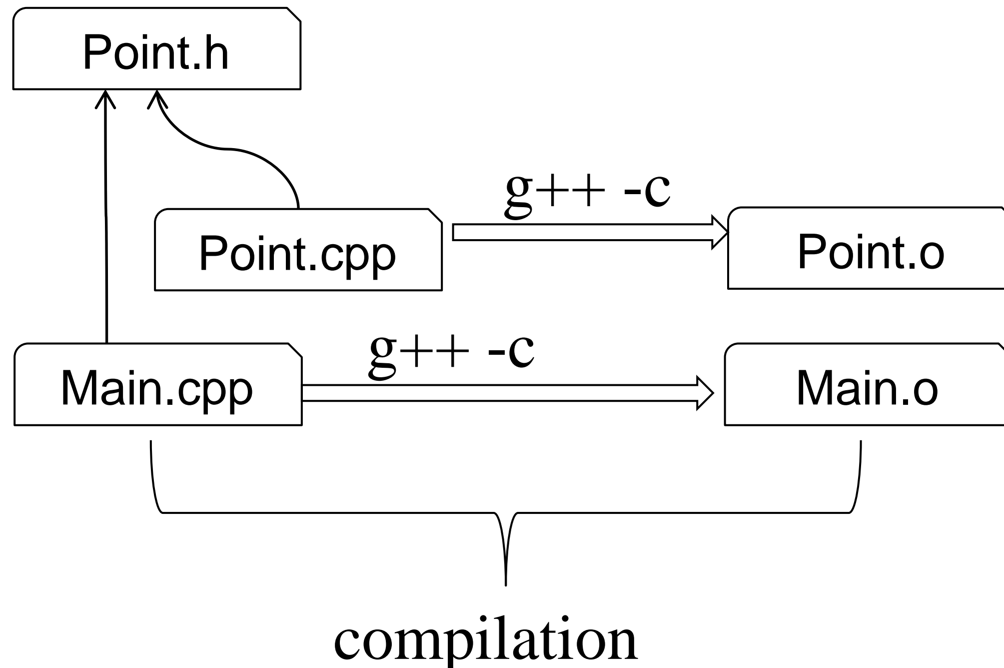
```
#ifndef MYCLASS_H
#define MYCLASS_H
// includes, déclarations
#endif
```

aussi : #pragma once
supporté par la plupart des compilos.

- Sources plateformes indépendantes ?
  - Lib standard OK, « stdunix.h » « windows.h »… NOK
- Le préprocesseur traite aussi les macros, dont on déconseille l'usage dans l'UE.

Compilation : unité de compilation .o

- Chaque fichier source est compilé séparément
  - Un fichier .cpp -> .o, fichier binaire plateforme dépendant
  - La compilation détecte un certain nombre de problèmes de syntaxe et de typage
  - La compilation cherche la déclaration adaptée, réalise les instanciations de paramètres et vérifie le typage des invocations



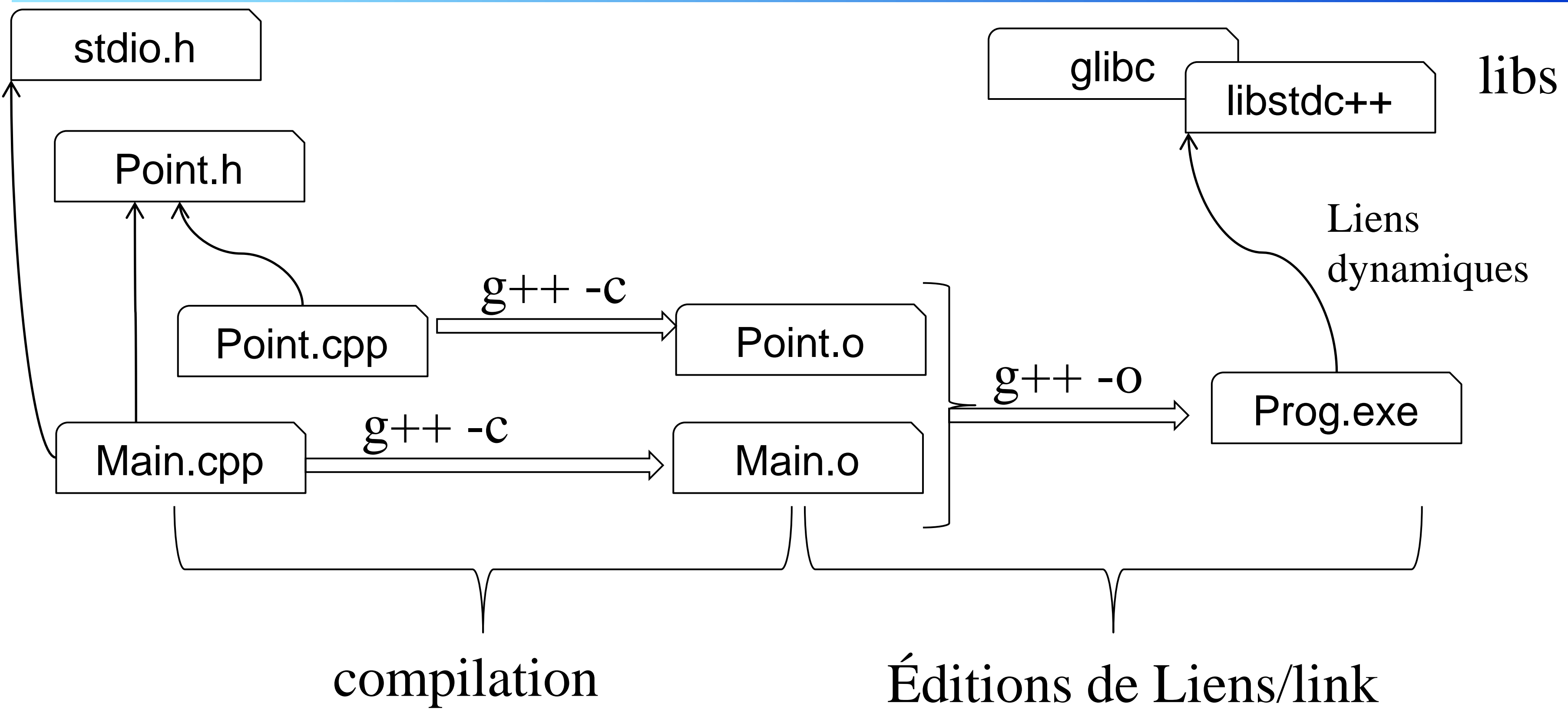
Compilation : unité de compilation .o

- Chaque fichier source est compilé séparément => .o
  - Contient : le code des fonctions du cpp, de l'espace pour les static déclarés et les littéraux (constantes, chaînes) du programme
  - Essentiellement : Une liste de fonctions en assembleur
  - Référence indirectement les divers .h dans lequel il a trouvé les références aux fonctions invoquées dans le code
- Concrètement on passe au compilateur
  - un MyClass.cpp source
  - -c pour arrêter la compilation avant le link
  - -Wall pour activer les warnings
  - -std=c++1y (selon compilo) pour le langage
  - -g pour activer les symboles de debug : le .o garde des liens vers les sources

g++ –Wall –std=c++1y –g –c Point.cpp

Produit : Point.o

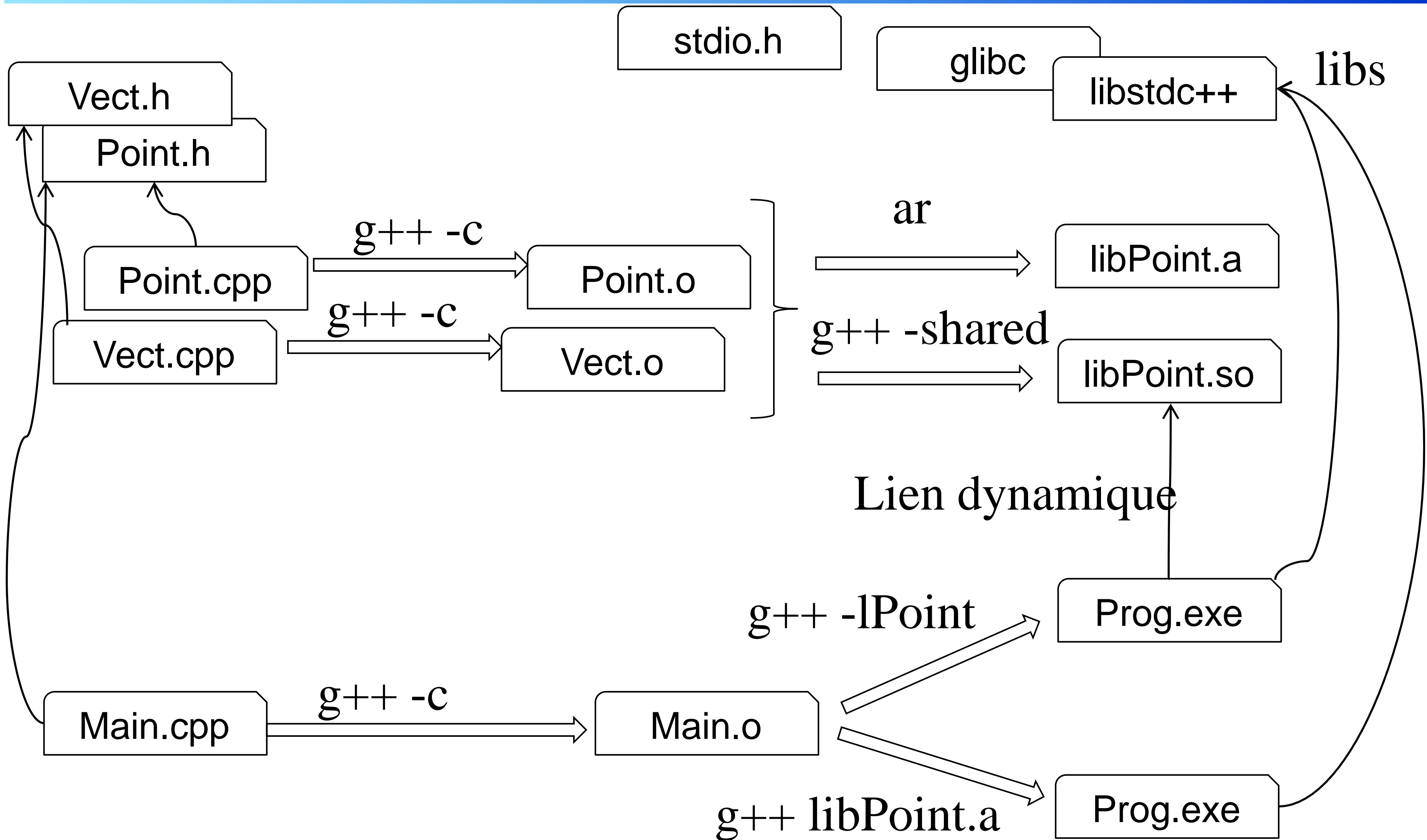
Compilation séparée : link



Compilation : link

- Edition de liens
  - Pour chaque fonction invoquée dans le .o => trouver son implantation
  - Construire un binaire complet : placer les invocations de fonctions
- Un exécutable est une application
  - Construit à partir d'un ensemble de .o et de librairies
  - Un seul des .o doit contenir une fonction **main**.
- Dépend de code dans des librairies (glibc, stdlibc++…)
  - Edition complète des liens : le binaire est auto-suffisant
  - Edition incomplète : dépendance du binaire à une lib dynamique

Compilation en lib



La commande « ldd » permet de savoir les déps dynamiques

Compilation : librairies

- Une librairie est un ensemble de .o agglomérés
  - Plateforme dépendant
  - Consistent (pas de double déclarations entre les .o)
  - On peut distribuer la librairie (pour une plateforme donnée, e.g. linux_x64) avec ses fichiers .h constituant son API
- Librairie statique ou dynamique
  - Change la façon dont l'exécutable se lie à la librairie
  - Lib dynamique : .so/.dylib/.dll, l'application invoque la lib (mise à jour possible de la lib sans recompiler l'application)
  - Lib statique : .a, .la l'application embarque le code utile de la lib
- Modifie le comportement du link
  - Embarque sélectivement le code des .o/.a fournis, lie sur les .so
  - Compilateurs modernes link optimisé : -fwhole-program

Sémantique d'un Executable

- Le « main » est le point d'entrée
  - On lance un processus (fork/exec) sur l'éxécutable
  - Instancie un processus : Espace d'adressage virtuel segmenté
    - Adresses de 0 à 2^64-1 sur x64
    - Segment de code : essentiellement le contenu des .o
    - Stack : pile pour les appels
    - Le reste de la mémoire n'est pas accessible (SEGMENTATION FAULT)
  - ✓ De base, on peut lire et écrire la mémoire dans cet espace
- Le processus interagit avec le système pour avoir des effets visibles
  - Allocation dynamique : obtenir une adresse vers une zone/libérer
  - Entrées/sorties : lire et écrire dans des descripteurs fichiers
    - Copie par le système depuis/vers l'espace d'adressage du processus
      - Fichiers, pipes, sockets, mémoire partagée…

Retour sur l'exemple

CPoint.h

```
#pragma once
struct Cpoint {
    double x;
    double y;
};

void init(CPoint* p, double x, double y);
double distOri(CPoint *p);
```

CPoint.cpp

```
#include "CPoint.h"
#include <cmath>

void init(CPoint *p, double x, double y)
{
        p->x = x;
        p->y = y;
}

double distOri(CPoint *p)
{
        return sqrt(p->x * p->x + p->y * p->y);
}
```

main.cpp

```
#include "CPoint.h"
#include <cstdio>

int main() {
        CPoint p1;
        scanf("%ld %ld", &p1.x, &p1.y);
        double d = distOri(&p1);
        printf("Distance :%ld\n", d);
        return 0;
}
```

Exécution :

- Stack, pointeurs
- Entrées sorties

/!\ Passage par valeur

CPoint.h

```
#pragma once
struct Cpoint {
    double x;
    double y;
};

void init(CPoint p, double x, double y);
double distOri(CPoint p);
```

CPoint.cpp

```
#include "CPoint.h"
#include <cmath>

void init(CPoint p, double x, double y)
{
        p.x = x;
        p.y = y;
}

double distOri(CPoint p)
{
        return sqrt(p.x * p.x + p.y * p.y);
}
```

main.cpp

```
#include "CPoint.h"
#include <cstdio>

int main() {
        CPoint p1;
        scanf("%ld %ld", &p1.x, &p1.y);
        init(p1,0.0,0.0);
        double d = distOri(p1);
        printf("Distance :%ld\n", d);
        return 0;
}
```

Copie des paramètres

- Lecture +/- ok (inefficace)
- Ecriture NOK

Passage au C++

Point.h

```
#pragma once
class Point
{
	double x;
	double y;
public :
	Point(double x, double y);
	double distOri();
};
```

Point.cpp

```
#include "Point.h"
#include <cmath>

Point::Point(double x, double y)
{
		this->x = x;
		this->y = y;
}
double Point::distOri()
{
return sqrt(this->x * this->x + this->y * this->y);
}
```

main.cpp

```
#include <iostream>
#include "Point.h"

int main() {
		double x, y;
		std::cin >> x >> y;
		Point p1 (x,y);
		double d = p1.distOri();
		std::cout << "Distance :" << d << std::endl;
		return 0;
}
```

- Class vs Struct
  - Ajouts de fonctions
- Fonction membre
  - Paramètre implicite this
- Constructeur
- Flux cin/cout

Les bases du C++

Hello world !

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
}
```

- Opérateur de résolution de namespace ::
- Headers standard, sans .h, entre <>
- Flux standard : cout, cerr, cin
- Opérateur << pour « pousser » dans le flux
- Fin de ligne + flush : endl

Entrée/sorties

- $<$ *iostream* $>$ offre une interface O.O. plus sécuritaire que $<$ *stdio.h* $>$.
- Les flots d'entrées/sorties prédéfinis sont :
  - *cout* associé à **la sortie standard** ($\Leftrightarrow$ *stdout* en C),
  - *cerr* associé à **la sortie erreur standard** ($\Leftrightarrow$ *stderr* en C),
  - *cin* associé à **l'entrée standard** ($\Leftrightarrow$ *stdin* en C).
- La fonction de sortie *printf*(...) est remplacé par l'opérateur d'insertion $<<$.
- La fonction de sortie *scanf*(...) est remplacé par l'opérateur d'extraction $>>$.

```
std::cout << "bonjour" << std::endl;
// printf("%s \n","bonjour");
int s; std::cin >> s ; // scanf("%d",s);
```

Variables, structures de contrôle

```
#include <iostream>
using namespace std;

int main ()
{
  int a=5;            // initial value: 5
  int b(3);           // initial value: 3
  int result;         //  no initial value

  a = a + b;
  result = a – 2 * b;
  cout << result;

  return 0;
}
```

- Clause « using namespace »
  - Les éléments de std deviennent visible dans ::
- Initialisation des variables
  - Syntaxe affectation = ou fonctionelle ()
  - Pas de valeur par défaut /!\
- Opérations arithmétique et priorités classiques (?)
- std::ostream polymorphique, accepte divers types
  - Cas particulier pour char * interprété comme une string du C

Types de base

Group	Type names*	Notes on size / precision
Character types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than `char`. At least 16 bits.
	char32_t	Not smaller than `char16_t`. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as `char`. At least 8 bits.
	signed **short** *int*	Not smaller than `char`. At least 16 bits.
	signed **int**	Not smaller than `short`. At least 16 bits.
	signed **long** *int*	Not smaller than `int`. At least 32 bits.
	signed **long long** *int*	Not smaller than `long`. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	**float**	
	double	Precision not less than `float`
	long double	Precision not less than `double`
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(*type*)	C-style type-casting	
4	Pointer-to-member	.* ->*	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or	|	bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction	||	logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -= >>= <<= &= ^= |=	assignment / compound assignment	Right-to-left
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

Opérateurs

- C++ possède un type bool
  - Constantes true et false
  - && and ; || or ; ! not
  - Toute expression != 0 est vraie par promotion
- Les opérateurs sont nombreux et leur règles de priorité complexe
  - Ne pas hésiter à sur-parenthèser un peu les expressions complexes
- Opérateurs new et delete ainsi que sizeof pour la gestion mémoire
- Opérateur de (cast) opère des conversions numériques
- Un sous ensemble de ces opérateurs peut être redéfini pour vos propres types (classes)

Overloadable operators													
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

std::string

La chaîne standard du c++, vient avec ses opérateurs.

```
#include <iostream>
#include <string>

int main() {
  std::string s1;
  std::cin >> s1;
  std::cout << "s1 = " <<  s1
            << std::endl;

  std::string s2 = "abcd";
  std::cout << "s2 = " << s2
            << std::endl;

  if (s1 == s2)
    std::cout << "s1 == s2";
  else if (s1 < s2)
    std::cout << "s1 < s2";
  else
    std::cout << "s1 > s2";
  std::cout << std::endl;
```

```
  for (int i = 0; i < s1.length(); ++i)
    std::cout << s1[i] << s1[i];
  std::cout << std::endl;

  for (int i = 0; i < s1.length(); ++i)
    s1[i] = 'a';
  std::cout << "s1 = " <<  s1
            << std::endl;

  s1 = s2;
  s1 = s1 + s2;
  std::cout << "s1 = " <<  s1
            << std::endl;

  char s[10];
  strcpy(s, s1.c_str());
}
```

Constantes

- Les constantes sont déclarées via le mot clé **`const`**.

Syntaxe

`const` *<id type>* *<id constante>* = *<expr. constante>*

Exemple

```
const float pi = 3.1416;
```

- Cela indique que l'identificateur doit garder une valeur constante.
  ```
  pi = 3.14; //erreur : modif. interdite
  ```
- L'initialisation est **obligatoire**.
  ```
  const float pi; //erreur : cste non initialisée
  ```
- C'est le compilateur qui réalise ces vérifications.
- L'avantage par rapport au macro est essentiellement le contrôle de type.

Constantes et Pointeurs

- Un pointeur vers une variable ne peut pointer vers une constante

```
const float pi = 3.1416;
float* p1 = &pi;          // Erreur : on peut modifier pi via p1
```

- Déclaration de pointeur vers une constante

```
const float* p2 = &pi;         // Initialisation optionnelle
float e = 2.7; p2 = &e;        // OK
*p2 = 2.72;                    // Erreur : p2 pointe sur une constante
```

- Déclaration de pointeur constant

```
float* const p3 = &e;          // Initialisation obligatoire
*p3 = 2.72;                    // OK
p3 = &pi;                      // Erreur : p3 est une constante
```

- Déclaration de pointeur constant vers une constante

```
const float* const p4 = &pi; // Initialisation obligatoire
*p4 = 3.14159;               // Erreur : p4 pointe sur une constante
p4 = &pi;                    // Erreur : p4 est une constante
```

Constantes

Paramètres constants

```
float puissance(const float, const float);
```

- Le compilateur vérifie que les paramètres ne sont pas modifiés dans le corps de la fonction.
- `const` est peu employé dans ce cas car la fonction n'a pas d'effet de bord (les paramètres sont passés par valeur). On évite juste certains bogues.

```
void afficher (const Personne*);
void afficher (const Personne&);
```

- Le compilateur vérifie que la personne pointée (référencée) n'est pas modifiée.
- On a la vitesse du passage par adresse (par référence) et la sécurité du passage par valeur. En prime, on évite certains bogues !

Références

- En C, lors d'un appel de fonction, les paramètres effectifs sont toujours passés **par valeur**.

```
void f(int i) {
  int a = 10;
  i = a;
}
void g(int* i) {
  int a = 20;
  *i = a;
}
```

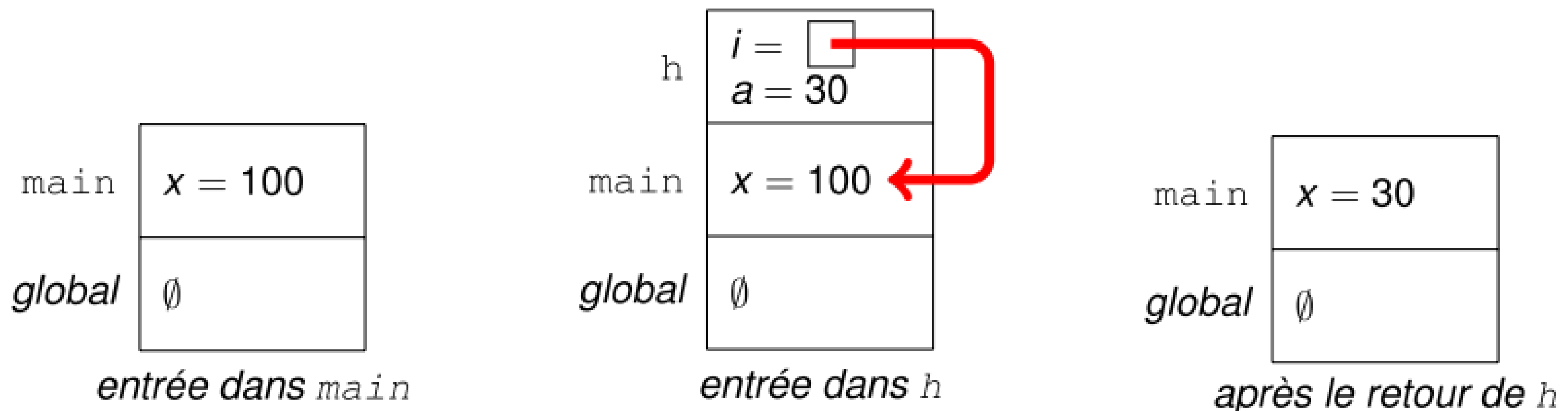
```
int main() {
  int x = 100;
  f(x);   // la valeur de x
  cout << x;
  g(&x); // la valeur de &x
  cout << x;
}
```

- Qu'affiche ce programme ?

- En C++, un nouveau type de passage de paramètres existe : le passage **par référence**.
- La modification de la valeur d'un paramètre passé par référence **est répercutée** au niveau de l'appelant.
- Exemple :

```
void h(int& i) {
  int a = 30;
  i = a;
}
```

```
int main() {
  int x = 100;
  h(x); // une référence sur x
  cout << x;
}
```



- Lors de l'appel `h(x)`, la variable locale `i` ***référence*** la variable `x`.
- Techniquement, c'est l'adresse de la variable fournie en paramètre qui est transmise à la fonction.

```
void swap(int& a, int& b) {
  int tmp = a;
  a = b; b = tmp;
}
```

- La fonction `swap` prend en paramètre deux références vers des entiers.
- Les paramètres effectifs d'un appel ne peuvent être que des variables.

```
int main() {
  int x = 10, y = 20;
  swap(x, y);          // x = 20 et y = 10
  swap(2, x+y);        // error
  // cannot convert param. 1 from 'const int' to 'int &'
  // cannot convert param. 2 from 'int' to 'int &'
}
```

```
struct BigStruct {
  int tab[10000];
  ...
};

void print(const BigStruct& b) {
  std::cout << b.tab[0] << ... << std::endl;
}
```

- Il n’y a aucune contrainte sur les paramètres effectifs.
- Le compilateur contrôle que le contenu de la variable n’est pas modifié.
- On cumule les avantages du passage de paramètre par adresse, la facilité d’écriture et les contrôles du compilateur.

Références

Utilisation remarquable – Référence retournée par une fonction

```
const int MAX = 10;
struct Personne {
  char nom[20];
  int age;
};

int& acces(Personne P[], const char nom[]) {
  for (int i = 0; i < MAX; ++i)
    if (strcmp(P[i].nom, nom) == 0)
      return P[i].age;
  // problème si le nom est introuvable
}
```

```
int main(void) {
  Personne Tab[MAX];
  ...
  acces(Tab, "toto") = 2;
}
```

- Une référence permet de créer un nom alternatif pour une variable.

```
int i = 10; int& ri = i; // i et ri désignent la même variable
int j = ri;              // j = 10
ri = 20;                 // i = 20
```

- Une référence doit être initialisée au moment de sa déclaration et référencera toujours la même variable.

```
int& ri;          // erreur, référence non initialisée
```

- Une référence peut être assimilée à un pointeur constant.

```
ri++;     // ri est inchangée, i est incrémentée
```

- Une référence peut désigner une constante uniquement si elle est déclarée comme étant une référence vers une constante.

```
char& r = 'a';              // illégal
const char& r = 'a';        // légal
char c;
const char& rc = c;         // légal
rc = 'a';                   // illégal
```

- Ne pas confondre l'opérateur & permettant d'obtenir l'adresse d'une variable avec le signe & employé pour déclarer des références.
- De manière générale, à partir de n'importe quel type `T`, on peut construire les nouveaux types suivants.
  - `T*` : pointeur vers une donnée de type `T` (`int *pi`).
  - `T[]` : tableau de données de type `T` (`char s[256]`).
  - `T&` : référence vers une donnée de type `T` (`int &ri`).

C++ :classe, instance, allocation

Une classe : déclaration

```
class <nom de la classe> {
public:
   <interface> => mode d'emploi de la classe
private:
   <partie déclarative de l'implémentation>
};
```

- L'interface et la partie déclarative de la classe comprennent :
  - ✓ des *prototypes de méthodes* (appelées aussi fonctions membres)
  - ✓ des *déclarations de champs* (appelées aussi données membres)

- Habituellement, l'interface **ne contient que** des prototypes de méthodes

Exemple : Pile en C++, utilisation de Classe

Données privées

Interface publique

```
class Pile {
private:
    static const int TAILLE = 10;
    double tab[TAILLE];
    unsigned int cpt;                    // nombre d'éléments empilés

public:
    Pile ();                             // constructeur vide
    Pile (double);                       // construit avec 1 élément
    bool   estPleine () const;           //const => lecture seule
    bool   estVide () const;             //const => lecture seule
    void   empiler  (double);            /// @pre !estPleine()
    double      depiler ();              /// @pre !estVide()
    double      getSommet () const;      /// @pre !estVide()
};
```

Une classe : objet et fonctions membres

- Dès qu'une classe est déclarée, on peut ***instancier des objets*** (i.e. déclarer des variables):

```
Pile p;
```

- Dès qu'un objet est instancié, on peut invoquer ces fonctions membres publiques :

```
p. empiler (1.);
Pile * pp = &p;
pp->getSommet();  // via un pointeur
```

- **Attention : l'appel d'une fonction membre est toujours associé à un objet de la classe.**

```
p. empiler (1.); // OK, p est l'objet invoqué
getSommet();  // erreur, pas d'objet destinataire
              // Sauf si instance courante (this)
```

Une classe : implémentation Pile.cpp

- Pour qu'une classe soit complètement définie, il faut préciser le code de toutes les méthodes apparaissant dans sa déclaration

```cpp
#include "Pile.h"

Pile::Pile() {
        // constructeur vide
        cpt = 0;
        // this->cpt = 0;
}
Pile::Pile(double x) {
        // autre constructeur
        empiler(x);
        // this->empiler(x);
}
void Pile::empiler(double x) {
        tab[cpt++] = x;
}
```

```cpp
double Pile::depiler() {
        return tab[--cpt];
}
bool Pile::estPleine() const {
        return (cpt == TAILLE);
}
bool Pile::estVide() const {
        return (cpt == 0);
}
double Pile::getSommet() const {
        return tab[cpt - 1];
}
```

Une classe : programme utilisateur

```
#include "Pile.h"
#include <iostream>

int main() {
        Pile p;
        p.empiler(4.5);
        std::cout << p.getSommet();
}
```