A Type-safe Generic Pointer

Francis Xavier Pulikotil, 2 Mar 2011 MIT

A safer alternative to void*, **any_ptr** can point to any type of object and provide type-safe, const-correct access to it.

Download source code - 4.09 KB
http://www.codeproject.com/Articles/159454/A-Type-safe-Generic-Pointer

Introduction

Sometimes there comes the need to store a pointer to an object whose type is not known at compile-time. The common method to achieve this in C++ is to store it in a void pointer. The void pointer can then be cast back to the appropriate type and used when required. Call-back functions[1] in libraries are a well-known example of this method; the user-data pointer is often a void pointer. This works, but has a glaring problem: it's not type-safe; it's up to the programmer to interpret the void pointer in the appropriate manner.

Although programmers are quite used to taking care when dealing with void pointers, every once in a while, a void pointer gets accidently cast to the wrong type of object. When this happens, the programmer is actually considered lucky if a crash occurs.

Enter **any_ptr**

The **any_ptr** smart pointer[2] can point to any type of object and provide type-safe access to it. In some sense,**any_ptr** is to pointers what `boost::any`[3] is to objects. There are no requirements to be fulfilled by the types of objects that **any_ptr** can point to. In fact, the performance and size penalty incurred by its use is so low that it could potentially be used almost wherever a void pointer is required.

To start using **any_ptr**, we simply add a single header ***any_ptr**.h* (see source code accompanying this article) to our project. **any_ptr** has no dependencies on any library and can be used independently. **any_ptr** also does not require exceptions[5] or RTTI[6] to be enabled.

Type-safety

any_ptr behaves almost like a void pointer, except for when it is type-cast. A cast to the wrong type of pointer will yield a null pointer which the programmer can check for.

Example

Collapse | Copy Code

```
int a = 10;
any_ptr pAny = &a;

int   *pA = pAny; // 'pA' will point to 'a'; type-compatible
float *pB = pAny; // 'pB' will point to null; type-incompatible
```

Const-correctness

any_ptr also takes care of const-correctness[4]; a cast to an incompatible type with respect to const-correctness will yield a null pointer.

Example 1

Collapse | Copy Code

```
int a = 10;
any_ptr pAny = &a;

const int *p1 = pAny; // 'p1' will point to 'a'; const-compatible
int       *p2 = pAny; // 'p2' will point to 'a'; const-compatible
```

Example 2

Collapse | Copy Code

```
const int a = 10;
any_ptr pAny = &a;

const int *p1 = pAny; // 'p1' will point to 'a'; const-compatible
int       *p2 = pAny; // 'p2' will point to null; const-incompatible
```

Example: passing user-data to call-back functions

Instead of using raw void pointers, **any_ptr** could be used to pass user-defined data into call-back functions. This would provide the author of the call-back function with type-safe access to the user-defined data. Consider the following pseudo-code:

Collapse | Copy Code

```
#include <map>
#include <cassert>

// Call back for key presses
typedef void (*KeyPressedCallback)(void *pUserData);

class KeyPressNotifier
{
private:
    struct RegistrationInfo
    {
        KeyPressedCallback  _pCallback;
        void              *_pUserData;
    };

    typedef std::map<int, RegistrationInfo> InfoMap;
    InfoMap _infoMap;

public:
```

```
    void Register(int key, KeyPressedCallback pCallback, void *pUserData)
    {
        RegistrationInfo &info = _infoMap[key];
        info._pCallback = pCallback;
        info._pUserData = pUserData;
    }

    void UpdateKeyPress(int key)
    {
        InfoMap::const_iterator itr = _infoMap.find( key );
        if( itr != _infoMap.end() )
            itr->second._pCallback( itr->second._pUserData );
    }
};

void OnAccelerate(void *pUserData)
{
    // Expect the speed to be passed in
    float *pSpeed = static_cast<float*>(pUserData);
    assert( pSpeed );

    // Increment speed
    ++(*pSpeed);
}

void OnDrinkPotion(void *pUserData)
{
    // Expect the health to be passed in
    int *pHealth = static_cast<int*>(pUserData);
    assert( pHealth );

    // Increment health
    ++(*pHealth);
}

enum Keys
{
    UpArrowKey,
    EnterKey
};

int main(int /*argc*/, char * /*argv*/[])
{
    KeyPressNotifier notifier;

    float speed = 10.5f;
    notifier.Register( UpArrowKey, OnAccelerate, &speed );

    float health = 100;
```

```
    notifier.Register( EnterKey, OnDrinkPotion, &health );

    notifier.UpdateKeyPress( UpArrowKey );
    notifier.UpdateKeyPress( EnterKey );

    assert( speed == 11.5f );
    assert( health == 101 );
}
```

On first look, the code above seems okay, but there's a bug. In the `OnDrinkPotion` function, the player health is expected to be passed in as an integer. If null is passed in, an assertion failure is triggered. Now although the player health is passed in, its type is `float`. In this case, since the pointer obtained from the void pointer to user-data is invalid but not null, no assertion failure is triggered. In this simple example, the consequences of this bug are not much. But imagine this simple example's real-life counterpart with complex objects being passed in as user-data instead of primitive types.

In this scenario, simply replacing the void pointers with **`any_ptr`**s provides increased type-safety. The following is the same code as above, but with **`any_ptr`**s used instead of void pointers:


```
#include "any_ptr.h"
#include <map>
#include <cassert>

// Call back for key presses
typedef void (*KeyPressedCallback)(any_ptr pUserData);

class KeyPressNotifier
{
private:
    struct RegistrationInfo
    {
        KeyPressedCallback  _pCallback;
        any_ptr             _pUserData;
    };

    typedef std::map<int, RegistrationInfo> InfoMap;
    InfoMap _infoMap;

public:
    void Register(int key, KeyPressedCallback pCallback, any_ptr pUserData)
    {
        RegistrationInfo &info = _infoMap[key];
        info._pCallback = pCallback;
        info._pUserData = pUserData;
    }

    void UpdateKeyPress(int key)
```

```
    {
        InfoMap::const_iterator itr = _infoMap.find( key );
        if( itr != _infoMap.end() )
            itr->second._pCallback( itr->second._pUserData );
    }
};

void OnAccelerate(any_ptr pUserData)
{
    // Expect the speed to be passed in
    float *pSpeed = static_cast<float*>(pUserData);
    assert( pSpeed );

    // Increment speed
    ++(*pSpeed);
}

void OnDrinkPotion(any_ptr pUserData)
{
    // Expect the health to be passed in
    int *pHealth = static_cast<int*>(pUserData);
    assert( pHealth );

    // Increment health
    ++(*pHealth);
}

enum Keys
{
    UpArrowKey,
    EnterKey
};

int main(int /*argc*/, char * /*argv*/[])
{
    KeyPressNotifier notifier;

    float speed = 10.5f;
    notifier.Register( UpArrowKey, OnAccelerate, &speed );

    float health = 100;
    notifier.Register( EnterKey, OnDrinkPotion, &health );

    notifier.UpdateKeyPress( UpArrowKey );
    notifier.UpdateKeyPress( EnterKey );

    assert( speed == 11.5f );
    assert( health == 101 );
```

```
}
```

When the author of the `OnDrinkPotion` function tries to access the player's health through the **`any_ptr`**assuming it to be an integer, the cast yields a null pointer. This causes the author's assertion in the next line to fail, alerting the author that something is wrong.

Using **any_ptr** with existing call-back functions

Unlike the previous example, existing library code can't always be freely modified to make use of **`any_ptr`**s in place of void pointers. In this case, instead of passing a void pointer to the required data, the programmer could pass a void pointer to an **`any_ptr`** which points to the required data. A convention could be followed that user-data void pointers in call-back functions always point to **`any_ptr`**s. Although (for various practical reasons) some exceptions to the convention might be required occasionally, if used properly, it could increase type-safety.

Limitations

- Although this should never really be an issue, an **`any_ptr`** occupies more space than a raw void pointer (its size is a void pointer plus an integer).
- Casting from an **`any_ptr`** is slower than casting from a void pointer. Again, this should never really be an issue.
- Casting to base class pointers from an **`any_ptr`** will not work. However, this is actually a safety feature! Casting from a void pointer to anything other than back to the *exact* same object type is not guaranteed to give you a valid object pointer (especially in the case of multiple inheritance).

Closing

The **`any_ptr`** source code is released under the MIT license and has been tested on the following compilers:

- Microsoft Visual C++ 2005/2008/2010
- GCC 4.4.1 (TDM-2 MinGW32)

There is much potential for improvement; if you make changes to the code, improve it, or have some better ideas, I would love to know. I can be reached by email at francisxavierjp [at] gmail [dot] com. Comments and suggestions are always welcome!

References

1. Callback (Computer Science), from Wikipedia
2. Smart Pointer, from Wikipedia
3. Boost.Any, from Boost library documentation
4. Const correctness, from C++ FAQ Lite
5. Exception handling, from Wikipedia
6. Run-time type information, from Wikipedia

License

Share

-
-
-
-
-
-
-

About the Author

Francis Xavier Pulikotil

Software Developer
United States
Besides loving spending time with family, Francis Xavier likes to watch sci-fi/fantasy/action/drama movies, listen to music, and play video-games. After being exposed to a few video-games, he developed an interest in computer programming. He currently holds a Bachelor's degree in Computer Applications.

Follow on LinkedIn