Instruction for General C++ Reflection Engine

Author Email: jcsfh2015@gmail.com

●Introduction

The main purpose of General C++ Reflection Engine (or gce::reflection) is to provide reflection capabilities to C++ language. As we all know, C++ RTTI has very limited ability to make reflection, which definitely becomes more and more important in modern programming. So it's quite crucial to improve the reflection ability for C++. Fortunately, C++11 and Meta Programming provide very easy ways to explore reflection. So based on C++11 and Meta Programming, this reflection engine implements a serial of general reflection functionalities for C++.

●Features

- examine normal and static fields
- examine methods
- set and get filed values without including the exact definition
- invoke normal, virtual and static methods and get return value dynamically
- examine class inheritance dynamically
- create instance without any header files
- support template class reflection as well
- serialization for complex object

●Compilers and Platforms

linux: g++ version 4.8.3 or higher, 64-bit
windows: Microsoft Visual Studio 2010 or higher

C++ 11 features required for compiler:

- auto and nullptr key word

• lambda expression

• default template type

• right value reference

• template traits

●Builds

• for g++:

gce/trunk/builds/so/make.sh —〉 libgcrt.so libreflect.so

• for Visual Studio

gce/trunk/builds/dll/gcrt/gcrt.vcxproj
gce/trunk/builds/dll/reflect/reflect.vcxproj

●Introduction to gcrt

Before we start to introduce gce::reflection, we need to explain gcrt first, a general C++ runtime library, which is a basic library to gce::reflection. If you don't have any interest, please skip over this part.

Description:

gcrt offers a series of ways to help programmers to troubleshoot bugs and diagnose exceptions or errors of programs, making programs more stable and reliable.

Memory leak detecting

gcrt provides an effective way to detect memory leak and print it out automatically at the end of the running time. But we should be aware that in order to improve performance, this mechanism works only under debug mode.

If there's a leak, the program displays as the following:

```
runtime detected memory leaks ------>
memory leak -> [sequence: 536021  address: 0x000000000263C980  size: 20 bytes]
```

where sequence value points out the exact sequence number of the memory allocation. We can use it to track leak by calling gce::runtime::memory_leak::globle_instance().set_break_alloc(seq number) in the static function pre_loader::_incept() at "src/runtime/impl/pre_loader.cpp" (the number might be inexact under multi-threading environment).

To make memory leak detecting more accurate, gcrt allows programs to release all dynamic memory or resources before memory leak logs are printed. To do so, we just write code as following:

```
    gce::runtime::init_loader::register_cleaner_callback([]()
    {…/*here’ s your code to release memory or resource*/});
```

It’s worth to note that this registration should be called only once for a specific clean, and the anonymous function will be executed after main function exits. For example:

```
#include <runtime/runtime.h>
int* p;
void alloc()
{
    p = new int;
}
int main()
{
    alloc();
    //if we don’ t know exactly where p should be deleted
    //so we release it at the end of the program.
    gce::runtime::init_loader::register_cleaner_callback([]()
    {
        delete p;
    });
    return 0;
}
```

Checking pointer at runtime

One of the depressed things for C++ programmers is to distinguish dangling pointers and check whether the program’s memory is wrecked by inappropriate memory operating. Now gcrt makes it possible to diagnose memory problems.

Example 1:

```
#include <stdio.h>
#include <src/runtime/runtime.h>
//must inherit from gce::Object
class Test : public gce::Object<Test>{…};
int main()
{
    Test* p = new Test;
    delete p;
    try
    {
        //check whether p is dangling
        gce::runtime_check(p);
    }
    catch(std::exception&)
    {
        std::cout<<"dangling pointer found"<< std::endl;
    }
    return 0;
}
```

This program usually (depend on system) displays the following:

```
dangling pointer found
```

Note:

runtime_check is going to fail only if the memory of p is reset by system after deleting operation.

Example 2:

```
#include <stdio.h>
#include <src/runtime/runtime.h>
struct Test
{
    Test(int n)
        : val(n){}
    int val;
};
int main()
{
    char buf[1];
    Test test(1);
    strcpy(buf, "hello");//overflow
    std::cout<<test.val<<std::endl;
    return 0;
}
```

This program usually (depend on system) displays unexpected value because of stack overflow.

Note:

Stack overflow occurs only if the address of test instance is next to that of buf.

That means the layout of stack, which depends on a specific compiler, must be compact. In this document, we assume layouts are all compact.

Now we improve example2 using gcrt:

```
#include <stdio.h>
#include <src/runtime/runtime.h>
struct Test : public gce::Object<Test>
{
    Test(int n)
        : val(n){}
    int val;
};
int main()
{
    char buf[1];
    //must use macro declare to define local instance
    //this macro is a variadic macro, we take 1 as the parameter
    declare(Test, test, 1);
    strcpy(buf, "hello");//overflow
    try
    {
        //check whether instance test is wrecked
        gce::runtime_check(&test);
        std::cout<<test.val<<std::endl; //never perform
    }
    catch(std::exception&)
    {
        std::cout<< "wrecked instance found"<< std::endl;
    }
    return 0;
}
```

This program usually (depend on system) displays the following:

```
wrecked instance found
```

Example 3:

```
#include <stdio.h>
#include <src/runtime/runtime.h>
struct Test//there’ s no need to inherit from gce::Object
{
    Test(int n)
        : val(n){}
    int val;
};
int main()
{
    char buf[1];
    //must use macro declare_ex to define local instance
    //that doesn’ t inherit from gce::Object
    declare_ex(Test, test, 1);
    strcpy(buf, "hello");//overflow
    try
    {
        //check whether instance test is wrecked
        gce::runtime_check(&test);
        std::cout<<test.val<<std::endl; //never perform
    }
    catch(std::exception&)
    {
        std::cout<<"wrecked instance found"<< std::endl;
    }
    return 0;
}
```

This program usually (depend on system) displays the following:

```
wrecked instance found
```

Warning:

Althouth exception caused by gce::runtime_check must indicate that the pointer or instance is invalid, there’s no guarantee that performing gce::runtime_check on a dangling pointer or wrecked instance raises exception. So we should know that even though gcrt is helpful to us for troubleshooting or diagnosing, we could not use it to resolve all memory problems.

●Reflection

It’s quite easy to use gce::reflection.

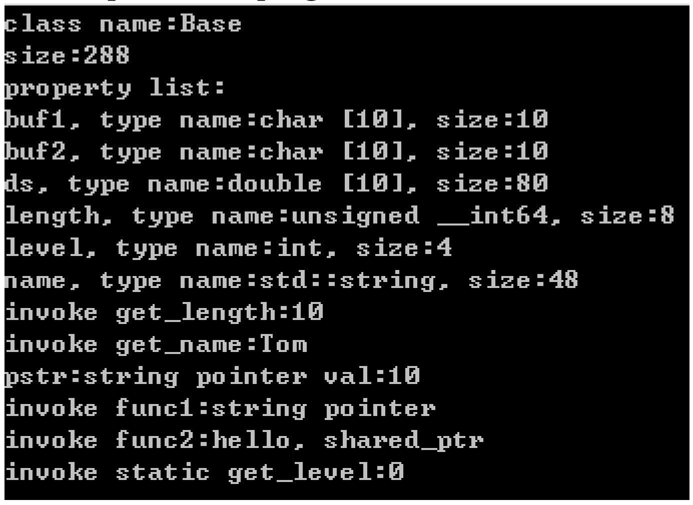
Demo 1:

The following piece of code shows how to define a reflectable class/struct:

```
#include <src/reflect/reflect.h>
#include <stdio.h>
#include <string>
```

```
//must inherit from gce::reflection::reflectable
class Base : public gce::reflection::reflectable<Base>
{
protected:
    //declare reflectable property
    member(unsigned long long, length);
    //declare reflectable property with mutable key word
    member_mutable(std::string, name);
    //declare reflectable property as array, with size 10
    member_array(char, buf1, 10);
    //declare reflectable property as array, with size 10 and mutable key
word
    member_array_mutable(char, buf2, 10);
    //declare reflectable static property
    member_static(int, level);
    //declare reflectable static property as array, with size 10
    member_static_array(double, ds, 10);

public:
    //declare and implement reflectable method
    method(void, set_length, (long long l))
    {
        length = l;
    }

    //declare and implement reflectable method
    method(void, set_name, (const std::string& str))
    {
        name = str;
    }

    //declare and implement reflectable method
    method(unsigned long long, get_length, ())
    {
        return length;
    }

    //declare reflectable method
    method(std::string, get_name, ());

    //declare and implement reflectable static method
    method(static int, get_level, ())
    {
        return level;
    }

    //declare and implement reflectable virtual method
    method(virtual void, do_something, ())
    {
        std::cout<<"Base class do_something invoked"<<std::endl;
    }

    //pointer and reference as parameters or return value
    method(std::string*, func1, (std::string* str, int& val))
    {
        std::cout<<"str:"<<*str<<" val:"<<val<<std::endl;
        return str;
    }

    //std::shared_ptr as return value
    method(std::shared_ptr<std::string>, func2, ())
    {
        return std::shared_ptr<std::string>
            (new std::string("hello, shared_ptr"));
    }
};
//define static property
int Base::level = 0;
double Base::ds[10];

//implement reflectable method out of its class body
std::string Base::get_name()
{
    return name;
}

int main()
{
    Base base;
    //get the class of Base
    auto& base_class = base.get_class();
    //print the class name
    std::cout<<"class name:"<<base_class.get_name()<<std::endl;
    //print the class size
    std::cout<<"size:"<<base_class.get_size()<<std::endl;

    //list all properties
    auto& members = base_class.members();
    std::cout<<"property list:"<<std::endl;
    for(auto it=members.begin(); it!=members.end(); ++it)
    {
        auto& member_class = it->second.get_class();
        std::cout<<it->first<<", type name:"
            <<member_class.get_name()<<", size:"
            <<member_class. get_total_size()
            <<std::endl;
    }

    try
    {
        //invoke method, we need to specify the return type as template
type.
        //10 refers to int implicitly, so this argument must be specified
unsigned long long explicitly
        base_class.get_method("set_length").invoke<void>
            (&base, (unsigned long long)10);
        std::string name = "Tom";
        base_class.get_method("set_name").invoke
            <void, Base, std::string>(&base, name);
        //invoke method and print return value
        std::cout<<"invoke get_length:"
            <<base_class.get_method("get_length").invoke
            <unsigned long long>(&base)<<std::endl;
        std::cout<<"invoke get_name:"
            <<base_class.get_method("get_name").invoke
            <std::string>(&base)<<std::endl;

        std::string str = "string pointer";
        int val = 10;
        char* psz = "char*";
        std::cout<<"invoke func1:"
            <<*base_class.get_method("func1").invoke
            <std::string*>(&base, &str, val, psz)<<std::endl;
        std::cout<<"invoke func2:"
            <<*base_class.get_method("func2").invoke
            <std::shared_ptr<std::string> >(&base)<<std::endl;

        //invoke static method, we don’t need to pass instance,
        //but nullptr instead, which must be interpreted to Base type.
        std::cout<<"invoke static get_level:"
            <<base_class.get_method("get_level").invoke<int>
            ((Base*)nullptr)<<std::endl;
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
    return 0;
}
```

The output of this program would be:



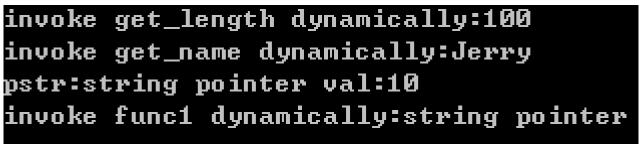
```
class name:Base
size:288
property list:
buf1, type name:char [10], size:10
buf2, type name:char [10], size:10
ds, type name:double [10], size:80
length, type name:unsigned __int64, size:8
level, type name:int, size:4
name, type name:std::string, size:48
invoke get_length:10
invoke get_name:Tom
pstr:string pointer val:10
invoke func1:string pointer
invoke func2:hello, shared_ptr
invoke static get_level:0
```

Demo 2 (dynamic invoke):

If we use template version of invoke as above codes show, we call it static invoke (like invoke<void>(…)), while non-template version called dynamical invoke. Please note that we have the ability to invoke method dynamically, which means all the arguments and return value must be std::string type, which will be converted to real argument type. For example, we pass "100" as string type, and then it will be converted exactly to unsigned long long type by the engine. So we don't need to specify the exact type for each argument or return type as hard code, we just pass string type as parameters and the method will be invoked successfully. By this way, it makes reflection more flexible.

```
int main()
{
    Base base;
    //get the class of Base
    auto& base_class = base.get_class();
    try
    {
        //we don’t have to indicate 100 as unsigned long long,
        //it will be parsed to integer type automatically.
        base_class.get_method("set_length").invoke(&base, "100");
        base_class.get_method("set_name").invoke(&base, "Jerry");
        //invoke method dynamically and print return value, non-template
        std::cout<<"invoke get_length dynamically:"
            <<base_class.get_method("get_length").invoke(&base)
            <<std::endl;
        std::cout<<"invoke get_name dynamically:"
            <<base_class.get_method("get_name").invoke(&base)
            <<std::endl;

        //invoke method dynamically and print return value
        std::cout<<"invoke func1 dynamically:"
            <<base_class.get_method("func1").invoke
            (&base, "string pointer", "10")<<std::endl;
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
    return 0;
}
```

This program prints the following:



```
invoke get_length dynamically:100
invoke get_name dynamically:Jerry
pstr:string pointer val:10
invoke func1 dynamically:string pointer
```

Warning:

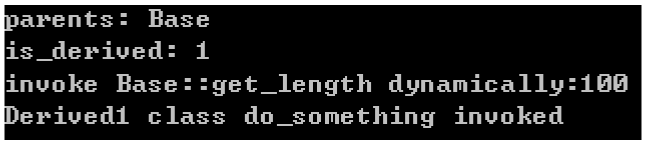
All the arguments and return value must be the type (just built-in type like integer, char or double) that could be converted to std::string and parsed from std::string, because there's no ways to pass argument or return type like std::shared_ptr or self-defined struct, which cannot be parsed to string.

Demo 3 (single inheritance):

```
//specify Base as base class
class Derived1 : public gce::reflection::reflectable<Derived1, Base>
{
public:
    //override virtual method
    method(virtual void, do_something, ())
    {
        std::cout<<"Derived1 class do_something invoked"<<std::endl;
    }
};
```

```
int main()
{
    try
    {
        Derived1 derived1;
        //get the class
        auto& derived1_class = derived1.get_class();

        //print base class
        auto& parents = derived1_class.parents();
        std::cout<<"parents:";
        for(auto it=parents.begin(); it!=parents.end(); ++it)
        {
            std::cout<<" "<<it->first;
        }
        std::cout<<std::endl;

        //check inheritance
        std::cout<<"is_derived: "
            <<gce::reflection::type_manager::is_derived_of
            ("Derived1", "Base")<<std::endl;

        //invoke method of base class
        derived1_class.get_method("set_length").invoke
            (&derived1, "100");
        std::cout<<"invoke Base::get_length dynamically:"
            <<derived1_class.get_method("get_length").invoke
            (&derived1)<<std::endl;

        //invoke virtual method on a base class pointer that points derived
instance
        Base* base = &derived1;
        derived1_class.get_method("do_something").invoke(base);
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
}
```

This program displays the following:

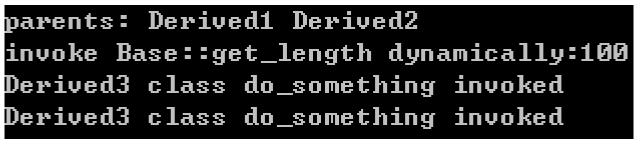


```
parents: Base
is_derived: 1
invoke Base::get_length dynamically:100
Derived1 class do_something invoked
```

Demo 4 (multiple inheritances, i.e diamond hierarchy):

```
//specify Base as base class
class Derived2 : public gce::reflection::reflectable<Derived2, Base>
{
};

//specify Derived1 and Derived2 as base classes
class Derived3 : public gce::reflection::reflectable<Derived3, Derived1,
Derived2>
{
public:
    //override virtual method
    method(virtual void, do_something, ())
    {
        std::cout<<"Derived3 class do_something invoked"<<std::endl;
    }
};

int main()
{
    try
    {
        Derived3 derived3;
        //get the class
        auto& derived3_class = derived3.get_class();

        //print base classes
        auto& parents = derived3_class.parents();
        std::cout<<"parents:";
        for(auto it=parents.begin(); it!=parents.end(); ++it)
        {
            std::cout<<" "<<it->first;
        }
        std::cout<<std::endl;

        //invoke method of base class
        derived3_class.get_method("set_length").invoke
            (&derived3, "100");
        std::cout<<"invoke Base::get_length dynamically:"
            <<derived3_class.get_method("get_length").invoke
            (&derived3)<<std::endl;
        //invoke virtual method on a base class pointer that points derived
instance
        Base* base = &derived3;
        derived3_class.get_method("do_something").invoke(base);

        //invoke by Derived2 instance
        Derived2* derived2 = &derived3;
        derived3_class.get_method("do_something").invoke(derived2);
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
}
```

This program displays the following:



```
parents: Derived1 Derived2
invoke Base::get_length dynamically:100
Derived3 class do_something invoked
Derived3 class do_something invoked
```

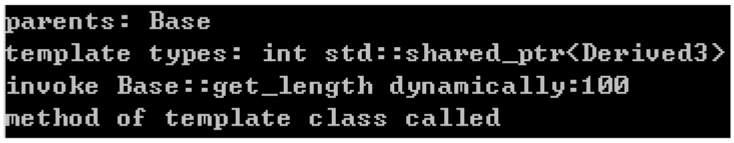
Demo 5 (template class):

For template class, we use gce::reflection::reflectable_template<> as base class, and use method_t& member_t instead of method&member:

```
template<class T1, class T2>
struct templates : public gce::reflection::reflectable_template
    <
    templates<T1, T2>,//Derived class itself
    gce::reflection::template_types<T1, T2>,//placeholer class
                                            //to parse template types
    Base//base class
    >
{
    declare_tempate(templates); //must declare class itself
    method_t(void, func, ())
    {
        std::cout<<"method of template class called"<<std::endl;
    }

    member_t(int, val);
};
```

```
int main()
{
    try
    {
        templates<int, std::shared_ptr<Derived3> > tpl;
        //get the class
        auto& templates_class = tpl.get_class();

        //print base classes
        auto& parents = templates_class.parents();
        std::cout<<"parents:";
        for(auto it=parents.begin(); it!=parents.end(); ++it)
        {
            std::cout<<" "<<it->first;
        }
        std::cout<<std::endl;

        //print template types
        std::cout<<"template types:";
        for(auto it=templates_class.list_sub_type_ptr.begin();
            it!=templates_class.list_sub_type_ptr.end(); ++it)
        {
            std::cout<<" "<<(*it)->get_name();
        }
        std::cout<<std::endl;

        //invoke method of base class
        templates_class.get_method("set_length").invoke(&tpl, "100");
        std::cout<<"invoke Base::get_length dynamically:"
            <<templates_class.get_method("get_length").invoke(&tpl)
            <<std::endl;

        //invoke method
        templates_class.get_method("func").invoke(&tpl);
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
}
```

This program prints out the following:



```
parents: Base
template types: int std::shared_ptr<Derived3>
invoke Base::get_length dynamically:100
method of template class called
```

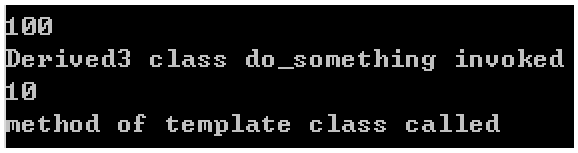
Demo 6 (dynamical instance):

In this doc, we call the instance that is created without any header the dynamical instance, which provides a way to operate instance without any hard code. There are 2 types of dynamical instance.

Type 1, create instance of a class that is defined as hard code in somewhere:

```
int main()
{
    try
    {
        //Before create dynamic instance, we have to register the class
itself.
        //The registration could be called at the start of the program.
        gce::reflection::type_manager::register_type<Derived3>();
        //Derived3 is defined as hard code in somewhere we don't need to
know,
        //so we use get_class to get the class of Derived3
        auto& derived3_class = gce::reflection::type_manager::get_class
            ("Derived3");
        //create instance
        auto instance = derived3_class.create_instance();

        //get/set property value dynamically
        derived3_class.set_value(instance, "length", "100");
        std::cout<<derived3_class.get_value(instance, "length")
            <<std::endl;
        //invoke method dynamically
        derived3_class.invoke(instance, "do_something");

        //must destroy instance
        derived3_class.destroy_instance(instance);

        //for template class
        gce::reflection::type_manager::register_type<
            templates<int, std::shared_ptr<Derived3> > >();
        auto& templates_class =
            gce::reflection::type_manager::get_class
            ("templates<int, std::shared_ptr<Derived3> >");
        auto tempplates_instance =
            templates_class.create_instance();
        templates_class.set_value(tempplates_instance, "length", "10");
        std::cout<<templates_class.get_value
            (tempplates_instance, "length")<<std::endl;
        templates_class.invoke(tempplates_instance, "func");
        templates_class.destroy_instance(tempplates_instance);
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
}
```

Output would be:

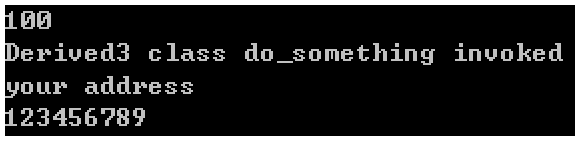


```
100
Derived3 class do_something invoked
10
method of template class called
```

Type 2, the class we are going to create is not defined before; it's totally "created dynamically":

```
int main()
{
    try
    {
        //your_class is never defined before,
        //so we use create_class to create the class of it.
        //Derived3 is base class!
        //test is namespace!
        auto& your_class = gce::reflection::type_manager::create_class
            ("test::your_class", "Derived3");
        //we can add member to your_class!
        your_class.add_member("addr", "std::string");
        your_class.add_member("phone", "unsigned long");
        //now create instance
        auto instance = your_class.create_instance();

        //get/set base class's property value dynamically
        your_class.set_value(instance, "length", "100");
        std::cout<<your_class.get_value(instance, "length")
            <<std::endl;
        //invoke base class's method dynamically
        your_class.invoke(instance, "do_something");
        //get/set property value dynamically added!
        your_class.set_value(instance, "addr", "your address");
        std::cout<<your_class.get_value(instance, "addr")<<std::endl;
        your_class.set_value(instance, "phone", "123456789");
        std::cout<<your_class.get_value(instance, "phone")<<std::endl;

        //must destroy instance
        your_class.destroy_instance(instance);
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
}
```

Output would be:



```
100
Derived3 class do_something invoked
your address
123456789
```

Warning:

- add_member must be called before an instance created.

- gce::reflection allows us to create a class with more than one base class.

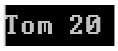
But it' s not a good idea to do so. Because there' s no way to implement virtual inheritance dynamically, which means if more than one of the base classes have a common father class, it' s going to be more than one father class instance for a single derived instance, which could cause ambiguities on method invoking or getting/setting property of the common father class.

Demo 7 (serialization):

A simple example:

```
struct Contact : public gce::reflection::reflectable<Contact>
{
    member(unsigned int, age);
    member(std::string, name);
};

int main()
{
    try
    {
        Contact contact;
        //set value, must use ref_ method to indicate this field need to
be serialized.
        contact.ref_age() = 20;
        contact.ref_name() = "Tom";

        //serialize
        std::string out;
        contact.serialize(out);

        //deserialize on a new instance
        Contact new_contact;
        new_contact.deserialize(out);

        std::cout<<new_contact.name<<" "<<new_contact.age<<std::endl;
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
}
```

Result is:



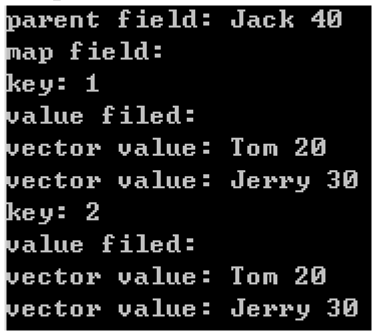
```
Tom 20
```

A more complicated example:

```
//inherited from Contact
struct Buddy : public gce::reflection::reflectable<Buddy, Contact>
{
    //must use COMMA other than ','
    member(std::map<int COMMA std::shared_ptr<std::vector<Contact> > >
    , list);
};

int main()
{
    try
    {
        Buddy buddy;
        //set base class's property value,
        //must use ref_ method to indicate this field need to be
        serialized.
        buddy.ref_age() = 40;
        buddy.ref_name() = "Jack";

        //initialize a contact instance
        Contact contact;
        contact.ref_name() = "Tom";
        contact.ref_age() = 20;

        auto vec1 = std::shared_ptr<std::vector<Contact> >
            (new std::vector<Contact>);
        //add a value
        vec1->push_back(std::move(contact));
        //add another value
        //ref_ method is supposed to be used only once on a same instance!
        contact.name = "Jerry";
        contact.age = 30;
        vec1->push_back(std::move(contact));

        //must use ref method
        auto& list = buddy.ref_list();
        //add a pair values
        list.insert(std::make_pair(1, vec1));

        //add another value
        auto vec2 = std::shared_ptr<std::vector<Contact> >
            (new std::vector<Contact>);
        contact.name = "Tom";
        contact.age = 20;
        vec2->push_back(std::move(contact));
        //add another value
        contact.name = "Jerry";
        contact.age = 30;
        vec2->push_back(std::move(contact));
        list.insert(std::make_pair(2, vec2));

        std::string out;
        buddy.serialize(out);

        Buddy new_buddy;
        new_buddy.deserialize(out);

        //print parent field
        std::cout<<"parent field: "<<new_buddy.name
            <<" "<<new_buddy.age<<std::endl;
        //print map field
```

```
        std::cout<<"map field: "<<std::endl;
        for(auto it_map=new_buddy.list.begin();
            it_map!=new_buddy.list.end(); ++it_map)
        {
            std::cout<<"key: "<<it_map->first<<std::endl;
            std::cout<<"value filed: "<<std::endl;
            for(auto it_vec=it_map->second->begin();
                it_vec!=it_map->second->end(); ++it_vec)
            {
                std::cout<<"vector value: "<<(*it_vec).name
                    <<" "<<(*it_vec).age<<std::endl;
            }
        }
    }
    catch(std::exception& e)
    {
        std::cout<<e.what()<<std::endl;
    }
}
```

Output:



```
parent field: Jack 40
map field:
key: 1
value filed:
vector value: Tom 20
vector value: Jerry 30
key: 2
value filed:
vector value: Tom 20
vector value: Jerry 30
```

Use your own serializer:

We are allowed call set_serializer to use our own serializer that implements gce::reflection::serializer_interface before we start to serialize.

Other Notes:

• When we create class dynamically, all class names must be the names with full namespace(s).

• All STL containers are able to be reflected using gce::reflection. But std::wstring is not supported by this version.

- Only the containers that have the capability to be iterated can be serialized and deserialized.
- try-catch is required.