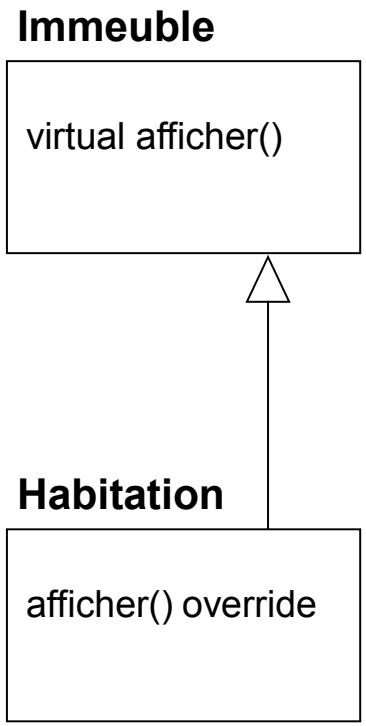
# Programmation orientée objet

Types de conversion

# Définition des classes Immeuble et Habitation

```
class Immeuble {
public:
        virtual void afficher();
};
```

```
class Habitation : public Immeuble {
public:
        void afficher() override;
};
```



# Upcasting

- Un objet de la classe dérivée peut être converti implicitement en un objet de la classe de base.
- Dans ce cas on parle de **upcasting**; on monte dans la hiérarchie de classe
- L'**upcasting** est aussi faisable dans le cas d'un pointeur ou d'une référence
- Ce type de conversion se fait naturellement puisque la classe dérivée est un objet de la classe de base
- Il y a seulement perte de spécificité

# Upcasting - Objet

```
void tester(Immeuble unImmeuble) {
   unImmeuble.afficher();
}

int main() {
   Immeuble   monImmeuble;
   Habitation monHabitation;
   monImmeuble = monHabitation;
   tester(monHabitation);
}
```

# Upcasting – Référence

```
void tester(Immeuble& unImmeuble) {
   unImmeuble.afficher();
}

int main() {
   Habitation monHabitation;
   Immeuble&  monImmeuble = monHabitation;
   tester(monHabitation);
}
```

# Upcasting - Pointeur

```
void tester(Immeuble* ptr){
    ptr->afficher();
}

int main() {
    unique_ptr<Habitation> ptrHabitation =
        make_unique<Habitation>();
    unique_ptr<Immeuble> ptrImmeuble =
        make_unique<Habitation>();
    tester(ptrHabitation.get());
}
```

# Downcasting

- On parle de **downcasting** quand on convertit un objet, un pointeur ou une référence de la classe de base vers un objet, un pointeur ou une référence de la classe dérivée; on descend dans la hiérarchie de classe
- Ce type de conversion est plus délicat à gérer puisqu'un objet de la classe de base ne connaît pas les spécificités de la classe dérivée

# Downcasting - Objet

- Le **downcasting** d'un objet n'est pas autorisé par défaut, il faut créer un constructeur de la classe dérivée qui accepte un objet de la classe de base:

```
class Habitation : public Immeuble {
public:
   Habitation();
   Habitation(const Immeuble& immeuble);
   void afficher() override;
};
```

# Downcasting - Objet

```
void tester(Habitation uneHabitation) {
      uneHabitation.afficher();
}

int main() {
   Immeuble monImmeuble;
   tester(monImmeuble);
}
```

# Downcasting – Référence & pointeur

- Le **downcasting** d'un pointeur ou d'une référence n'est légal que lorsque la classe réelle du pointeur ou de la référence est celle de la classe dérivée
- Une tentative de **downcasting** qui ne respecte pas cette condition pourrait causer une erreur d'exécution (« undefined behavior ») ou tout simplement produire un comportement inattendu
- Ce type de conversion n'est possible qu'en utilisant les opérateurs de conversion définies en C++

Des opérateurs de conversion sont définies en C++ qui nous permettent de faire des conversions de type de manière explicite. L'utilisation de ces fonctions est fortement conseillée lorsqu'il pourrait y avoir des problèmes liés à la conversion. Les opérateurs static_cast<> et const_cast<> font des conversions de type à la compilation, tandis que l'opérateur dynamic_cast<> fait une conversion de type à l'exécution.

Nous allons reprendre les classes Immeuble et Habitation :

```
class Immeuble {
public:
    virtual void afficher();
};

class Habitation : public Immeuble {
public:
    Habitation();
    void afficher() override;
    int getAtt() const;
private:
    int att_;
};
```

## static_cast

L'opérateur static_cast<> peut faire tout type de conversion standard (int en double, char* en void *, etc.):

```
int main() {
    char lettre = 'A';
    unsigned int asciiLettre = static_cast<unsigned int>(lettre);

}
```

Cet opérateur peut aussi faire la conversion de réferences ou de pointeurs, mais ne fait pas de vérification à l'exécution étant donné qu'il n'agit qu'à la compilation. Si on l'utilise avec des références ou des pointeurs, il faut être sûr que la conversion est légale sinon on risque d'avoir un comportement inattendu ou une erreur d'exécution. L'exemple suivant présente un downcast de pointeurs :

```
int main() {
    unique_ptr<Immeuble> ptrImmeuble = make_unique<Habitation>();
    Habitation* hab = static_cast<Habitation*>(ptrImmeuble.get());
}
```

Il est aussi possible de faire la même chose avec un downcast de références :

```
int main() {
    Habitation habitation;
    Immeuble& refImmeuble = habitation;
    Habitation& habref = static_cast<Habitation&>(refImmeuble);
}
```

## const_cast

Cet opérateur est utilisé pour modifier la constance d'une référence ou d'un pointeur. Ainsi, on peut passer d'une expression constante à une expression modifiable :

```
void tester(const Habitation& uneHabitation) {
    const_cast<Habitation&>(uneHabitation).afficher();
}
```

Étant donné que la méthode afficher n'est pas constante, elle ne peut pas être appelée sur une réference constante. Le const_cast nous permet de faire appel à cette fonction puisqu'il retourne une réference non constante vers l'objet uneHabitation. On peut faire la même chose avec un pointeur constant:

```
void tester(const Immeuble* ptr){
    const_cast<Immeuble*>(ptr)->afficher();
}
```

## dynamic_cast

Cet opérateur fait la conversion d'un pointeur ou d'une référence en faisant une vérification à l'exécution de la légalité de cette conversion. Le dynamic_cast peut être utilisé pour faire du upcasting, mais il est surtout utilisé pour faire du **downcasting** de pointeurs ou référence **sur des classes polymorphes**. Si la conversion n'est pas possible :

Il retourne un pointeur nul dans le cas d'une conversion de pointeurs

Il lance une exception dans le cas d'une conversion de référence.

Prenons l'exemple suivant :

```
int main() {
    vector<unique_ptr<Immeuble>> immeubles;
    immeubles.push_back(make_unique<Immeuble>());
    immeubles.push_back(make_unique<Habitation>());

    for (auto& immeuble: immeubles) {
        immeuble->afficher();
    }
}
```

Une conversion dynamique sera nécessaire afin d'atteindre l'interface spécifique de la classe Habitation :

```
int main() {
    vector<unique_ptr<Immeuble>> immeubles;
    immeubles.push_back(make_unique<Immeuble>());
    immeubles.push_back(make_unique<Habitation>());

    for (auto& immeuble: immeubles) {
        immeuble->afficher();
        if (Habitation * ptr = dynamic_cast<Habitation*>(immeuble.get())) {
            cout << ptr->getAtt() << endl;
        }
    }
}
```

Le dynamic_cast peut aussi être utilisé pour faire une vérification de type :

```
int main() {
    vector<unique_ptr<Immeuble>> immeubles;
    immeubles.push_back(make_unique<Immeuble>());
    immeubles.push_back(make_unique<Habitation>());

    int compteurHabitation = 0;
    for (const auto& immeuble: immeubles) {
        if (dynamic_cast<Habitation*>(immeuble.get())) {
            compteurHabitation++;
        }
    }
}
```

## explicit

Le mot clé explicit est utilisé pour s'assurer qu'un constructeur qui accepte un seul paramètre ne puisse pas être utilisé pour construire un objet de manière implicite. Prenons l'exemple suivant :

```
class Base {
public:
    Base() : att_(0) {}
    int getAtt() const;
private:
    int att_;
};

class Habitation : public Immeuble {
public:
    Habitation(Base att) : att_(att) {}

    void afficher() override {}
    Base getBaseAtt() const {
        return att_;
    }
private:
    Base att_;
};

int main() {
    Base base;
    Habitation hab = base;
    cout << hab.getBaseAtt().getAtt() << endl;
}
```

Le code présenté ici-haut compile parfaitement puisque le constructeur par paramètre est appelé implicitement lorsqu'on construit l'objet hab. En rendant le constructeur explicit, on s'assure que ce genre de manipulation causera une erreur de compilation :

```
class Habitation : public Immeuble {
public:
    explicit Habitation(Base att) : att_(att) {}

    void afficher() override {}
    Base getBaseAtt() const {
        return att_;
    }
private:
    Base att_;
};
```