Conversion de types

Conversion implicite et explicite

on a vu qu'il était possible que le compilateur puisse faire des converison automatiquement (on dit qu'elles sont implicite : http://en.cppreference.com/w/cpp/language/implicit_cast). Par exemple, quand on écrit :

```
long int l { 123 };
```

La littérale “123” est de type `int`, la valeur est d'abord convertie en long int avant d'être affectée à la variable l (en pratique, le compilateur sait faire directement cette conversion, cela n'a pas d'impact sur les performances)

Dans d'autres cas, la conversion va produire un avertissement (par exemple quand on perd des informations - narrowing ou arrondi) ou s'il n'existe pas de conversion implcite possible.

```
unsigned int u { -1 };     // avertissement, narrowing
int i { "hello, world!" }; // erreur, pas de conversion
implicite d'une chaîne en entier
```

Dans un conversion explicite (cast), on ajouter un code qui indique que l'on souhaite faire la conversion. Plusieurs opérateur de cast :

- static_cast, vérifié à la compilation, entre types compatibles ;
- dynamic_cast, vérifié à la compialtion entre classes parent (vu plus loin, avec l'héritage) ;
- reinterpret_cast, sans vérificatio (donc très dangeureux), également pour l'héritage.

Donc pour le moment, seul qui intéresse : static_cast. Fonction template,

il faut indiquer le type que l'on souhaite obtenir comme argument template et la valeur à convertir comme argument de fonction. En cas de cast explicite, le compilateur ne génère pas d'avertissement, il considère que l'on sait ce que l'on fait.

```
int i { static_cast<int>(-1) }; // ok, pas de narrowing
auto i = static_cast<int>(-1);  // ok, pas de narrowing
```

Il existe une ancienne syntaxe que l'on rencontre encore souvent (plus courte à écrire), mais qui ne fait pas de vérification, donc a éviter. http://en.cppreference.com/w/cpp/language/explicit_cast

```
int i { int(-1) }; // ok, pas de narrowing
auto i = int(-1);  // ok, pas de narrowing
```

Constructeur de conversion

Lorsque l'on écrit un constructeur qui peut prendre 1 argument (donc avec 1 paramètre ou plusieurs paramètres dont des paramètres par défaut), peut être implicitement utilisé par le compilateur pour faire une conversion.

```
struct A {
    A(int i) : i_(i) {}
};

int main() {
    A a { 123 }; // ok, conversion implicite, équivalent à :
                 // A a { A(123) }
};
```

Par défaut, il est préférable de bloquer les convesion implicite, sauf si cela a clairement un sens au niveau sémantique. Par exemple, si on écrit une classe Arme, qui prend un int pour les dégats, cela n'a pas de sens d'écrire : Arme a { 10 }; (on ne comprend pas à quoi correspond la valeur 10). Par contre, pour un complexe, écrire complex c { 10 } a un sens puisque 10 est un nombre complexe particulier (c'est un réel, donc un nombre complexe de la forme a+i*0)

Pour interdir l'appel d'un constructeur implicitement, il faut ajouter le mot clé `explicit` :

```
struct A {
    explicit A(int i) : i_(i) {}
};

int main() {
    A a { 123 };                 // erreur, implcite
conversion
    A a { static_cast<A>(123) }; // ok, conversion explicite
};
```

Opérateur de conversion

Opération inverse : convertir un type dans un autre. http://en.cppreference.com/w/cpp/language/cast_operator

```
class A {
    operator int();
};
```

autorise à écrire :

```
A a {};
int i { a }; // ok, conversion avec l'opérateur int()
```

Littérales utilisateur

Permet d'écrire un littérale correspond à un type crée par l'utilisateur. Par exemple, si on écrit A(int) et que l'on veut utiliser auto, il faudra écrire :

```
struct A {
    A(int); // opérateur de conversion implicite (puisque
pas le mot clé explicit)
};
```

```
auto i = 1;       // i de type int
auto a = A { 1 }; // a de type A
```

Littérale utilisateur permet d'écirre :

```
auto a = 1_a; // 1 de type A
```

le suffixe peut être n'importe quoi. Par exmple, pour une classe time (qui enregistre des secodnes) :

```
auto secondes = 1_s; // en interne, t = 1 (secondes)
auto minutes = 1_m; // en interne, t = 60 (secondes)
auto heures = 1_h; // en interne, t = 3600 (secondes)
```

A chaque fois le meme type (time), mais valeur différentes

signature :

```
template <char...> double operator "" _x();

long double operator "" _w(long double);
std::string operator "" _w(const char16_t*, size_t);
unsigned operator "" _w(const char*);

constexpr long double operator"" _deg ( long double deg )
{
    return deg*3.141592/180;
}
```

Par exemple, pour A :

```
struct A {
};

constexpr A operator"" _a (int i) {
    return A { i };
}

auto a = 1_a; // ok
```

Pour time :

```
struct time {
    int secondes;
};

constexpr time  operator"" _s (int i) {
    return time { i };
}

constexpr time  operator"" _m (int i) {
    return time { i * 60 };
}

constexpr time  operator"" _h (int i) {
    return time { i * 3600 };
}

auto s = 1_s; // ok
auto m = 1_m; // ok
auto h = 1_h; // ok
```

remarque : ajouté dans le C++14

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++