

Faut-il bannir l'héritage en C++

D'après des propositions originales de Sean Parent

22 Juin 2015



Sean Parent

Principal Scientist at Adobe Systems

(Apple/Adobe/Google/Adobe)

ASL : Adobe Source Libraries <http://stlab.adobe.com>

Value Semantics and Concepts-based Polymorphism

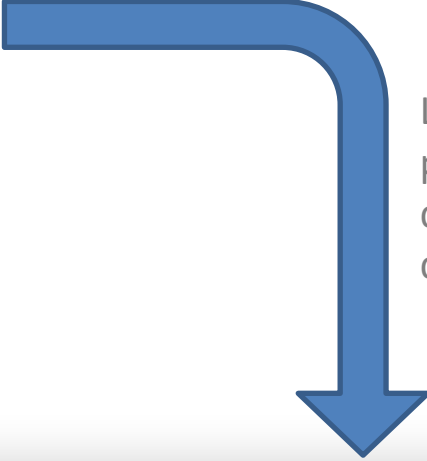
Présentation aux BoostCon 2012 (boost con=>cppnow)

<http://2012.cppnow.org/session/value-semantics-and-concepts-based-polymorphism/>

Polymorphisme d'inclusion



```
using object_t = int;
void draw(const object_t& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }
using document_t = vector<object_t>;
void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (const auto& e: x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```



L'héritage utilisé pour
permettre la construction
d'un document contenant
des objets de type différent

```
class object_t {
public:
    virtual ~object_t() {}
    virtual void draw(ostream&, size_t) const = 0;
};
using document_t = vector<shared_ptr<object_t>>;
void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (const auto& e: x) e->draw(out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```

```
class my_class_t : public object_t
{
public:
void draw(ostream& out, size_t position) const
{ out << string(position, ' ') << "my_class_t" << endl; }
/* ... */
};
int main()
{
document_t document;
document.emplace_back(make_shared<my_class_t>());
draw(document, cout, 0);
}
```

- Modification des sémantiques de copie, d'affectation et d'égalité de `document_t`
- Inefficacité : appel à `draw` et destructeur virtuel même dans un contexte non polymorphique, allocation dynamique, synchronisation
- Intrusif : impact sur le code client, création de type artificiel (`object_t`), besoin de Boxing sur les types à sémantique de valeur (e.g. `int`)

Inheritance is the base class of Evil

```
class object_t {  
public:  
private:  
struct concept_t {  
virtual ~concept_t() = default;  
virtual void draw_(ostream&, size_t) const = 0;  
};  
};
```

- Un concept est un ensemble de propriétés nécessaires pour l'algorithme en cours de définition
- Le type incarnant le concept utilisé par la bibliothèque est propre à son implémentation

```
class object_t {  
public:  
private:  
    struct concept_t {  
        virtual ~concept_t() = default;  
        virtual void draw_(ostream&, size_t) const = 0;  
    };  
    template <typename T>  
    struct model : concept_t {  
        model(T x) : data_(move(x)) { }  
        void draw_(ostream& out, size_t position) const  
        { draw(data_, out, position); }  
        T data_;  
    };  
};
```

- Un modèle est un type qui satisfait les exigences du concept
- Ici aussi le modèle générique reste un détail d'implémentation

Concepts-based polymorphism



```
template <typename T>
void draw(const T& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
public:
template <typename T>
object_t(T x) : self_(new model<T>(move(x)))
{ }
friend void draw(const object_t& x, ostream& out, size_t position)
{ x.self_->draw_(out, position); }

private:

struct concept_t {
virtual ~concept_t() = default;
virtual void draw_(ostream&, size_t) const = 0;
};

template <typename T>
struct model : concept_t {
model(T x) : data_(move(x)) { }
void draw_(ostream& out, size_t position) const
{ draw(data_, out, position); }
T data_;
};
unique_ptr<concept_t> self_;
};

using document_t = vector<object_t>;
void draw(const document_t& x, ostream& out, size_t position)
```

```
{
out << string(position, ' ') << "<document>" << endl;
for (const auto& e: x) draw(e, out, position + 2);
out << string(position, ' ') << "</document>" << endl;
}

class my_class_t {
/* ... */
};
void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
document_t document;
document.emplace_back(0);
document.emplace_back(string("Hello!"));
document.emplace_back(2);
document.emplace_back(my_class_t());
draw(document, cout, 0);

return 0 ;
}
```

- Pas d'héritage imposé aux types clients
- Séparation de l'implémentation des concepts, des types clients et de leur utilisation dans la bibliothèque

Regular type



```
class object_t {  
public:  
template <typename T>  
object_t(T x) : self_(new model<T>(move(x)))  
{ }  
object_t(const object_t& x) : self_(x.self_->copy_())  
{ }  
object_t(object_t&&) noexcept = default;
```

```
friend void draw(const object_t& x, ostream& out, size_t position)  
{ x.self_->draw_(out, position); }
```

```
private:  
struct concept_t {  
virtual ~concept_t() = default;  
virtual concept_t* copy_() const = 0;  
virtual void draw_(ostream&, size_t) const = 0;  
};
```

```
template <typename T>  
struct model : concept_t {
```

```
model(T x) : data_(move(x)) { }  
concept_t* copy_() const { return new model(*this); }  
void draw_(ostream& out, size_t position) const  
{ draw(data_, out, position); }  
T data_;  
};  
unique_ptr<concept_t> self_;
```

- Sémantique de copie « régulière »
- Copie profonde
- 2 objets réellement différents
- Pas de couplage caché

```
class object_t {  
public:  
template <typename T>  
object_t(T x) : self_(new model<T>(move(x)))  
{ }  
object_t(const object_t& x) : self_(x.self_->copy_())  
{ }  
object_t(object_t&&) noexcept = default;
```

```
object_t& operator=(const object_t& x)  
{ object_t tmp(x); *this = move(tmp); return *this; }  
object_t& operator=(object_t&&) noexcept = default;
```

```
friend void draw(const object_t& x, ostream& out, size_t position)  
{ x.self_->draw_(out, position); }
```

```
private:  
struct concept_t {  
virtual ~concept_t() = default;  
virtual concept_t* copy_() const = 0;  
virtual void draw_(ostream&, size_t) const = 0;
```

```
};  
  
template <typename T>  
struct model : concept_t {  
model(T x) : data_(move(x)) { }  
concept_t* copy_() const { return new model(*this); }  
void draw_(ostream& out, size_t position) const  
{ draw(data_, out, position); }  
T data_;  
};  
unique_ptr<concept_t> self_;
```

- Sémantique d'affectation
« régulière »
- 2 objets réellement différents
- Pas de couplage caché
- Copy&Move idiom

A lire !!!



Value Semantics and Concepts-based Polymorphism

<http://2012.cppnow.org/session/value-semantics-and-concepts-based-polymorphism/>



A lire absolument

Pour aller plus loin

Extension de l'idiome :

Runtime Concepts for the C++ Standard Template Library

Sean Parent, Mat Marcus - Adobe Systems, Inc.

Bjarne Stroustrup, Peter Pirkelbauer - Texas A&M University

<http://www.stroustrup.com/oops08.pdf>

Nantes C++ Meetup



Merci