



Chapitre 7

struct, enum et std::pair

Structure du chapitre



- **struct**
 - Définition, déclaration, initialisation et affectation
 - Accès aux membres
 - Passage en paramètre, valeur de retour
 - Comparaison
 - Récupération des membres
- **std::pair**
- **enum**
 - Définition, déclaration, initialisation et affectation
 - Limitations
 - enum class
 - Passage en paramètre, valeur de retour
 - Comparaison

```
struct Section {  
    int number;  
    string name;  
};  
  
struct Chapter {  
    string title;  
    Section section1;  
    Section section2;  
    Section section3;  
};  
  
int main() {  
    Chapter chapter5 {  
        .title { "struct, enum et pair"},  
        .section1 { 1, "struct" },  
        .section2 { 2, "std::pair" },  
        .section3 { 3, "enum" }  
    };  
}
```



1. struct

struct – définition



- Le langage C/C++ permet de définir des **types structurés** grâce au mot-clé **struct**
- On définit d'abord un **modèle de structure** (type), puis on déclare autant **d'instances** (variables) que nécessaire
 - NB : souvent les instances sont aussi appelées **structures**
- Une structure regroupe un ou plusieurs éléments appelés **membres** ou **champs**, qui peuvent avoir **des types différents**
 - chaque membre a un **nom**, qui permet d'y accéder
 - contrairement aux tableaux où on utilise les index [n]



struct – définition

La forme générale d'une déclaration **struc**

- **nom_de_type** (optionnel), donne un **nom au type** structuré, ce qui permet de le réutiliser plus tard
- **declaration_membre**, déclaration du membre, séparés par un « ; »
- **Identificateur** (optionnel), déclare des **variables** de ce type

```
struct [ nom_de_type ] {  
    declaration_membre;  
    [ declaration_membre; ... ]  
} [ identificateur, ... ];
```

On peut utiliser une *majuscule* initiale pour le modèle de structure.

Les membres (champs, en *minuscules*) sont stockés en mémoire dans l'ordre où ils ont été définis.



struct – exemples

- Les membres peuvent être du même type .. ou pas
- Une structure peut contenir un objet, un membre structuré ...
- ... et les membres peuvent être **const** et/ou avoir une **valeur par défaut**

```
struct Coord {  
    double x, y, z;  
};
```

```
struct Date {  
    int jour;  
    int mois;  
    int annee;  
};
```

```
struct Personne {  
    string nom;  
    int age;  
    Date naissance;  
};
```

```
struct UneStruct {  
    const char cste = 'A';  
    int entier;  
    double* ptr = nullptr;  
};
```

[C++ Core Guideline C.12](#): Don't make data members **const** or references in a copyable or movable type



- Comme d'habitude, une déclaration se fait sur la base du **type** et peut être **initialisée**.
- A défaut, les membres ne seront pas initialisés à moins qu'un constructeur le fasse
- Lors d'une initialisation partielle (**agrégat**), les valeurs sont utilisées par **position**. Les valeurs manquantes sont mises à « 0 »

```
Date date1;                                // ??, ??, ????
Date date2 = {1, 2, 2019};                  // 1, 2, 2019
Date date3 = {1, 2};                        // 1, 2, 0

Personne Vide;                            // «», ??, ??, ??, ???
Personne Anna = {"Anna", 4, date2};        // «Anna», 4, 1, 2, 2019
Personne Jean = {"Jean", 18, {11, 3, 2005}}; // «Jean», 18, 11, 3, 2005
Personne Paul = {"Paul", 27, 31, 7, 1996 }; // «Paul», 27, 31, 7, 1996
```



struct – déclaration / affectation

- L'**initialisation** peut également se faire par **nom** ce qui permet d'omettre certains membres

```
Date date4 = { .jour=13, .mois=1, .annee=2007}; // 13, 1, 1998  
Date date5 = { .jour=13, .annee=1998}; // 13, 0, 1998
```

Note : Selon la norme C++20, même en nommant les membres, il faut respecter leur ordre pour l'initialisation. La plupart des compilateurs permettent un ordre quelconque pour des membres nommés, comme autorisé par la norme C99 du langage C.

- L'**affectation** globale (tous les membres) se fait avec le symbole « = »

```
Date date1; // ??, ??, ???  
Date date2 = {1, 2, 2019}; // 1, 2, 2019  
date1 = date2; // 1, 2, 2019
```



struct – accès aux membres

- L'accès aux membres d'une structure se fait en combinant le nom de l'instance et le nom du membre séparés d'un « . » (en **lecture** comme en **écriture**)

```
// une date
cout << date1.mois;
date1.jour = 12;
```

```
// référence sur une date
Date& refDate = date1;
cout << refDate.jour;
refDate.jour = 12;
```

- Lorsque qu'un **pointeur contient l'adresse d'une structure**, il faut d'abord **déréférencer** le pointeur avec l'opérateur « * » pour accéder au membre avec l'opérateur « . »
- Mais la priorité de l'opérateur « . » est supérieure à celle de l'opérateur « * »
L'opérateur « -> » fait ces deux opérations dans l'ordre et allège la syntaxe et la lecture

```
Date* ptrDate = &date1;      // pointeur sur une date
cout << (*ptrDate).jour;    // ( ) nécessaires
cout << ptrDate->jour;
ptrDate->jour = 12;
```



struct – accès aux membres

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right →
2	a++ a-- type() type{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! (type) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of ^[note 1] await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left ←



struct – passage en paramètre, retour

- Les structures s'utilisent comme les autres variables
- Elle peut être utilisée comme **paramètre**. Mais composée de plusieurs membres, un passage par **valeur (copie)** n'est **pas le meilleur choix**.
- Pour éviter que le système ait à recopier toute la structure, un passage par **référence** sera bien plus « **léger** » (pensez au **const** ☺)
- Une fonction ne peut **retourner qu'une seule valeur** mais peut tout de même retourner des valeurs groupées dans une structure

```
bool est_bissextile (const Date& date);
void decaller      (      Date& date, int n);

Date prochaine_date (const Date& date, int n);
```



struct – comparaison

- La comparaison entre deux structures n'est pas implémentée
- Il faudra prévoir une fonction de comparaison qui traitera les membres choisis
- Des fonctions opérateurs seront vues au chapitre 10

```
bool operator< (...);
```

```
date1 < date2 // ne compile pas  
date1 == date2 // ne compile pas
```

```
bool avant (const Date& lhs, const Date& rhs) {  
    if (lhs.annee != rhs.annee)  
        return lhs.annee < rhs.annee;  
    if (lhs.mois != rhs.mois)  
        return lhs.mois < rhs.mois;  
    return lhs.jour < rhs.jour;  
}
```

```
bool egale (const Date& lhs, const Date& rhs) {  
    return lhs.jour == rhs.jour and  
           lhs.mois == rhs.mois and  
           lhs.annee == rhs.annee;  
}
```



struct – récupération des membres

- Nous pouvons récupérer les types et les valeurs des membres d'une struct

```
struct Date {  
    int jour;  
    int mois;  
    int annee;  
};  
  
struct Personne {  
    string nom;  
    int age;  
    Date anniversaire;  
};  
  
Personne Anna = {"Anna", 4, {1, 2, 2019}};
```

```
auto [j, m, a] = Anna;  
  
// est équivalent à  
string j = Anna.nom;  
int m = Anna.age;  
Date a = Anna.anniversaire;
```



2. std::pair<T1, T2>



Structure en retour de fonction

- Une fonction doit permettre de calculer le **quotient** et le **reste** de la division de deux valeurs
- La fonction ne peut pas faire **return** de plusieurs valeurs
- .. mais avec une **structure**

```
struct Quotient_Reste
{
    int quotient;
    int reste;
};
```

```
void divmod(int numerateur,
            int denominateur,
            int& quotient,
            int& reste) {
    quotient = numerateur / denominateur;
    reste     = numerateur % denominateur;
}

Quotient_Reste divmod(int numerateur,
                      int denominateur) {

    Quotient_Reste qr;

    qr.quotient = numerateur / denominateur;
    qr.reste     = numerateur % denominateur;

    return qr;
}
```



Structure en retour de fonction

- ... en initialisant la structure directement

```
Quotient_Reste divmod(int numerateur,  
                      int denominateur) {  
  
    Quotient_Reste resultat {numerateur / denominateur,  
                            numerateur % denominateur};  
    return resultat;  
}
```

- ... et encore mieux, sans créer de variable intermédiaire

```
Quotient_Reste divmod(int numerateur,  
                      int denominateur) {  
  
    return Quotient_Reste {numerateur / denominateur,  
                          numerateur % denominateur};  
}
```



std::pair

- `std::pair<T1,T2>` permet de simplifier encore notre tâche en évitant de définir une structure `Quotient_Reste` qui ne servirait qu'ici

```
pair<int,int> divmod(int numerateur, int denominateur) {
    return {numerateur / denominateur, numerateur % denominateur};
}
```

- Elle est équivalente à `struct { T1 first; T2 second; };`, de sorte que le code appelant l'utilise comme suit

```
int n = 42, d = 12;
auto qr = divmod(n, d);
cout << n << " / " << d << " = " << qr.first << endl;
cout << n << " % " << d << " = " << qr.second << endl;

auto [q, r] = divmod(n, d);
cout << n << " / " << d << " = " << q << endl;
cout << n << " % " << d << " = " << r << endl;
```

42 / 12 = 3
42 % 12 = 6
42 / 12 = 3
42 % 12 = 6



std::pair - définition

- std::pair est un template générique mis à disposition par la librairie <utility>
- Un std::pair contient deux membres, first et second de types quelconques

```
// Le record mondial absolu est de 56,7° C. Il a été établi à Furnace
// Creek, dans la vallée de la mort, en Californie, le 10 juillet 1913

pair<Date, double> Furnace_Creek{{10, 7, 1913}, 56.7};

pair<Date, double> record_mondial = Furnace_Creek;
cout << "Record de " << record_mondial.second
    << " degrés en l'an " << record_mondial.first.annee << endl;

auto [date, temperature] = Furnace_Creek;
auto [jour, mois, an] = date;
cout << "Record de " << temperature
    << " degrés en l'an " << an << endl;
```

Record de 56.7 degrés en l'an 1913



3. enum



Nombres magiques ...

- Considérons le code ci-contre
- Il déborde de nombres magiques:
1, 2, 3, 4, 5, 6, 7

```
switch (day) {  
    case 1: [[fallthrough]];  
    case 2: [[fallthrough]];  
    case 3: [[fallthrough]];  
    case 4: [[fallthrough]];  
    case 5: cout << "Jour ouvrable";  
            break;  
  
    case 6: [[fallthrough]];  
    case 7: cout << "Week-end";  
            break;  
  
    default: cout << "N/A";  
             break;  
}
```



- Pour l'écrire plus proprement, on pourrait définir des constantes entières
- Mais C++ propose une solution plus élégante : les énumérations

```
const int LUNDI    = 1;
const int MARDI    = 2;
const int MERCREDI = 3;
const int JEUDI    = 4;
const int VENDREDI = 5;
const int SAMEDI   = 6;
const int DIMANCHE = 7;
```

```
enum JourDeLaSemaine {
    LUNDI = 1, MARDI,
    MERCREDI, JEUDI,
    VENDREDI, SAMEDI,
    DIMANCHE};
```



enum – définition

La forme générale
d'une déclaration **enum**

```
enum [ nom_de_type ]
{ const_enum [ = expression ]
[ , const_enum [ = expression ] ... ]
} [ identificateur [ , ... ] ];
```

- **nom_de_type**
optionnel, donne un **nom au type** énuméré, ce qui permet de le réutiliser plus tard
- **const_enum**
entre { } liste toutes **constantes d'énumération possibles**
- **= expression**
optionnel, donne une **valeur entière équivalente** à la constante d'énumération
Par défaut la première constante vaut 0, les autres la valeur précédente + 1.
- **identificateur**
après }, optionnel, **déclare des variables** de ce type



enum – déclaration

- Pour déclarer deux variables saison1 et saison2 pouvant prendre les valeurs PRINTEMPS, ETE, AUTOMNE, HIVER, on peut écrire :

```
enum Saison {PRINTEMPS, ETE, AUTOMNE, HIVER};  
enum Saison saison1, saison2; // comme en C
```

```
enum Saison {PRINTEMPS, ETE, AUTOMNE, HIVER };  
Saison saison1, saison2; // pas valable en C
```

```
enum Saison {PRINTEMPS, ETE, AUTOMNE, HIVER}  
    saison1, saison2; // une seule ligne
```

```
enum {PRINTEMPS, ETE, AUTOMNE, HIVER} saison1, saison2;  
// si on ne réutilise pas le type
```



enum – affectation

- Pour affecter la valeur ETE à saison1, on peut écrire

```
saison1 = ETE;  
saison1 = saison2;      // si saison2 vaut ETE;  
saison1 = (Saison)1;    // PRINTEMPS = 0, ETE = 1, ...  
saison1 = Saison(1);
```

Mais pas

```
saison1 = 1;           // possible en C
```

- Par contre, la **conversion de type** énuméré → entier se fait implicitement

```
int entier = saison1; // entier vaut 1  
int entier2 = ETE;   // entier2 vaut 1
```



enum – limitations

- Il n'y a **pas de vérification** lors de la conversion depuis un type numérique

```
enum Saison {PRINTEMPS, ETE, AUTOMNE, HIVER};  
Saison saison1 = Saison(12);  
// valide, mais seules les valeurs de 0 à 3 ont un sens.
```

- De plus, une constante d'énumération ne peut pas être réutilisée

```
enum Couleur {VERT, ROSE, BLEU};  
enum Fleur {MARGUERITE, ROSE, VIOLETTE};
```

```
error: redefinition of enumerator 'ROSE'  
enum Fleur {MARGUERITE, ROSE, VIOLETTE};  
^
```

```
note: previous definition is here  
enum Couleur {VERT, ROSE, BLEU};  
^
```



enum class

- Pour résoudre ces problèmes, C++11 a introduit des **types énumérés fortement typés** : `enum class`

```
enum class Saison {PRINTEMPS, ETE, AUTOMNE, HIVER};  
Saison saison1;
```

- Pour utiliser les constantes d'énumération, il faut spécifier le nom de l'énumération

```
saison1 = Saison::ETE; // il faut spécifier Saison::
```

- Le reste de l'affectation reste valable

```
saison1 = saison2;          // si saison2 vaut ETE;  
saison1 = (Saison)1;        // PRINTEMPS = 0, ETE = 1, ...  
saison1 = Saison(1);
```



enum class

- Il n'y a **plus de conversion implicite** de l'énumération vers les entiers

```
int entier = saison1;
```

```
error: cannot initialize a variable of type
'int' with an lvalue of type 'Saison'
    int entier = saison1;
    ^           ~~~~~~
```

- Mais la conversion explicite est possible

```
int entier = (int)saison1;
```

- Au contraire des **enum**, avec les **enum class**, les identificateurs des constantes d'énumération peuvent être réutilisées

```
enum class DirH {GAUCHE, CENTRE, DROITE};
enum class DirV {HAUT, CENTRE, BAS};
DirH horizontal = DirH::CENTRE;
DirV vertical   = DirV::CENTRE;
```



enum class – comparaison et paramètre

- Au final, les `enum` ou `enum class` ne sont que des entiers. Leurs comparaisons se font naturellement

```
enum class DirH {GAUCHE, CENTRE, DROITE};  
bool meme_direction (DirH lhs, DirH rhs) {  
    return lhs == rhs;  
}
```

- Le passage de paramètre se fera par copie voire par référence si besoin
- Nous pouvons retourner un `enum` ou `enum class`

```
Jour prochain_jour (Jour jour, int n = 1) {  
    return (Jour)((int)jour + n) % 7;  
}
```