
Hello
my name is
zeptolab
Viktor Korsun

Implementing class properties effectively

The dark side of C++

Property

Is something between
a class field and method.

A real-life example

```
class Text : public BaseElement
{
public:
    ZOBJ(Text);
    Text() : string(nullptr) { }
    int align;
    ZString *string;
    int stringLength;
    float drawOffsetY;
    FontGeneric *font;
    float wrapWidth;
    DynamicArray *formattedStrings;
    DynamicArray *multiDrawers;
    float maxHeight;
    bool wrapLongWords;
    virtual Text *initWithFont(FontGeneric *i);
...
```

A LOT of legacy code

```
Text* text;

...
text->setText(ZS(STR_LOC_OMNOM));
...
text->width = 42;
text->height = GetQuadOffset(IMG_OMNOM__top_offs).y;
...
text->draw();
```

Class declaration

```
class Bar
{
public:
    int size;
/* ... */
};

class Foo
{
    const int& getSize();
    void setSize(const int& size);
/* ... */

private:
    int size;
/* ... */
};
```

Class usage

```
Bar bar;
bar.size = 10;

Foo foo;
foo.setSize(10);
```

Desired beahvior

```
Bar bar;
bar.size = 10; //sets the member explicitly

Foo foo;
foo.size = 10; // sets the member implicitly through the setter
```

Possible implementation

```
template<typename T>
class Property
{
public:
    Property<T>& operator= (const T& v)
    {
        value = v;
        return *this;
    }

    operator const T& ()
    {
        return value;
    }

private:
    T value;
};
```

Possible usage

```
Property<int> p;

p = 3;
(p = 5) = (p == 3);

std::cout << p;
```

Going deeper

1. Properties have to call object's methods
2. Properties **must store nothing**, being a refined syntacic sugar

Straightforward implementation

```
template<typename T> class Property {
public:
    Property(std::function<void(const T&)> s, std::function<const T&()> g) :
setter(s), getter(g) {};
    Property<T>& operator= (const T& v) {
        setter(v);
        return *this;
    }
    operator const T& () {
        return getter();
    }
private:
    std::function<void(const T&)> setter;
    std::function<const T&()> getter;
};
```

Usage::outer class

```
class Test {
public:
    const float& getArea() {
        return dimension * dimension;
    }
    void setArea(const float& val) {
        dimension = std::sqrt(val);
    }
    Property<float> area = Property<float>([this](const float&
value){setArea(value);}, [this](){return getArea();});
private:
    float dimension;
};
```

Usage

```
Test test;

test.area = 9;
std::cout << test.area;
```

Efficiency

```
std::cout << sizeof(Property<int>);
```

32 bytes on property + 4 bytes on integer
3 indirection calls

How to improve efficiency?

Involving member pointers

Will it compile?

```
#include <iostream>

struct Test
{
    static void doSomething();
};

void Test::doSomething()
{
}

int main()
{
    typedef void (Test::*abcde)();
    return 0;
}
```

Syntax

```
struct Test {
    void doSomething() {}
};

int main()
{
    typedef void (Test::*Mf)();
    Mf pf = &Test::doSomething;
    Test test;
    (test.*pf)();
    return 0;
}
```

Usage w/o aliases

```
struct Test {
    void doSomething() {}
};

int main()
{
    Test test;
    (test.*(&Test::doSomething))();
    return 0;
}
```

applications

```
#include <string>
#include <iostream>
#include <map>

struct Test {
    void meow() { std::cout << "meow"; }
    void arff() { std::cout << "arff"; }
};

int main()
{
    std::map<std::string, void (Test::*)()> sounds;
    sounds["cat"] = &Test::meow;
    sounds["dog"] = &Test::arff;

    ((Test()).*sounds["cat"])();

    return 0;
}
```

Using in properties

```
template <typename T, typename Host> class Property {
public:
    Property(void (Host::*setter)(const T&),
             const T& (Host::*getter)(), Host* host):
        setter(setter), getter(getter), host(host) {}
    const Property<T, Host>& operator = (const T& value) {
        (host->*setter)(value);
        return *this;
    }
    operator const T& () {
        return (host->*getter)();
    }
private:
    void (Host::*setter)(const T& value);
    const T& (Host::*getter)();
    Host* host;
};
```

Usage

```
struct Test
{
    void set(const int& value) { std::cout << value; };
    const int& get() { return 0; };
};

int main()
{
    Test test;
    Property<int, Test> p(&Test::set, &Test::get, &test);
    p = 5;

    return 0;
}
```

Improved efficiency

```
std::cout << sizeof(Property<int, Test>);
```

5 pointers on property
2 indirection calls

Going deeper

1. “Offsets“ are compile-time constants!
2. Is it possible to move them into compile-time completely?

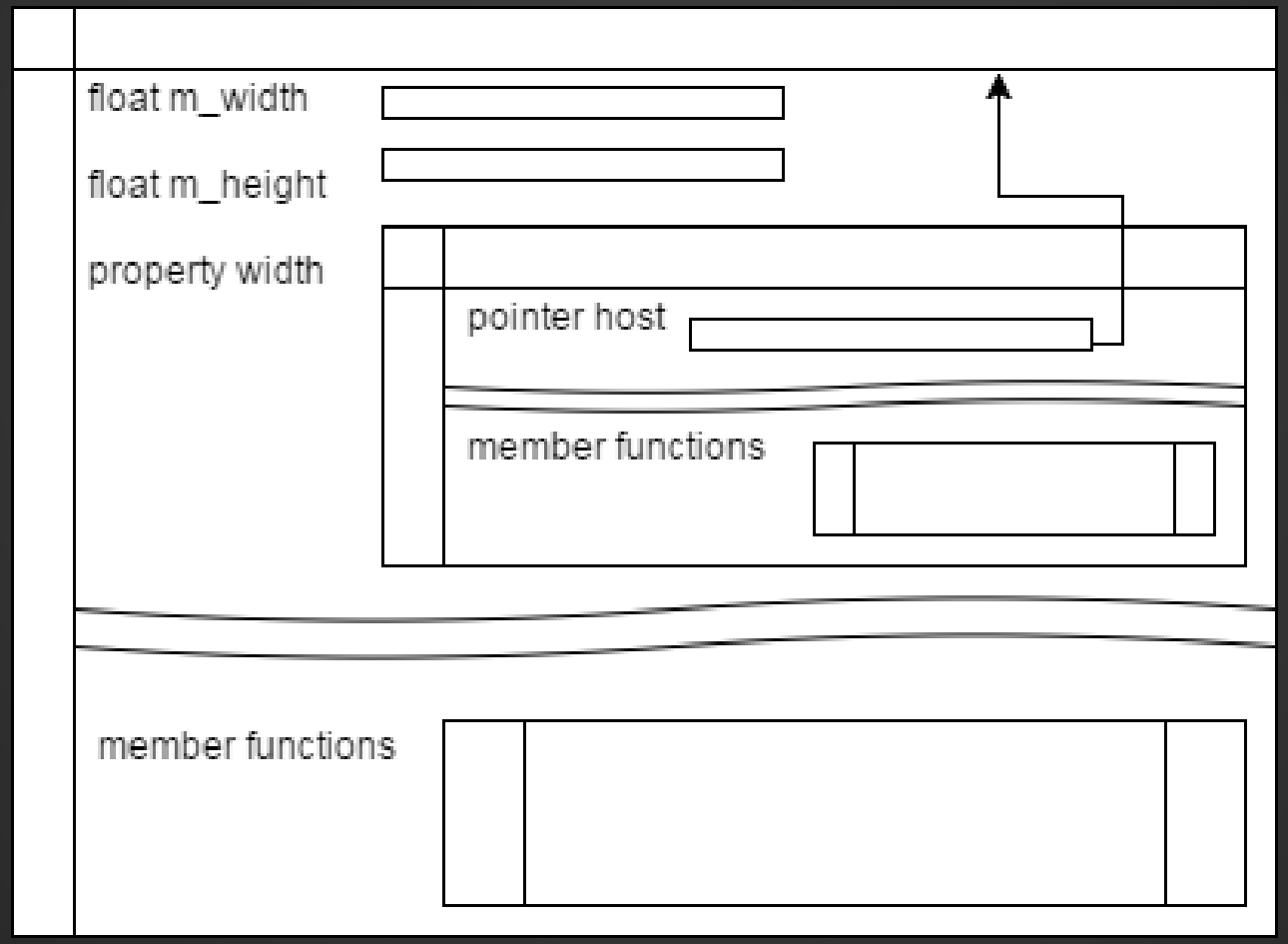
Live demo

Final efficiency

```
std::cout << sizeof(Property<int, Test>);
```

1 pointer on property
1 indirection call

getting rid of memory overhead



We did it!

Thanks to Gašper Ažman!

https://github.com/bitekas/properties/

viktork@zeptolab.com