Programmation C++ (Examen)

COMPILATION

- compilation simple : `g++ -std=c++11 prog.cc -o test.exe`
- compilation par morceaux : `g++ -std=c++17 -c truc.cc`
  `g++ -std=c++17 truc.o prog.cc -o test.exe`
- exécution : `./test.exe`

Pour éviter les problèmes de double compilation, dans le .h : `#ifndef truc_h #define truc_h #endif`

TYPES

- Lambda fonctions: `-std=c++14 <algorithm>`

```
auto NOM = [capture](TYPE1 arg1, …) -> TYPE_RETOUR {…};
```

Le type de retour n'est pas utile si le compilateur peut le retrouver tout seul.
La capture permet de modifier les variables extérieures :
`[=]` : toutes les variables nécessaires sont capturées par copies
`[&]` : toutes les variables nécessaires sont capturées par références
`[a,b,&c]` : a et b par copie, c par référence
`[=,&c]` : toutes par copie sauf c
Avantages des lambdas fonctions : optimisation du compilateur, utilisation locale et portée locale

- Complexes `<complex>` `complex<double> z = 1.+2i; z.real(); z.imag(); z.abs()`
- Paires `<utility>` `p = std::make_pair(a,b); p.first(); p.second()`
- Algorithmes `<algorithm> <numeric>` `std::all_of; none_of; count; find_if; fill; swap..`

`int n = std::count(v.begin(), v.end(), 3)` où `v` est un vecteur de `int`
`int n = std::count_if(v.begin(), v.end(), f)` où `f` est une lambda fonction qui retourne un `bool`

- Chaînes de caractères `<string>` `std::string s("abcdef"); s[i]; s.size(); s+='a'; s1+s2; s.insert(i,s1); std::getline(std::cin,s); s.erase(i); s.erase(i,p)`
- Conteneurs `<vector> <set> <map> <list> <deque>`

<vector>	<list>	<map>	<queue>
Tableau statique : contigu en mémoire	Tableau dynamique : non contigu en mémoire, chaque élément possède les addresses du précédent et du suivant	Conteneur trié associatif contenant des paires clé-valeur avec des clés uniques. Les clés sont triées en utilisant la fonction de comparaison Compare	File d'attente

+ Opération d'accès rapide O(1)
- Insertion lente O(N) car il faut trouver une autre place plus grande en mémoire et tout copier
	+ Insertion/suppression rapides O(1)
	- Accès lent O(N)
	- Pas d'opérateur d'accès direct (trop coûteux)
		+ Recherche, suppression, insertion en O(log(N))
		+ unordered_map en O(1)
			`priority_queue<int, vector<int>, Compare> PQ;`
			
			`struct Compare { bool operator()(a,b)}`

L'équivalent de `push_front()` est `insert()`	`std::list<TYPE>::iterator it;`	`std::map<int,int> A; A[1]=1000; A.find(1) A.insert(pair<int,int>(1,1000)); A.at(1)`	`std::queue<int> Q; Q.push(2); Q.pop();`

TEMPLATES

```
template <typename TYPE, class NOM_CLASSE, int n>
PROTOTYPE DE LA FONCTION {…}
template<typename TYPE, class NOM_CLASSE, int n>
class NOM_CLASSE {…}
```

Appel de fonction
`fonction<TYPE1, TYPE2>(args)`
Déclaration d'objet
`CLASSE<TYPE1, TYPE2> NOM_VARIABLE`

```
template <typename U>
class A { ... friend template<typename V> void f(A<V>); };
```

→ Toutes les fonctions `void f(A<U>)` sont amies avec toutes les classes `A<V>` pour toutes valeurs de `U` et `V`

```
template <typename U>
class A { ... friend void f<U>(A); };
```

→ Chaque fonction `void f(A<U>)` est amie avec la seule classe `A<U>` associée.

```
template <typename T>
void Print(T x) {…}
```

spécialisation →

```
template <>
void Print(int x) {…}
```

```
template <int N, class A >
class C {…}
```

```
template<int N>
class<N, int>{…}
```

POINTEURS

Un pointeur est une variable dont la valeur est l'adresse d'une autre variable. Il permet la manipulation d'adresses.

```
int n = 34;
int* p = nullptr;
int* q = &n; ← affectation de l'adresse de n à q
std::cout << *q << std::endl; // 34
*q = 1; ← change la valeur de n = 34 à n = 1
std::cout << *q << std::endl; // 1
int* const k; ← adresse non modifiable mais valeur de *k changeable
const int* l; ← pointeur vers un entier non modifiable
p = new int[6]; ← plage de mémoire plus grande
q = new double;
std::cout << p[i] << (*p+i) << std::endl; ← accès au i-ème élément
delete [] p;
delete q;
```

- Lien avec les tableaux
  - Tableau statique : int q[5] → Allocation de la mémoire pour 5 objets de type int stocké consécutivement mais gestion de la mémoire automatique. De plus, on fige la nature q.
  - Tableau dynamique : int* q[5] → On réserve/libère la mémoire. Possible de réutiliser un pointeur pour un usage différent dans la suite du programme et qu'on évite les erreurs de segmentations (si l'allocation de mémoire échoue)
- Dans les classes

```
class Matrice {
private :
      unsigned int n;
      double* coeffs;
public :
      Matrice(const Matrice & A) { ← constructeur par copie
            this->n = A.n; this->coeffs = new double[n*n]
            for (unsigned int i=0; i<n; i++) this->coeffs = A.coeffs[i];};
      ~Matrice(){delete [] coeffs;} ← destructeur
};
```

- Passage de paramètres

```
int i = 3; int j = 7;
swap(&i,&j);
std::cout << i << j << std::endl; // 37
```

```
void swap(int* m, int* n) {
      int t = *m; *m *= *n; *n = t;
}
```

- Retour de valeur (allocation dynamique)

```
int* p = f(); *p = 3;
std::cout << *p << std::endl; // 3
```

```
int* f() {
      int i = 7; return &i;
};
```

TABLEAUX DIMENSION 1 (statique)

```
int t[6] = {1,2,3,4,5,6};
t[2] = 7;
```

- Passage de paramètres

Version constante : marche pour les tableaux de taille 6

```
double sum(double t[6]){…}
```

Version universelle : marche pour n'importe quelle taille

```
double sum(double t[], int size){…}
```

TABLEAUX DIMENSION 2 (statique et dynamique)

- Méthode 1

```
const int nlin = 3;
const int ncol = 4;
double t[nlin][ncol] = {{1,2,3,4}, {…}, {…}};
double sum(double matrix[][ncol], int m, int n){…};
```

• Méthode 2

```
const int nlin = 3;
const int ncol = 4;
double t[nlin*ncol] = {…};
init(t, nlin, ncol);
```

```
void init(double t[], int m, int n){
      for (int i=0; i<m; i++){
            for (int j=0; j<n; j++){
                  t[i*ncol+j] = 1/(i+j+1);}}}
```

• Méthode 3 (dynamique)

```
double *t[nlin];
for (int i=0; i<nlin; i++){
      t[i] = new double[ncol]}
```

```
void init(double *t[], int m, int n){
      for (int i=0; i<m; i++){
            for (int j=0; j<n; j++){
                  t[i][j] = 1/(i+j+1);}}}
```

CLASSES

```
class NOM_CLASSE {
private: protected:
      TYPE1 champ1;
public:
      // Constructeurs: par défaut, par paramètres, par copie, move
      // Accesseurs, mutateurs, destructeur
      // Operateur d'affectation, de comparaison, d'affichage
};
```

• `private` : visible par le programmeur, manipulé par le reste de la classe, inaccessible de l'extérieur de la classe.
• `protected` : visible par le programmeur, manipulé par le reste de la classe, accessible par les classes filles.
• `public` : utilisable par l'utilisateur pour les objets de type `NOM_CLASSE`

• Attributs statiques

Lorsque qu'un attribut est `static`, cela signifie que la variable n'existe qu'en un seul exemplaire, elle est globale à la classe en quelque sorte. Autrement, chaque objet du type de la classe dispose de sa propre copie.
Si le membre statique est `public` :

```
// dans le fichier prog.h
class Class {
public:
      static const int x = 0;
      initialisation ici uniquement si static const int
      static int x;
};
```

```
// dans le fichier prog.cc
int Class::x = 0;
// dans le main()
Class::x;
```

Si le membre statique est `private` :

```
// dans le fichier prog.h
class Serial {
private:
      static int number;
public:
      static int getNumber() {return number;}
};
```

```
// dans le fichier prog.cc
int Serial::number = 0;
// dans le main()
Serial::getNumber();
```

! Pour la compilation, pas besoin d'initialiser la variable mais problème à l'exécution.

• Fonction membres statiques

C'est une fonction membre déclarée qui a la particularité de pouvoir être appelée sans devoir instancier la classe. Elle ne peut utiliser que des variables et des fonctions membres `static` elles aussi, c'est-à-dire qui ont une existence en dehors de toute instance.

```
// dans prog.h
class A {
public:
      A(int n = 0) : v(n) {}
      void f() { ++v; ++w; } ← peut accéder aux champs static et non static
      static void g() { ++w; } ← ne peut accéder qu'aux champs static
private:
      int v;
      static int w;
};

// dans prog.cc
int A::w = 0;
```

```
// dans main()
int main() {
 A a;
 a.f(); // Call non static method with objet
 A::f();//ERROR: Non static method must be called with an object
 a.g();          // Call static method with objet
 A::g();         // Call static method with class name
}
```

HERITAGE

• Héritage multiple

```
class A { public : void f();};
class B {public: void f();};
class C {public: void g(){f();}}
```

Il y a ambiguïté donc il faudra plutôt appeler `A::f()` ou `B::f()`. Par contre, si `f` est définie directement dans la classe C, elle a la priorité et pas de problème.

Base class member access specifiers	Visibility of base class member in derived class		
	Public derivation	Protected derivation	Private derivation
Private	Invisible	Invisible	Invisible
Protected	Protected	Protected	Private
public	Public	Protected	Private

```
class B: public A, public A {} ← Erreur
class A {};
class B: public virtual A {}; class C: public virtual A{};
class D: public B, public C {} ← Ok, c'est l'héritage en diamant
```

• Types

- Type statique : type déclaré dans le code source, déterminé à la compilation
- Type dynamique : type de l'objet en mémoire pointé ou référencé
- Fonction virtuelle : fonction dans la classe mère qui peut être redéfinie dans les classes filles.
- Fonction virtuelle pure : fonction dans la classe mère qui **doit** être définie dans toutes les classes filles.

```
virtual TYPE_RETOUR & methode() const = 0;
```

- Fonction `override`: on impose qu'elle doit dériver d'une autre fonction
- Fonction `friend`: elle n'est pas une méthode associée à un objet mais une fonction qui peut utiliser ses attributs.

Par défaut le type statique à la priorité sur le type dynamique **SAUF** si la méthode est `virtual`. On peut aussi choisir soi-même la méthode avec `var.classe::methode()`

```
class Polygone {
  virtual int val() {…}}
class Carre : public Polygone {
  int val() {…}}                    type statique
class A {
  virtual void g() const;
  void k();}
class B: public A {
void g() override; ← Erreur : ne dérive pas de A::g() car pas la même signature
void k() override; ← Erreur : A::k() n'est pas virtuel}
friend std::ostream& operator<<(std::ostream& flux, Classe const& objet)
```

```
Carre C;
Polygone P;
Carre *p = &C;          Carre
Polygone *r = &C;       Carre
Polygone *t = &P        Polygone
Carre &q = C;           Carre   type dynamique
```

• Héritage vs délégation

```
class A {…}                    class A{…}
class B: public A {…}          class B {private: A objetA; …}
```

• Classe abstraite : c'est une classe qui possède au moins une fonction membre virtuelle pure.

SURCHARGE D'OPERATEUR CONST

```
class Test {
public:
      Test(int v) : val(v) {}
      // accesseur non const
      int& access() { return val; }
      // accesseur const
      const int& access() const { return val; }
private:
      int val;
};
```

```
int main() {
 // objet non const
 Test t1 = 1;
 // objet const
 const Test t2 = 2;
 cout << t1.access() << " " << t2.access() << endl;
 // appel de l'accesseur non const
 t1.access() = 3;
 t2.access() = 4;// ERROR : car le type retour est const
 cout << t1.access() << " " << t2.access() << endl;
}
```

MOVE CONSTRUCTOR + ASSIGNMENT OPERATOR

```
MemoryBlock(MemoryBlock&& other): _data(nullptr), _length(0){
   _data = other._data;
   _length = other._length;
   other._data = nullptr;
   other._length = 0;
}
MemoryBlock& operator=(MemoryBlock&& other){
   if (this != &other){
      delete[] _data;
      _data = other._data;
      _length = other._length;
      other._data = nullptr;
      other._length = 0;
   }
   return *this;
}
```

ITERATEUR

```
class Iterator {
public:
      Iterator(T c) : current(c) {}
      bool operator==(Iterator it) const { return current == it.current; }
      bool operator!=(Iterator it) const { return current != it.current; }
      bool operator<(Iterator it) const { return current < it.current; }
      const T& operator*() { return current; }
      Iterator& operator++() { ++current; return *this; }
private:
      T current;
};
Iterator begin() { return Iterator(min); }
Iterator end() { return Iterator(max); }
```

SMART POINTERS

```
class MyObj {
 public:
  explicit MyObj(int seed);
  ~MyObj();

  // NOT default- and copy-constructible!
  MyObj() = delete;
  MyObj(const MyObj&) = delete;

  int Value() const { return value_; }
 private:
  const int value_;
};
```

```
int ComputeMedianMyObj(int N){
    int median;
    {
        std::vector<int> vec(N);
        for (int i=0; i<N; i++){
            vec[i] = MyObj(i).Value();
        }
        std::sort(vec.begin(), vec.end());
        median = vec[N/2];
    }
    return median;
};
```

CLASSE ABSTRAITE

```
class Sequence {
public:
    Sequence(): count_(0){};
    virtual double Value() const =0;
    void Next();
    virtual void InternalNext() =0;
    int Step() const;
    virtual ~Sequence(){};
protected:
    int count_;
};
```

```
// 1, 1, 2, 3, 5, 8, ...
class FibonacciSequence : public Sequence {
public:
    FibonacciSequence();
    double Value() const;
    void InternalNext();
private:
    int step_;
    double current_;
    double prev_;
};
```

```
int ParseInt(const char* str);

using std::exception;
class NullPtrException : public exception {};
class EmptyStrException : public exception {};
class BadFormatException : public exception {
 public:
  explicit BadFormatException(const char* data) : data_(data) {}
  // See http://www.cplusplus.com/reference/exception/exception/what/
  const char* what() const throw() override { return data_; }
 private:
  const char* data_;
};
class OverflowException : public exception {
 public:
  explicit OverflowException(const char* data) : data_(data) {}
  const char* what() const throw() override { return data_; }
 private:
  const char* data_;
};

int ParseInt(const char* str){
    if (str==nullptr) throw NullPtrException();
    const char* p;
    std::unordered_set<char> Digits {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
    int nbChar = 0;
    bool negative_value = false;
    const char* max = "2147483647";
    const char* min = "-2147483648";
    for (p= str; *p != '\0'; p++){
        nbChar++;
        if (nbChar==1 && *p == '-') negative_value = true;
        if (((negative_value) && (nbChar>11))||((!negative_value) && (nbChar>10)))
throw OverflowException(str);
        if (*p == ' ') throw BadFormatException(str);
        if ((*p == '-') && (nbChar != 1)) throw BadFormatException(str);
        if ((Digits.find(*p) == Digits.end()) && (*p != '-')) throw
BadFormatException(str);
        if (((*p == '0') && (nbChar==1)) && *(p+1) != '\0') throw
BadFormatException(str);
        if ((*p == '0') && (*(p-1) == '-')) throw BadFormatException(str);
    }
    if (nbChar==0) throw EmptyStrException();
    if (((negative_value) && (nbChar==11))||((!negative_value) && (nbChar==10))) {
        for (int i=0; i<nbChar; i++){
            if (negative_value && *(str+i)>*(min+i)) throw OverflowException(str);
            if (!negative_value && *(str+i)>*(max+i)) throw OverflowException(str);
        }
    }
    int number;
    std::string string = str;
    std::istringstream stream(string);
    stream >> number;
    return number;
};
```