INF3105 – Introduction au langage C++

Éric Beaudry

Université du Québec à Montréal (UQAM)

2017A

Sommaire

C++ dans INF3105

- L'objectif principal d'INF3105 $\neq$ apprendre le langage C++.
- C++ est plutôt le langage que nous allons utiliser pour mettre en pratique les concepts fondamentaux de structures de données.
- Les séances en classe ne font pas un tour complet de C++.
- Nous nous limiterons aux bases du C++.
- Il faut compléter l'apprentissage de C++ dans les labs et dans ses heures de travail personnel.
- Conseil : prenez une journée complète durant un week-end pour faire un tutoriel en ligne sur C++.

Historique

Origine du C++

- « ++ » dans « C++ » signifie un incrément au langage C.
- C++ $\approx$ C + Extension de **programmation orientée objet**.
- Développé par Bjarne Stroustrup au Bell labs d'AT&T dans les années 1980.

Standardisation / Normalisation

- Normalisé par ISO (Organisation mondiale de normalisation) depuis 1998 (C++98). C++03, C++11, C++14, C++17.

Influence

- Le C++ est très utilisé en industrie et en recherche (efficacité).
- Le C++ a influencé d'autres langages comme Java et C#.

Caractéristiques et paradigmes

- Impératif.
- Procédural.
- Fortement typé.
- Orienté objet.
- Générique.
- Langage de haut niveau (mais plus bas que Java).
- Compilé pour une machine cible (jeu d'instructions d'un type de CPU).
- Multiplateforme.
- ...

Exemple de fichier source C++

bienvenue.cpp

```
#include <iostream>

// La fonction main est le point d'entree a l'execution
int main(){
    std::cout << "Bienvenue au cours INF3105 en C++ !" << std::endl;
    return 0;
}
```

Fichiers sources

Fichiers d'entête (.h, .hpp)

Les fichiers d'**entête** (*header*), ayant pour extension `.h` ou `.hpp`, contiennent généralement des **déclarations**.

Fichiers sources (.cc, .cpp, .c++)

Les fichiers **sources** ayant pour extension `.cc`, `.cpp` ou `.c++`, contiennent généralement les **définitions** (l'implémentation). Ces fichiers peuvent aussi contenir des déclarations.

Déclaration vs Définition

Déclaration

- La compilation se fait en une seule passe (excluant l'édition des liens).
- Tout doit être déclaré avant d'être utilisé.
- Une déclaration ne fait que déclarer l'existence de quelque chose lié à un identificateur (symbole). Exemples : variables, fonctions, classes, etc.

Définition

- La définition est le code des functions, constructeurs, etc.
- Après la compilation, il y a une passe d'édition des liens (*linker*).
- Tout symbole utilisé doit être défini à l'édition des liens.

Déclaration vs Définition : Exemple 1

helloworld.cpp

```
#include <iostream>
int main(int argc, char** argv)
{
    allo(); // Error: symbol allo undefined!
    return 0;
}
// Declaration et definition d'une fonction allo()
void allo(){
    std::cout << "Hello World!" << std::endl;
}
```

Déclaration vs Définition : Exemple 2

helloworld.cpp

```
#include <iostream>
// Declaration et definition d'une fonction allo()
void allo(){
    std::cout << "Hello World!" << std::endl;
}
int main(int argc, char** argv)
{
    allo();
    return 0;
}
```

Déclaration vs Définition : Exemple 3

helloworld.cpp

```
#include <iostream>
// Declaration de la fonction allo(). Aussi appelle prototype de fonction.
void allo();
int main(int argc, char** argv)
{
    allo();
    return 0;
}
// Definition de la fonction allo()
void allo(){
    std::cout << "Hello World!" << std::endl;
}
```

Exemple de fichiers sources

allo.h

```
void allo(); // Declaration
```

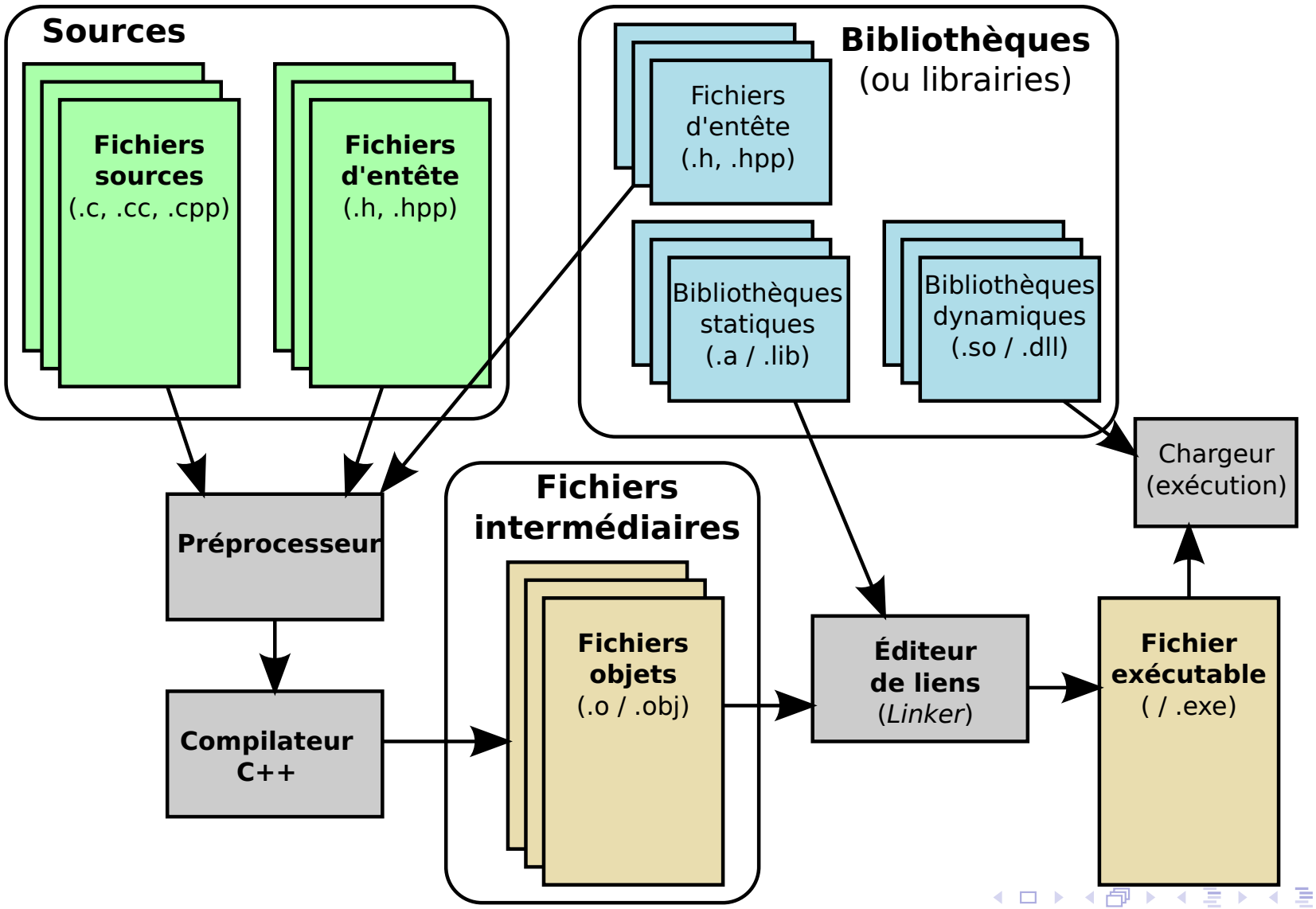
allo.cpp

```
#include <iostream>
#include "allo.h" // optionnelle
void allo(){ // Definition
    std::cout << "Hello World!" << std::endl;
}
```

helloworld.cpp

```
#include "allo.h" // inclure allo.cpp pourrait fonctionner, mais fortement deconseille
int main(int argc, char** argv) {
    allo();
    return 0;
}
```

Organisation et compilation



Directive `#include`

```
#include <iostream> // < > demande de charger depuis la lib standard
                    // ou dans les chemins specifies par une option -I
                    // g++ -I ../mabibliotheque/include/

#include "fichier.h" // " " demande de charger depuis le repertoire
                     // contenant le fichier courant
```

Quelques mots réservés

Types

void, bool, char, short int, int, long, float, double
unsigned

Boucles et instructions de contrôles

if, else, while, do, for, switch ... case

Structures

class, struct, union

Types

Type (mot clé)	Description	Taille (octets)	Capacité
`bool`	Booléen	1	$\{false, true\}$
`char`	Entier / caractère ASCII	1	$\{-128, \ldots, 127\}$
`unsigned char`	Entier / caractère ASCII	1	$\{0, \ldots, 255\}$
`unsigned short` `unsigned short int`	Entier naturel	2	$\{0, \ldots, 2^16 - 1\}$
`short` `short int`	Entier	2	$\{-2^{15}, \ldots, 2^{15} - 1\}$
`unsigned int`	Entier naturel	4	$\{0, \ldots, 2^{32} - 1\}$
`int`	Entier	4	$\{-2^{31}, \ldots, 2^{31} - 1\}$
`unsigned long`	Entier naturel	8	$\{0, \ldots, 2^{64} - 1\}$
`long`	Entier	8	$\{-2^{63}, \ldots, 2^{63} - 1\}$
`float`	Nombre réel	4	$\pm 3.4 \times 10^{\pm 38}$ (7 chiffres)
`double`	Nombre réel	8	$\pm 1.7 \times 10^{\pm 308}$ (15 chiffres)
`long double`	Nombre réel	8	$\pm 1.7 \times 10^{\pm 308}$ (15 chiffres)

Déclaration variables

Une variable est une **instance** d'un type de données. En C++, les variables sont considérées comme des **objets**. Chaque variable est nommée à l'aide d'un identificateur. L'identificateur doit être unique dans sa portée.

Exemple

```
// Declaration d'un entier (sans initialisation)
int a;
/* Declaration de nombres reels (sans initialisation) */
float f1, f2, f3;
/* Declaration de nombres avec initialisation explicite*/
int n1(12), n2=20;
```

Initialisation des variables

- Constructeur : lors d'une initialisation explicite.
- Constructeur sans argument : si aucune initialisation n'est explicitée.
- Par défaut, les types de base ne sont pas initialisés.
  - Avantage : Efficacité.
  - Inconvénient : L'exécution peut dépendre du contenu précédent en mémoire.
  - Inconvénient : L'exécution peut être pseudo non déterministe (comportement « aléatoire »).
  - Problème : Source potentielle de bugs.

Énoncés et expressions

Comme dans la plupart des langages de programmation, le corps d'une fonction en C++ est constitué d'énoncés (*statements*). Sommairement, un énoncé peut être :

- une déclaration de variable(s) ;
- une expression d'affectation ;
- une expression ;
- une instruction de contrôle ;
- un bloc d'énoncés entre accolades `{ }`.

À l'exception d'un bloc `{ }`, un énoncé se termine toujours par un point-virgule (`;`).

Énoncés / Affectation

Affectation

```
// Declaration
int a;
// Affectation
a = 2 + 10;
```

Expressions

En C++, une expression peut être :

- un identificateur (variable) ou un nombre ;
- une expression arithmétique ou logique ;
- un appel de fonction ;
- une autre expression entre parenthèses ( ) ;
- un opérateur d'affectation (=, +=, etc.) ;
- etc.

Exemples d'expressions

```
4+5*6-8;
(4+5)*(6-8);
a * 2 + 10;
```

```
a = b = c = d;
// est l'equivalent de :
c = d; b = c; a = b;
```

Exemples d'expressions

```
a++; // a = a + 1;
a+=10; // a = a + 10;
a*=2; // a = a * 2;
a/=2; // a = a / 2;
```

```
b = a++; // b=a; a=a+1;  // post-increment
b = ++a; // a=a+1; b=a;  // pre-increment
b = a--; // b=a; a=a-1;  // post-decrement
b = --a; // a=a-1; b=a;  // pre-decrement
```

Instructions de contrôle

if, while, for, do...while, switch ... case, break, ...

Tableaux

```
int tableau1[5] = {0, 5, 10, 15, 20};
int tableau2[10] = {0, 5, 10, 15, 20};
int tableau3[] = {0, 5, 10, 15, 20};
```

Non-vérification des indices des tableaux

```
#include <iostream>
using namespace std;
int main() {
    int tab1[5], tab2[5];
    for(int i=0;i<5;i++){
      tab1[i] = i;      tab2[i] = i + 10;    }
    for(int i=0;i<16;i++)  cout << " " << tab1[i];
    cout << endl;
    for(int i=0;i<15;i++)  tab1[i] = 99 - i;
    for(int i=0;i<5;i++)   cout << " " << tab1[i];
    cout << endl;
    for(int i=0;i<5;i++) cout << " " << tab2[i];
    cout << endl;
    return 0;
}
```

Résultat sur Ubuntu 16.04 / g++ 5.4)

```
0 1 2 3 4 0 4197261 0 10 11 12 13 14 0 4196528 0
99 98 97 96 95
91 90 89 88 87
```

```
#include <iostream>
using namespace std;
int main() {
    int tab1[5], tab2[5];
    for(int i=0;i<5;i++){
      tab1[i] = i;      tab2[i] = i + 10;    }
    for(int i=0;i<16;i++)  cout << " " << tab2[i];
    cout << endl;
    for(int i=0;i<15;i++)  tab2[i] = 99 - i;
    for(int i=0;i<5;i++)   cout << " " << tab1[i];
    cout << endl;
    for(int i=0;i<5;i++) cout << " " << tab2[i];
    cout << endl;
    return 0;
}
```

Résultat sur Malt (CentOS 6.8 / g++ 4.9.0)

```
10 11 12 13 14 0 4196043 0 0 1 2 3 4 52 4196960 0
91 90 89 88 87
99 98 97 96 95
```

Non-vérification des indices des tableaux

- Aller chercher un indice dans tableau se fait par une arithmétique de pointeurs.
- Exemple : `tab2[10]` est équivalent à `*(tab2 + 10)`.
- Note : le `+10` est implicitement multiplié par `sizeof(int)` à la compilation.
- La non-vérification des indices = Source potentielle de bugs.

Pointeurs et références

- Pointeur = adresse mémoire.
- Pointeurs différents en Java.
- Référence : a pour objectif d'être manipulable comme un objet (même syntaxe).
- Référence : officiellement un «alias» pour une autre variable ou objet.
- Référence : généralement implémentée par une adresse mémoire.
- Passage de paramètres par valeur ou par référence.
- Le passage par pointeur est un passage par valeur d'une adresse pointant vers un objet donné.

Pointeurs

Dans la déclaration de variable, la porté d'un symbole étoile * se limite à une variable.

```
int n = 3;
int* ptr_n = &n;
int* tableau = new int[100];
```

```
//Declare le pointeur p1 et l'objet o1
int* p1, o1;
//Declare les pointeurs p2, p3, p4 et l'objet o2
int *p2, *p3, o2, *p4;
```

Déférencement de pointeurs

Déférencer = aller chercher (le contenu de) la case mémoire.

```
int n=0;
int *pointeur = &n;
*pointeur = 5; // effet : n=5
std::cout << "n=" << *pointeur << std::endl;
```

Arithmétique des pointeurs

Code 1 (meilleure lisibilité)

```
int tableau[1000];
int somme = 0;
for(int i=0;i<1000;i++)
    somme += tableau[i];
```

Code 2 (potentiellement plus efficace)

```
int tableau[1000];
int somme = 0;
int* fin = tableau+1000; // pointe sur l'element suivant le dernier element
for(int* i=tableau;i<fin;i++)
    somme += *i; // une opération de moins : tableau + 1
```

Références

```
int n = 2;
int& ref_n = n;
n = 3;
std::cout << "ref_n=" << ref_n << std::endl;
```

Fonctions

- Permet de faire de la programmation procédurale.
- Similaire à Java.
- Les paramètres sont passés sur la pile d'exécution.
- Fonctions globales et membres d'une classe.

```
int somme(int a, int b){
    return a+b;
}
class A{
    void b(){std::cout << "Appel de b()!" << std::endl; }
};
A a;
a.b();
int x=2;
int z = somme(x, 5);
```

Signature de fonction (1)

- Chaque fonction a une **signature unique**.
- Une **signature de fonction est définie** par :
  - le nom de la fonction ;
  - le nombre de paramètres ;
  - le type de chacun des paramètres (incluant le paramètre implicite) ;
  - le modificateur `const` (ou son absence) pour chaque paramètre.
- Le type de retour d'une fonction ne fait par partie de sa signature.
- Une fonction peut être **déclarée** plusieurs fois.
- Une fonction doit être **définie** au plus une fois. Exactement une fois si elle est utilisée.
- Le type de retour d'une fonction déjà déclarée ne peut pas être modifiée.

Signature de fonction (2)

```
void f1(int a){ cout << "A\n"; }
void f1(int x, int y){ cout << "B\n"; }
void f1(const string& s){ cout << "C\n"; }
void f1(string& s){ cout << "D\n"; }
void f1(int k){ cout << "E\n"; } // ERREUR: redéfinition
void f1(int a); // OK, seulement redéclaration
int  f1(int x, int y); // ERREUR, changement de type de retour
double f2(); // OK, seulement déclarée, puisque jamais utilisée, définition non requise
void f3(int& k) { k++; }

int main(){
   f1(0); // ==> A
   f1(0, 2); // ==> B
   string x = "aa";
   f1(x); // ==> D (la version la moins contraignante : non const)
   const string& y = x;
   f1(y); // ==> C
   const int z = 8;
   f3(z); // ERREUR: incomptabilité de type (attendu non const)
}
```

Signature de fonction (3)

```
class A{
  public:
    void f() const;  //A
    void f();  //B
    void f(int x) const; //C
    void f(int x); //D
    void g(int& x); // E
    void g(const int& x); //F
};
int main(){
    A a1;
    const A* a2 = &a1;
    a1.f(); //==> B
    a2->f(); // ==> A
    a1.f(2); //==> D
    a2->f(2); // ==> C
    int x;
    const int& y = x;
    a1.g(x); //==> E
    a1.g(y); // ==> F
}
```

Valeurs par défaut (omission derniers paramètres)

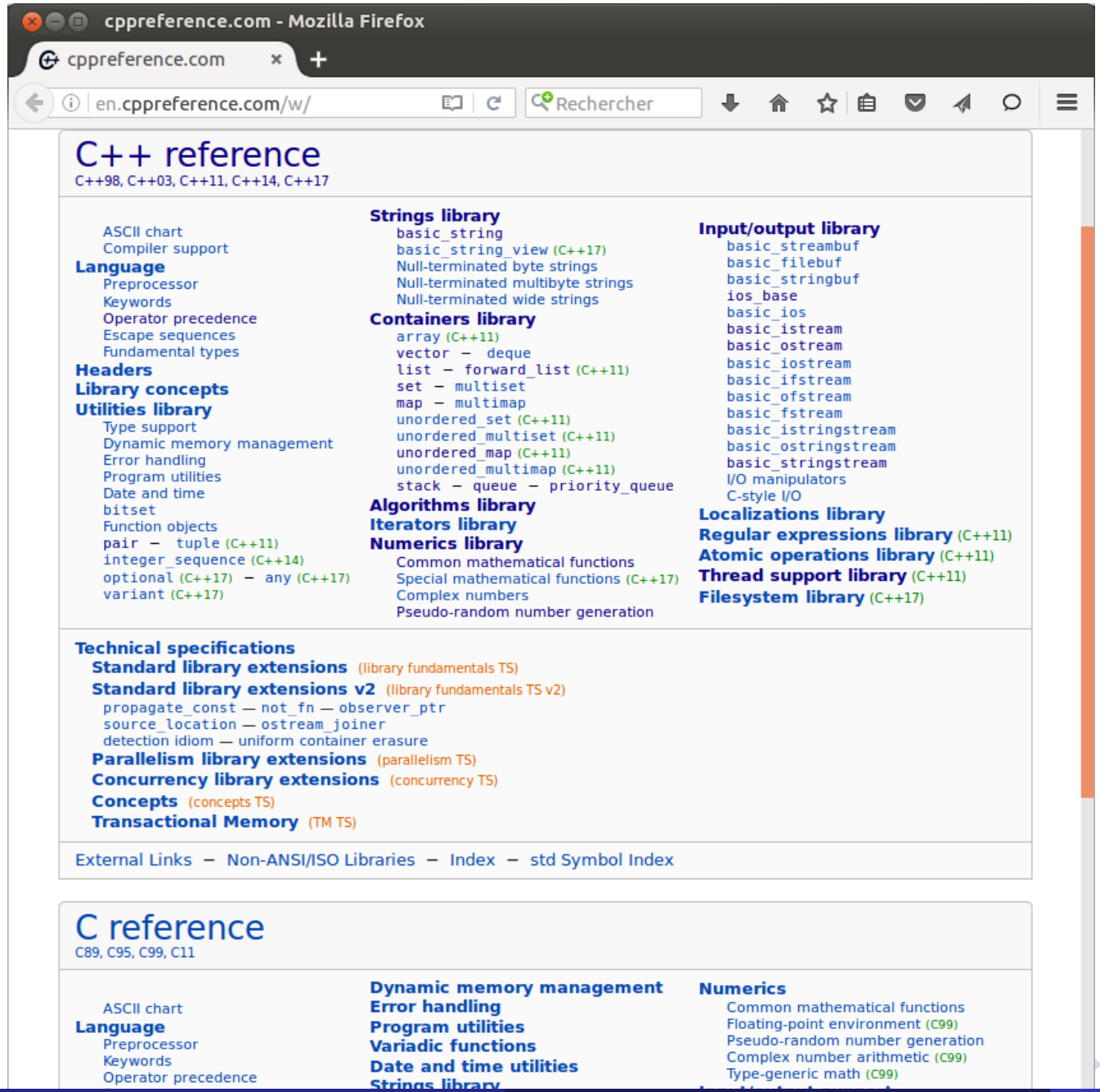
```
void test1(int a=5, float b=0.8, char c='k'){
    cout << "a=" << a << " b=" << b << " c='" << c << "'" << end;
}
void test2(int a, float b, char c='k'){
    cout << "a=" << a << " b=" << b << " c='" << c << "'" << end;
}
int main(){
    test1(); //idem test1(5, 0.8, 'k');
    test1(0); //idem test1(0, 0.8, 'k');
    test1(0, 0.0); //idem test1(0, 0.0, 'k');
    test1(0, 0.0, '0'); //idem test1(0, 0.0, '0');
    test2(); // erreur compilation
    test2(0); // erreur compilation
    test2(0, 0); // OK
    test2(0, 0, '0'); // OK
}
```

Passage de paramètres par valeur et référence

```
void test(int a, int* b, int* c, int& d, int*& e){
    a=11; // effet local
    b++; // change l'adresse locale de b
    *c=13; // change la valeur pointee par c
    d=14; //  change la valeur referee par d
    e=c; // change la valeur du pointeur (adresse) pour celle de c.
}
int main(){
    int v1=1, v2=2, v3=3, v4=4, *p5=&v1;
    test(v1, &v2, &v3, v4, p5);
    cout<<v1<<'\t'<<v2<<'\t'<<v3<<'\t'<<v4<<'\t'<<*p5<<'\t'<<endl;
    // affiche : 1  2  13  14 13
    return 0;
}
```

Bibliothèque standard de C++

- Un compilateur C++ vient généralement avec la bibliothèque standard de C++ (*C++ Standard Library*) contenant les fonctions, constantes, objets, classes, etc. de base.
- Fonctionnalités principales en C :
  - Entrées/Sorties : `printf`, `scanf`, `fopen`, `fread`, `fwrite`, ...
  - Fonctions mathématiques : `fabs`, `floor`, `sin`, `cos`, `sqrt`, ...
  - Manipulation de chaines de caractères C (`char*`) : `strcpy`, `strcmp`, ...
- Fonctionnalités principales en C++ :
  - Classe `std::string` (qui encapsule des chaines `char*`)
  - Entrées/Sorties : flux C++ (`iostream`, `ofstream`, `ifstream`, ...)
  - Conteneurs : `vector`, `list`, `set`, `map`, ...
  - ...


cppreference.com - Mozilla Firefox
cppreference.com
en.cppreference.com/w/
Rechercher
C++ reference
C++98, C++03, C++11, C++14, C++17
ASCII chart
Compiler support
Language
Preprocessor
Keywords
Operator precedence
Escape sequences
Fundamental types
Headers
Library concepts
Utilities library
Type support
Dynamic memory management
Error handling
Program utilities
Date and time
bitset
Function objects
pair — tuple (C++11)
integer_sequence (C++14)
optional (C++17) — any (C++17)
variant (C++17)
Strings library
basic_string
basic_string_view (C++17)
Null-terminated byte strings
Null-terminated multibyte strings
Null-terminated wide strings
Containers library
array (C++11)
vector — deque
list — forward_list (C++11)
set — multiset
map — multimap
unordered_set (C++11)
unordered_multiset (C++11)
unordered_map (C++11)
unordered_multimap (C++11)
stack — queue — priority_queue
Algorithms library
Iterators library
Numerics library
Common mathematical functions
Special mathematical functions (C++17)
Complex numbers
Pseudo-random number generation
Input/output library
basic_streambuf
basic_filebuf
basic_stringbuf
ios_base
basic_ios
basic_istream
basic_ostream
basic_iostream
basic_ifstream
basic_ofstream
basic_fstream
basic_istringstream
basic_ostringstream
basic_stringstream
I/O manipulators
C-style I/O
Localizations library
Regular expressions library (C++11)
Atomic operations library (C++11)
Thread support library (C++11)
Filesystem library (C++17)
Technical specifications
Standard library extensions (library fundamentals TS)
Standard library extensions v2 (library fundamentals TS v2)
propagate_const — not_fn — observer_ptr
source_location — ostream_joiner
detection idiom — uniform container erasure
Parallelism library extensions (parallelism TS)
Concurrency library extensions (concurrency TS)
Concepts (concepts TS)
Transactional Memory (TM TS)
External Links — Non-ANSI/ISO Libraries — Index — std Symbol Index
C reference
C89, C95, C99, C11
ASCII chart
Language
Preprocessor
Keywords
Operator precedence
Dynamic memory management
Error handling
Program utilities
Variadic functions
Date and time utilities
Numerics
Common mathematical functions
Floating-point environment (C99)
Pseudo-random number generation
Complex number arithmetic (C99)
Type-generic math (C99)

Entrée standard et sortie standard

Pour avoir accès aux flux standards de la bibliothèque standard de C++, il faut ajouter l'entête `iostream` à l'aide de la directive `#include <iostream>`. Ce fichier d'entête définit entre autres les trois flux suivants :

- `std::cin` : flux d'entrée depuis l'entrée standard (*stdin*) ;
- `std::cout` : flux de sortie vers la sortie standard (*stdout*) ;
- `std::cerr` : flux de sortie vers la sortie d'erreurs (*stderr*).

Exemple

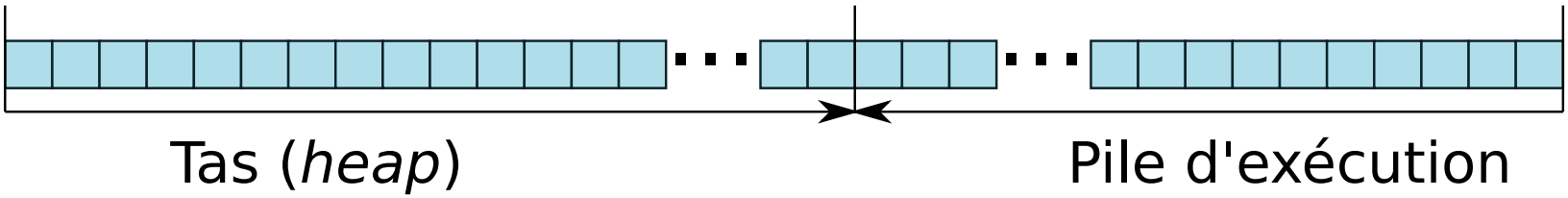
demo01.cpp

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

int main(int argc, char** argv){
   int a, b;
   cout << "Entrez deux nombres:" << endl;
   cin >> a >> b;
   int somme = a + b;
   cout << "La somme est " << somme << endl;
   return 0;
}
```

demo02.cpp

```
#include <fstream>
int main(int argc, char** argv){
   int a, b;
   std::ifstream in("nombres.txt");
   // Lire deux nombres depuis le fichier d'entrée nombres.txt
   in >> a >> b;
   if(in.fail()) {
       std::cerr << "Il y a eu un problème lors de la lecture!" << std::endl;
   }
   int somme = a + b;
   std::ofstream out("somme.txt");
   out << somme << endl;
   return 0;
}
```



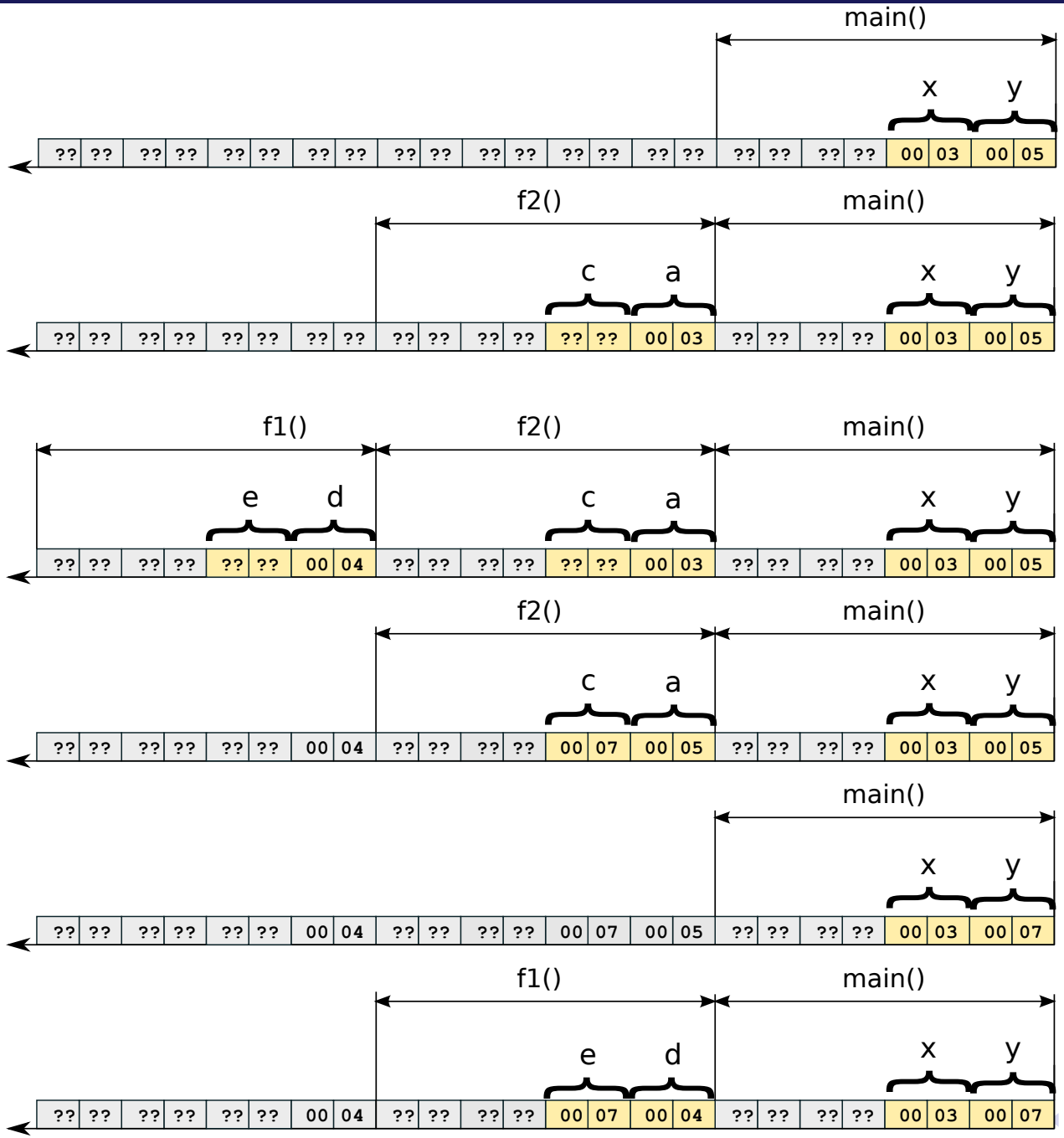
- La mémoire doit être vue comme une grand bloc contiguë (vecteur linéaire) de mémoire.
- La pile d'exécution a souvent une taille fixe.
- Le tas (*heap*) a souvent une taille variable (allouable par le système d'exploitation).

Allocation mémoire

- Deux types d'allocation :
  - Automatique : fait automatiquement par le compilateur.
  - Dynamique : allouée explicitement sur le tas (*heap*).
- Toute mémoire allouée dynamiquement doit être libérée dynamiquement.
- Algorithmes d'allocation/libération (dépendant du compilateur/système d'exploitation).

Allocation sur la pile

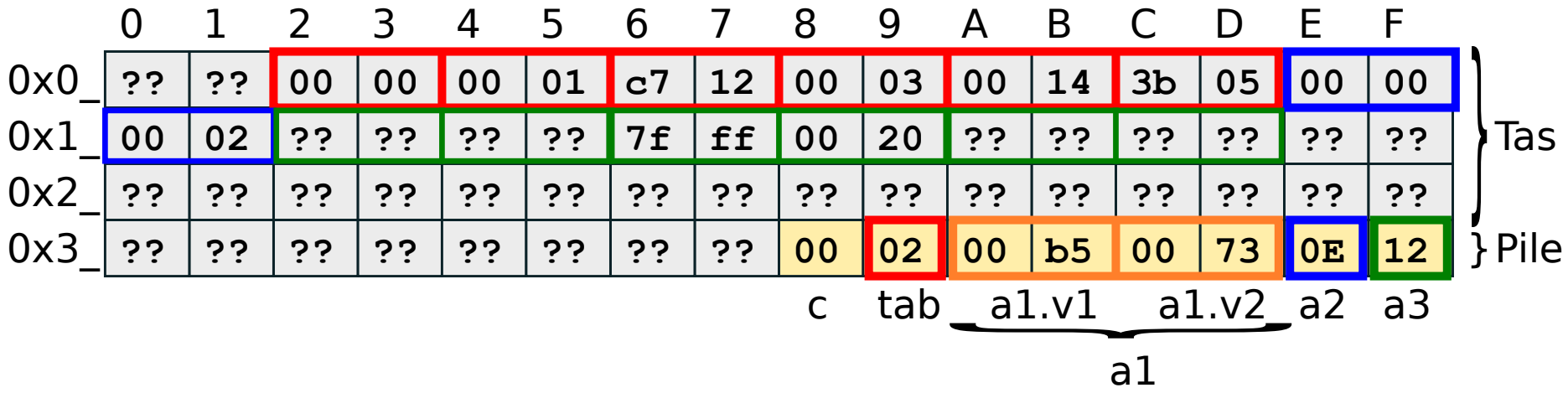
```
short int f1(){
    short int d = 4;   short int e;
    return d;
}
short int f2(short int a){
    short int c = a + f1();
    a += 2;
    return c;
}
int main(){
    short int x=3;   short int y=5;
    y = f2(x);
    f1();
    return 0;
}
```


main()
x y
?? 00 03 00 05
f2()
main()
c a
x y
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 00 03 ?? ?? ?? ?? 00 03 00 05
f1()
f2()
main()
e d
c a
x y
?? ?? ?? ?? ?? ?? 00 04 ?? ?? ?? ?? ?? ?? 00 03 ?? ?? ?? ?? 00 03 00 05
f2()
main()
c a
x y
?? ?? ?? ?? ?? ?? 00 04 ?? ?? ?? ?? 00 07 00 05 ?? ?? ?? ?? 00 03 00 05
main()
x y
?? ?? ?? ?? ?? ?? 00 04 ?? ?? ?? ?? 00 07 00 05 ?? ?? ?? ?? 00 03 00 07
f1()
main()
e d
x y
?? ?? ?? ?? ?? ?? 00 04 ?? ?? ?? ?? 00 07 00 04 ?? ?? ?? ?? 00 03 00 07

Allocation sur le tas (*heap*)

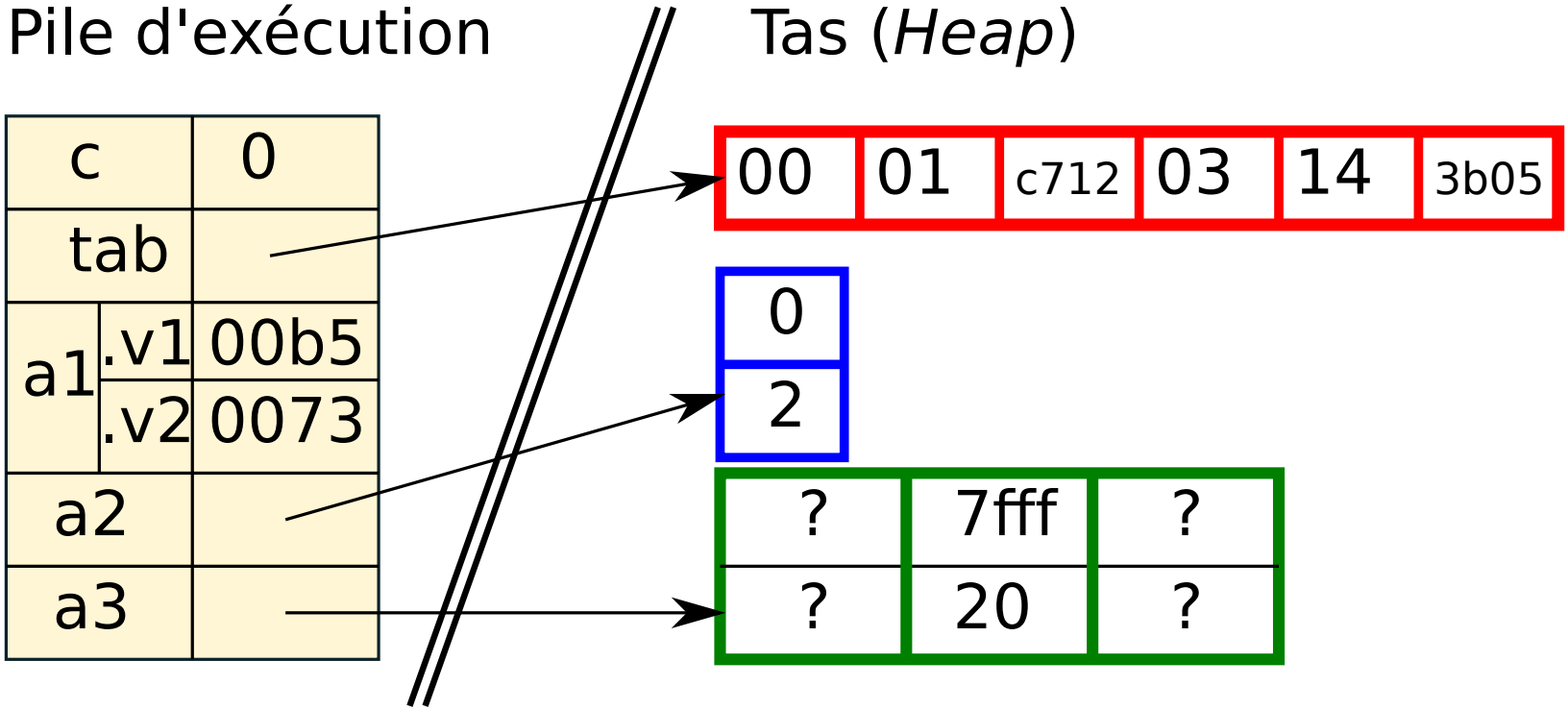
```
struct A{
    short int v1, v2;
};
int main(){
    char c = 0;
    short int *tab = new short int[6] {0x00, 0x01, 0xc712, 0x03, 0x14, 0x3b05};
    A a1; a1.v1=0x00b5; a1.v2=0x0073;
    A* a2 = new A();
    a2->v1=0; a2->v2=2;
    A* a3 = new A[3];
    a3[1].v1=0x7fff; a3[1].v2=0x0020;
    // Sur la diapo suivante : etat de la memoire jusqu'ici
    delete[] tab;    delete a2;    delete[] a3;
    return 0;
}
```

Allocation mémoire




```
struct A{
    short int v1, v2;
};
int main(){
    char c = 0;
    short int *tab = new short int[6] {0x00, 0x01, 0xc712, 0x03, 0x14, 0x3b05}; // supposons une machine 8 bits (taille des pointeurs = 8 bits)
    A a1; a1.v1=0x00b5; a1.v2=0x0073;
    A* a2 = new A();
    a2->v1=0; a2->v2=2;
    A* a3 = new A[3];
    a3[1].v1=0x7fff; a3[1].v2=0x0020;
    // Le schema ci-haut montre l'etat de la memoire jusqu'ici.
    delete[] tab;    delete a2;    delete[] a3;
    return 0;
}
```

Représentation abstraite de la mémoire



Ici, on fait abstraction des adresses mémoire des blocs alloués sur le tas (*heap*).

Classes en C++

- Similaire à Java.
- En fait, c'est Java qui est inspiré (et simplifié !) de C++.
- Constructeurs.
- Constructeur par défaut.
- Constructeur par copie.
- Destructeurs.
- Surcharge d'opérateurs (+, -, +=, -=, =, ==, <, (), etc.).
- Héritage simple et multiple (nous n'en aurons pas besoin en INF3105).

Classe Point

```
class Point {
  public:
    double distance(const Point& p) const;
  private:
    double x, y;
};
```

Constructeur

Un constructeur porte le nom de la classe et peut avoir zéro, un ou plusieurs arguments. Comme son nom l'indique, le rôle d'un constructeur est de construire (instancier) un objet. Un constructeur effectue dans l'ordre :

1. appelle le constructeur de la ou des classes héritées ;
2. appelle le constructeur de chaque variable d'instance ;
3. exécute le code dans le corps du constructeur.

Destructeur

Un destructeur porte le nom de la classe et a aucun argument. Comme son nom l'indique, le rôle d'un destructeur est de détruire («désinstancier») un objet. Un destructeur effectue dans l'ordre :

1. exécute le code dans le corps du destructeur ;
2. appelle le destructeur de chaque variable d'instance ;
3. appelle le destructeur de la ou des classes héritées.

Déclaration

```
class Point {
  public:
    Point(); // constructeur sans argument
    Point(double x, double y);
    ...
};
```

Définition

```
Point::Point(){
    x = y = 0.0;
}
Point::Point(double x_, double y_)
 : x(x_), y(y_) // le deux-points (":") est pour l'initialisation
{
}
```

Classe avec gestion de mémoire

```
class Tableau10int{
  public:
    Tableau10int();
    ~Tableau10int();
  private:
    int* elements;
};
```

Constructeur et Destructeur

```
Tableau10int::Tableau10int() {
    elements = new int[10];
}
Tableau10int::~Tableau10int() {
    delete [] elements ;
}
```

Héritage et Fonctions virtuelles

```
class FormeGeometrique{
  public:
    virtual double aire() = 0;
};
class Carre : public FormeGeometrique{
  public:
    Carre(double dimension) { m_dimension(dimension); }
    virtual double aire() {return m_dimension*m_dimension; }
  protected:
    int m_dimension;
};
class Rectange : public FormeGeometrique{
  public:
    Rectange(double h, double l) { m_hauteur(h); m_largeur(l); }
    virtual double aire() {return m_hauteur*m_largeur; }
  protected:
    int m_hauteur, m_largeur;
};
```

Mot clé `this`

- Le pointeur `this` pointe sur l'objet courant.
- C'est le **paramètre implicite**.

```
class A{
  public:
    int f(int v=0);
  private:
    int x;
};
int A::f(int v){
    int t = this->x; // int t=x;
    this->x = v; // x=v;
    return t;
}
```

```
int main(){
  A a1(2), a2(6);
  int w = a1.f(33);//dans A::f(int v), this=&a1
  int z = a2.f(10);//dans A::f(int v), this=&a2
  return 0;
}
```

Mot clé `friend` (relations d'amitié directionnelles)

```
class Point{
  public:
    Point(double x=0, double y=0);
  private:
    double x, y;
  friend class Rectangle; // Certains bris d'abstraction sont parfois nécessaires ou comodes.
};
class Rectangle{
  public:
    Rectangle(const Point& p1, const Point& p2);
    double perimetre() const;
    double aire() const;
  private:
    Point p1, p2;
};
double Rectangle::perimetre() const{
    return 2 * (fabs(p1.x-p2.x) + fabs(p1.y-p2.y));
}
double Rectangle::aire() const{
    return fabs(p1.x-p2.x) * fabs(p1.y-p2.y);
}
```

Mot clé `const`

- Le mot clé `const` est très important en C++.
- Similaire au mot clé `final` en Java, mais géré différement.
- L'objectif est d'aider le programmeur à éviter des bogues.
- Permet de spécifier que quelque chose ne doit pas être modifiée.
- Si le programme tente de modifier un objet `const`, le compilateur va générer une erreur.
- Important : `const` ne doit être utilisé comme un dispositif de sécurité. Le mécanisme de `const` est seulement une aide au programmeur.

Contexte d'utilisation de `const`

- Utile pour spécifier :
  - une constante (ex. : `const double pi=3.141592654;`)
  - que l'objet référencé par une référence doit être constant (ex. : `const Point& rp = p;`)
  - que l'objet pointé par un pointeur ne doit pas être modifié (ex. : `const Point* rp = &p;`)
- Très utilisé dans le passage de paramètres.

Exemple

```
double Point::distance(Point& p2){
    p2.x −= x;
    y−=p2.y;
    return sqrt(p2.x∗p2.x + y∗y);
}
int main(){
    Point p1 = ... , p2 = ...;
    double d = p1.distance(p2);
    // p1 et p2 ont été modifiés par Point::distance(...).
}
```

Exemple

```
double Point::distance(const Point& p2) const{
    p2.x −= x; // genère une erreur car p2 est const
    y−=p2.y;   // genère une erreur car ∗this est const
    return sqrt(p2.x∗p2.x + y∗y);
}
int main(){
    Point p1 = ... , p2 = ...;
    double d = p1.distance(p2);
}
```

Exemple

```
double Point::distance(const Point& p2) const{
    double dx = p2.x − x; // OK
    double dy = p2.y − y;  // OK
    return sqrt(dx∗dx + dy∗dy);
}
int main(){
    Point p1 = ... , p2 = ...;
    double d = p1.distance(p2);
}
```

Opérateurs

- Symboles : `+`, `-`, `*`, `/`, `!`, `+=`, `-=`, `^`, `<<`, `>>`, `()`, etc.
- Les opérateurs peuvent être appelés de façon «naturelle».
  - Exemple : au lieu d'écrire `int a = plus(b,c);`, on écrit tout simplement `int a=b+c;`.
  - Exemple : au lieu d'écrire `int a = divise(plus(b,c),2);`, on écrit tout simplement `int a=(b+c)/2;`.
- Nous pouvons faire la même chose avec nos propres types de données.
  - `Vecteur a(10,5), b(5,10); Vecteur c=a+b;` au lieu de `c=plus(a,b);`.

Surcharge d'opérateurs

- Les opérateurs sont des fonctions avec le mot clé `operator` comme préfixe.
- Différence : appel plus naturel qu'un appel de fonction.

vecteur.h

```
class Vecteur {
  public:
    Vecteur(double vx_=0, double vy_=0):vx(vx_),vy(vy_){}
    Vecteur& operator += (const Vecteur& v);
    Vecteur operator + (const Vecteur& v) const;
  private:
    double vx, vy;
};
```

Surcharge d'opérateurs

vecteur.cpp

```
#include "vecteur.h"
Vecteur& Vecteur::operator += (const Vecteur& autre) {
    vx+=autre.vx;
    vy+=autre.vy;
    return *this;
}
Vecteur Vecteur::operator + (const Vecteur& autre) const{
    return Vecteur(vx+autre.vx, vy+autre.vy);
}
```

Surcharge d'opérateurs amis (`friend`) (1)

vecteur.h

```
class Vecteur {
  public:
    //Vecteur operator + (const Vecteur& v) const;
  private:
    double vx, vy;

  friend Vecteur operator + (const Vecteur& v1, const Vecteur& v2)
      const;
};
```

mot clé `friend`

- Le mot clé `friend` déclare une relation d'amitié directionnelle.

Surcharge d'opérateurs amis (`friend`) (2)

vecteur.cpp

```
// La fonction suivante n'est pas une fonction membre de Point,
// mais une fonction globale.
Vecteur Vecteur::operator + (const Vecteur& v1, const Vecteur& v2) const{
   return Vecteur(v1.vx+v2.vx, v1.vy+v2.vy);
}
```

main.cpp

```
int main(){
    Vector v1(2, 3);
    Vector v2(4, 8);
    Vector v3 = v1 + v2;
}
```

Opérateurs << et >> pour les E/S

point.h

```
class Point{  private:
    double x, y;
    friend std::istream& operator >> (std::istream& is, Point& p);
    friend std::ostream& operator << (std::ostream& os, const Point& p);
};
```

mot clé `friend`

- Le mot clé `friend` déclare une relation d'amitié directionnelle.
- Les fonctions `operator >>(...)` et `operator <<(...)` ne sont pas des fonctions membres de la classe `Point`.
- Il s'agit d'une déclaration d'amitié. Ces fonctions étant amis avec `Point`, elles ont accès à ses membres `private`.

Opérateurs << et >> pour les E/S

point.cpp

```
std::istream& operator >> (std::istream& is, Point& p){
    char parouvr, vir, parferm;
    is >> parouvr >> p.x >> vir >> p.y >> parferm;
    assert(parouvr=='(' && vir==',' && parferm==')');
    return is;
}
std::ostream& operator << (std::ostream& os, const Point& p){
    os << "(" << p.x << "," << p.y << ")";
    return os;
}
```

Exercice d'abstraction

- Un type d'objet doit savoir comment se lire et s'écrire...
- Mais doit faire abstraction des types des objets qui le composent.

```
class Immeuble {
  public:
  private:
    string nom;
    Point  position;
    double hauteur;
    int    nbclients;

    friend std::istream& operator >> (std::istream& is, Immeuble& im);
};
```

Mauvaise approche...

```
std::istream& operator >> (std::istream& is, Immeuble& im){
   is >> im.nom;
   // Debut mauvais code
   char parouvr, vir, parferm;
   is >> parouvr >> im.position.x >> vir >> im.position.y >> parferm;
   assert(parouvr=='(' && vir==',' && parferm==')');
   // Fin mauvais code
   is >> im.hauteur;
   is >> im.nbclients;
   return is;
};
```

C'est une mauvaise idée d'aller lire directement le point. Il faut traiter un point de façon abstraite.

Bonne approche...

```
std::istream& operator >> (std::istream& is, Immeuble& im){
   is >> im.nom;
   is >> im.position; // Appel operateur >> pour Point
   is >> im.hauteur;
   is >> im.nbclients;
   return is;
};
```

Chaîne d'appels à >>

Version en plusieurs énoncés

```
istream& operator>>(istream&
      is, Immeuble& im)
{
   is >> im.nom;
   is >> im.position;
   is >> im.hauteur;
   is >> im.nbclients;
   return is;
};
```

Version en un seul énoncé

```
istream& operator>>(istream&
      is, Immeuble& im)
{
   is >> im.nom
      >> im.position
      >> im.hauteur
      >> im.nbclients;
   return is;
};
```

Chaîne d'appels à >>

Pourquoi `return is;` ?

Version en un seul énoncé

```
istream& operator>>(istream& is,
        Immeuble& im){
   is >> im.nom
      >> im.position
      >> im.hauteur
      >> im.nbclients;
   return is;
};
```

Équivalence d'appels

```
istream& operator>>(istream& is,
        Immeuble& im){
   operator>>(
     operator>>(
       operator>>(
         operator>>(is, im.nom),
         im.position),
       im.hauteur),
     im.nbclients);
   return is;
};
```