Plan

- Introduction, rappels, généralités
- Approche impérative
- Approche fonctionnelle
- Approche objet
- Approche générative
- Compléments

Référence sur valeur gauche (rappel)

```
void f()
{
  int     i{ 3 };     // auto     i{ 3 };
  int & ri{ i };      // auto & ri{ i };

  i  =  5;
  ri = -2;

  int j{ 7 };
  ri = j;
  j = -1;

  //     int & rk{ 8 };
  const int & ck{ 8 };
}
```

Référence sur valeur droite

```
void g()
{
          int     i { 3 };
          int     i1{ i };
          int &   i2{ i };
    const int &   i3{ i };
//        int && i4{ i };

          int     j { 7 };
          int     j1{ j + 1 };
//        int &   j2{ j + 1 };
    const int &   j3{ j + 1 };
          int && j4{ j + 1 };
}
```

Passage d'arguments (5)

- Passage par ***référence sur valeur droite***

```
void f5( int && i ) {
  int k{ i };
  i = 3;
}
void g5() {
  f5( 5 );
  int j{ 8 };
  // f5( j );
  f5( j + 2 );
}
```

- Le paramètre est une temporaire qui va être détruite juste après l'appel

Opérateur d'affectation (rappel)

```
Pile & Pile::operator=( const Pile & p ) {
  Pile tmp{ p };
  swap( tmp );
  return *this;
}

void Pile::swap( Pile & p ) {
  using std::swap;
  swap( taille_, p.taille_ );
  swap( tab_,    p.tab_    );
  swap( index_,  p.index_  );
}
```

swap (1)

```
template< typename T >
void swap( T & t1, T & t2 ) {
  T tmp{ t1 };
  t1 = t2;
  t2 = tmp;
}
```

1 construction par copie
2 affectations par copie

swap (2)

- Une ***valeur gauche*** peut être « convertie » en ***référence sur valeur droite*** avec `std::move()`
- Seules possibilités après : destruction ou affectation

```
template< typename T >
void swap( T & t1, T & t2 ) {
  T tmp{ std::move( t1 ) };
  t1 =   std::move( t2 );
  t2 =   std::move( tmp );
}
```

Copie ≠ déplacement

```
class Pile {
public:
  // Construction par copie
  Pile( const Pile & p );
  // Affectation par copie
  Pile & operator=( const Pile & p );

  // Construction par déplacement
  Pile( Pile && p );
  // Affectation par déplacement
  //   Ne définir que si l'opérateur
  //   d'affectation utilise un passage
  //   par référence constante
  Pile & operator=( Pile && p );
  // ...
};
```

Implémentation pour Pile

```
Pile::Pile( Pile && p )
    : taille_{ p.taille_ }
    , tab_   { p.tab_    }
    , index_ { p.index_  }
{
  p.tab_ = nullptr;
}

Pile & Pile::operator=( Pile && p )
{
  swap( p );
  return *this;
}
```

Démo

GitLab

cours/complement/Move.cpp

Opérateurs C++

+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	++a --a a++ a--	a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	a == b a != b a < b a > b a <= b a >= b a <=> b	!a a && b a || b	a[b] *a &a a->b a.b a->*b a.*b	a(...) a, b ? :

Notation alternative : **and**, **bitand**, **and_eq**...

Opérateurs spéciaux : **xxx_cast**, **sizeof**, **typeid**, **decltype**, **new**, **delete**, **new[]**, **delete[]**, **noexcept**, **alignof**, **co_await**

Propriétés d'un opérateur

- Arité : nombre d'opérandes
- Position des opérandes : préfixé, infixé, postfixé
- Associativité (utilisation multiple d'un même opérateur) : à gauche ou à droite
- Priorité (utilisation d'opérateurs différents) : utilisation de parenthèses
- Ordre d'évaluation : non spécifié dans la majorité des cas

```
tab[i] = i++;    // ???
```

Surcharge d'opérateurs

- Opérateurs binaires
  - `operator Δ( a, b );`
  - `a Δ b` ou
  - `a.operator Δ( b );`
- Opérateurs unaires
  - `operator Δ( a );`
  - `Δ a` ou
  - `a.operator Δ();`

Opérateurs pouvant être surchargés (1)

- `+  −  ∗  /  %` (arithmétiques, binaires)
- `<<  >>  &  |  ^` (arithmétiques, binaires)
- `+  −  ~` (arithmétiques, unaires)
- `++  −−` (arithmétiques, unaires)
- `==  !=  <  <=  >  >=  <=>` (relationnels, binaires)
- `&&  ||` (logiques, binaires)
- `!` (logique, unaire)
- `=  +=  −=  ∗=  /=  %=` (affectations, binaires)
- `<<=  >>=  &=  |=  ^=` (affectations, binaires)

Opérateurs pouvant être surchargés (2)

- `[]` (indexation, binaire, n-aire en C++20)
- `->` `->*` (accès a un membre, unaire !)
- `&` `*` (adresse et indirection, unaires)
- **`new`** **`delete`** **`new[]`** **`delete[]`** (allocation dynamique)
- `()` (appel de fonction, n-aire)
- `,` (séquencement, binaire)
- `co_await` (coroutines)

Opérateurs arithmétiques (1)

```
class Nombre {
public:
  Nombre( int n );
  // ...

  // Approche objet
  Nombre operator+( const Nombre & n );

 // ...
};

// Approche fonctionnelle
Nombre operator*(
      const Nombre & n1, const Nombre & n2 );
```

Opérateurs arithmétiques (2)

```
Nombre n1{ 1 }, n2{ 2 };

n1 = n1 + n2;
n1 = n1 * n2;

n1 = n1 +  2;
n1 = n1 *  2;

n1 =  1 + n2;
n1 =  1 * n2;
```

Opérateurs d'incrémentation

- Approche objet ou fonctionnelle

```
class Nombre {
public:
  Nombre & operator++() {
    *this += 1;
    return *this;
  }
  // ...
};

Nombre operator++( Nombre & n, int ) {
  Nombre res{ n };
  ++n;
  return res;
}
```

Opérateurs de comparaison (1)

- Avant C++ 2020

```
bool operator==(
     const Nombre & n1, const Nombre & n2 );
bool operator<(
     const Nombre & n1, const Nombre & n2 );

Nombre n1{ 1 }, n2{ 2 };
bool b1{ n1 == n2 };

using namespace std::rel_ops;
bool b2{ n1 != n2 };
bool b3{ n1 <= n2 };
```

Opérateurs de comparaison (2)

- Depuis C++ 2020

```
// 4 valeurs possibles :
// less, equivalent, equal, greater
std::strong_ordering operator<=>(
    const Nombre & n1, const Nombre & n2 );

Nombre n1{ 1 }, n2{ 2 };
bool t1{ n1 == n2 };
bool t2{ n1 != n2 };
bool t3{ n1 >= n2 };

// Autres (pas de valeur equal) :
// std::partial_ordering : ajoute unordered
// std::weak_ordering
```

Opérateurs d'affectation

- Approche objet obligatoire

```
class Nombre {
public:
  Nombre & operator= ( const Nombre & n );
  Nombre & operator+=( const Nombre & n );
  // ...
};
```

- Écriture des opérateurs arithmétiques

```
Nombre operator+(
     const Nombre & n1, const Nombre & n2 ) {
  Nombre res{ n1 };
  return res += n2;
}
```

Opérateur d'indexation

```
class Tableau {
public:
  explicit Tableau( int taille );
  Tableau( const Tableau & );
  ~Tableau();
  Tableau & operator=( const Tableau & );

  double & operator[]( int indice ) {
    if( indice < taille_ )
      return tab_[indice];
    //...
  }
private:
  int taille_;
  double * tab_;
};
```

Les pointeurs intelligents

- L'expression

  a–>b

  est valide si a est un pointeur pointant sur un objet ayant un membre de nom b

  ou si la classe de a redéfinit l'opérateur –>

- La redéfinition de l'opérateur unaire –> :
  - approche objet obligatoire
  - fonction membre avec 0 argument

Exemple de pointeur intelligent

```
template< typename T >
class ptr {
public:
   ptr( T * p ) : p_{ p } {}
   ~ptr() { delete p_; }

   T * operator ->() { return  p_; }
   T & operator  *() { return *p_; }
private:
   T * p_;
};

void foo() {
   ptr< int > p1{ new int{ 1 }};
   ptr< Forme > p2{ new Rectangle{ ... }};
   p2->dessine();
}
```

std::shared_ptr

```
template< typename T >
class shared_ptr {
public:
  template< class Y >
  explicit shared_ptr( Y * p = nullptr );
  ~shared_ptr();

  template< class Y >
              shared_ptr( shared_ptr< Y > const & p );
  template< class Y >
  shared_ptr & operator =( shared_ptr< Y > const & p );

  T * operator ->() const;
  T & operator  *() const;
  // ...
};
```

std::weak_ptr

```
template< typename T >
class weak_ptr {
public:
  template< class Y >
  weak_ptr( shared_ptr< Y > const & p );
  ~weak_ptr();

  template< class Y >
               weak_ptr( weak_ptr< Y > const & p );
  template< class Y >
  weak_ptr & operator =( weak_ptr< Y > const & p );

  shared_ptr< T > lock() const;
  // ...
};
```

std::unique_ptr

```
template< typename T >
class unique_ptr {
public:
  explicit unique_ptr( T * p = nullptr );
  ~unique_ptr();
                unique_ptr( unique_ptr && p );
  unique_ptr & operator =( unique_ptr && p );

                unique_ptr( unique_ptr const & )
                                    = delete;
  unique_ptr & operator =( unique_ptr const & )
                                    = delete;

  T * operator ->() const;
  T & operator  *() const;
  // ...
};
```

Démo

GitLab

cours/complement/SmartPointer.cpp

Entrées/sorties : extension aux types utilisateurs (1)

```cpp
#include <iostream>
#include "Nombre.hpp"

int main() {
  Nombre n;
  std::cin >> n;
  std::cout << "carré de " << n
            << " = " << n * n
            << std::endl;
  return 0;
}
```

Entrées/sorties : extension aux types utilisateurs (2)

```
std::ostream & operator<<(
        std::ostream & out,
        const Nombre & n );

std::istream & operator>>(
        std::istream & in,
              Nombre & n );
```

new et delete

- Possibilité de redéfinir l'allocation dynamique au niveau d'une classe
  - Ces méthodes sont alors utilisées pour allouer la mémoire et la libérer pour les instances de cette classe
- Possibilité de surcharger l'allocation dynamique au niveau global en ajoutant des arguments
  - Exemple du *placement new* :

```
void * operator new( size_t, void * p ) {
   return p;
}
```

Démo

GitLab

cours/complement/New.cpp

Opérateurs de conversion

```
class Nombre {
public:
   // Conversion int vers Nombre
   Nombre( int n );

   // Conversion Nombre vers double
   double operator double() const;

   // ...
};
```

std::ranges, std::views

- Les algorithmes de la STL :
  - imposent l'utilisation systématique d'une paire d'itérateurs
  - sont difficilement combinables
- Nouvelle bibliothèque
  - incubée chez boost
  - Un *range* est itérable
  - Une *view* est une sélection d'une partie d'un *range*, sans recopie mais avec une éventuelle transformation, c'est un *range* aussi
  - Une view peut aussi générer ses valeurs à la demande.

Exemple STL

```
using namespace std;

auto const data = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto odd    = []( int i ) { return i % 2 == 1; };
auto square = []( int i ) { return i * i; };

vector< int > odds;
copy_if( begin( data ), end( data ),
         back_inserter( odds ), odd );

vector< int > squared;
transform( begin( odds ), end( odds ),
           back_inserter( squared ), square );

copy( squared.begin(), squared.end(),
      ostream_iterator< int >( cout, " " ));
cout << '\n';
```

Exemple C++ 2020

```
auto r1{ views::filter( data, odd )};
auto r2{ views::transform( r1, square )};
copy( r2.begin(), r2.end(),
      ostream_iterator< int >( cout, " " ));
cout << '\n';

// "pipe" syntax
ranges::copy(
        data
          | views::filter( odd )
          | views::transform( square ),
        ostream_iterator< int >( cout, " " ));
cout << '\n';
```

Démo

GitLab

cours/complement/Ranges.cpp

Démo

GitLab

cours/complement/Fibonacci.cpp

Histoire de sacs

- Une pomme ***est un*** fruit.
- Est-ce qu'un sac de pommes ***est un*** sac de fruits ?
  - quand je veux retirer un fruit ?
  - quand je veux ajouter un fruit ?
- Est-ce qu'un sac de fruits ***est un*** sac de pommes ?
  - quand je veux retirer une pomme ?
  - quand je veux ajouter une pomme ?

Démo

GitLab

cours/complement/Sac.cpp

Ressources

- http://www.cppreference.com
- http://www.cplusplus.com
- https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

- ! NIH
- http://www.boost.org
- GitHub...