C++ Metaprogramming

Weak dynamic typing and garbage collection with C++

Passau, Germany

Aless Lasaruk

FORWISS, University of Passau

September 7, 2010

Motivation

- Type systems in symbolic languages, e.g. Maple

```
a := 1
a := Matrix([[1,2],[3,4]])
a := plot(...) // What happens with the matrix?
```

- The identifier “a” seems to be “typeless”
- ... Not really

```
a + 1 // Some kind of error or escape message
type(a) // Someone seems to know it
```

- Is this possible in C++?

```
MagicType a = new int(1);
MagicType a = new QImage("lena.jpg");
cast<QImage>(a)->width(); // 512
*cast<int>(a)+*cast<int>(a); // Kind of runtime error
std::cout << a.type_name()
          << std::endl;  // Results in "QImage"
```

Metaprogramming and C++

- Metaprogramming is a tool to implement what we want to ...
- **Conventional:** Metaprogramming with C++ means “programming with templates”
- **More general:** Automatic code generation
- Here

Metaprogramming

Automatic code modifying mechanisms “beyound” normal programmers work

- **Tools:** Templates, macros, static initialization, code generators, etc.
- Metaprogamming is the future of sceintific computing!

Why weak typing?

- **Plugin**-pattern requires unified interface, e.g.

```
WeakType* plugin_call(std::list<WeakType*>& arguments)
```

- “WeakType” must reflect **all types** used with the plugin!
- **Conventional:** Base type for all objects (everything is a ”WeakType”)

```
WeakType* a =
    new SubType(...) // "SubType" extends "WeakType"
```

Problem

What to do with ”QImage” (Qt) or ”CvPoint” (OpenCV)?

- The answer is: “generalization” to

Weak typing

Syntactical constructs (identifiers, etc.) do not care static (e.g. compile-time) type information.

Dynamic typing

- **Question:** Where is the type information?

Dynamic typing

Type information is stored within the objects and is available at runtime.

- **RTTI:** Type info class in C++ (introduced with *dynamic_cast*)

```
const type_info& type_id(T); // T is either a typename
    or an instance
```

- Type info even available for pure C data types at compile time
- **Conventional:** *dynamic_cast* (Objects **must** belong to the same hierarchy)

```
SubType* b =
  // Hurts badly if "a" is a "QImage"
  dynamic_cast<SubType*>((WeakType*)a);
```

Problem

Sounds like "base type for everything" is necessary ...

On the way to DPtr

- Suppose the common base type is “WeakType”
- We can reimplement “CvPoint” as a class (**too much** code)
- Wrap “CvPoint” in a special “Data”-Object
- **Data object**

```
class Data : public WeakType {
  public:
    // Constructor from any type
    Data(void* data, TypeInfo* type)
      : data(data), type(type) { ... }
    ...
  protected:
    void* data;
    TypeInfo* type;
};
```

- **Experience:** External types are used more frequently (in ContRap)

DPtr and SPtr

- Why to subclass "WeakType" at all? ...
- **Everything** is wrapped in a "smart"-pointer class DPtr

```
class DPtr {
  ...
  protected:
    DPtr(void* data, TypeInfo* type) : ... { ... }
    void* data;
    TypeInfo* type;
};
```

- Type-templated **identical** "DPtr"-subclass "SPtr"

```
template<typename T> class SPtr : public DPtr {
  public:
    SPtr(T* data) : DPtr(data, static_type) { ... }
    ...
  protected:
    static TypeInfo* static_type;
};
```

- **Idea:** Use "DPtr" for weak typed interfaces, "SPtr" as a pointer replacement

SPtr in detail

- Each “SPtr”-instance implicitly knows its type info (even at compile time)

```
template<typename T> class SPtr : public DPtr {
  public:
    SPtr(T* data):DPtr(data, get_static_type()) { ... }
    // On-first-use static info generator
    static TypeInfo* get_static_type() {
        if (!static_type) static_type = TYPE_INFO(T);
        return static_type;
    }
    // Dereferencing operator
    T& operator*() {return *reinterpret_cast<T*>(data);}
    ...
  protected:
    static TypeInfo* static_type;
};
```

- **Homework:** Why not to instantiate “static_info” statically?

Weak typing with DPtr and SPtr

- **Idea:** Use “SPtr<T>” but return a ”DPtr”

```
DPtr plugin_call(std::list<DPtr>& arguments) {
  ...
  return SPtr<String>(new String("Blah"));
}
```

- Syntactical sugar for not typing the type name twice:

```
#define SPTR(type,...) \
  (SPtr<type>(new type(__VA_ARGS__)))
```

- **Notice:** SPTR is a variadic macro.
- Example

```
SPtr<String> s = SPTR(String,"Blah");
std::string a = *s + *s; // SPtr has an operator*
size_t b = s->find("blup"); // SPtr has an operator->
DPtr ws = s; // The type is not lost
```

Type casts

Big difference between "void*" and "DPtr"

A "DPtr"-instance is weak typed but knows its type! (**weak dynamic typing**)

- Cast from "SPtr" to "DPtr" is easy, what about the other direction?

```
template <typename T> SPtr<T> cast(const DPtr& ptr);
```

- If the static meta info of "T" equals to the meta info in "ptr", we are done!
- **Otherwise:** We **must** know if "T" and the "ptr"-content are in the same hierarchy of base class "B"!
- If this "hierarchy element"-problem is solved

```
template <class T, class B>
  SPtr<T> cast(const DPtr& ptr) {
  if (ptr.in_hierarchy(T,B))
    return SPtr<T>(dynamic_cast<T*>((B*)ptr.get_ptr()));
  return 0;
}
```

The hierarchy element problem

... is a **hard one**!

- **Solution:** The "meta-registration"-concept
- **Idea:** Singleton class "ObjectFactory" knows the base types of objects

```
#define REGISTER_OBJECT(name,base) \
  // Meta object cast function for the special type
  bool _cast_ ## name(void* ptr) \
    { dynamic_cast<name*>((base*) ptr); } \
  // Static variable is initialized on library loading
  static bool _registred_ ## name =
    ObjectFactory::instance()->register_object(
      _cast_ ## name,TYPE_INFO(name),TYPE_INFO(base));
```

- **Disadvantage:** For each type you want to use

```
REGISTER_OBJECT(QImage,QPaintDevice)
```

ObjectFactory in detail

- “ObjectFactory” hashes the object meta information

```
class ObjectFactory {
...
public:
  // Returns true if ptr can be casted to info
  bool can_cast(DPtr ptr, TypeInfo* info) {
    std::map<TypeInfo*,MetaInfo>::iterator it = objects.find(info);
    if (it != objects.end() &&
      // Equal base means the same type hierarchy
      ptr.get_base_info() == (*it).second.base_info)
        // Dynamic cast always legal
        return (*it).second.cast(ptr);
  }
};
```

- The “hierarchy element” problem is solved by a call to *can_cast*!

Working with SPtr and DPtr

- An error throwing version of cast easily available

```
template <class T> SPtr<T> hard_cast(const DPtr& ptr){
  SPtr<T> c = cast<T>(ptr);
  if (!c)
    throw(...);
  return c;
}
```

- Back to the introduction example

```
DPtr a = SPTR(String,"blah");
SPtr<String> b = cast<String>(a);
std::string c =
   *hard_cast<String>(a)+*hard_cast<String>(b);
a = SPTR(String,"blup"); // What happens to blah
```

Why reference counting garbage collection?

Why garbage collection?

- Initialized pointers which run out of scope due to exception

```
SPtr<String> a = new String("blup");
if (...) throw (...)
// What happens with a?
```

- Return values of functions are created with *new*-operator

```
CONTRAP_FUNCTION Integer* integer_add(Integer* a,
    Integer* b) {
 return new Integer(*a+*b);
}
```

- The intermediate results must be destroyed (e.g. $2 \cdot (3 + 4)$)

Why reference counting garbage collection?

Why reference counting?

- If you want to reuse the memory ...

```
CONTRAP_FUNCTION SPtr<Integer> integer_add(Integer* a, Integer* b) {
  static SPtr<Integer> result = 0;
  if (!result && result.get_rc() > 1)
    result = new Integer(*a+*b);
  else
    *result = *a+*b;
  return result;
}
```

- High benefit for large objects

Non-intrusive reference counting

- Use “RefPtr”-object instead of void pointer in DPtr

```
class RefPtr {
  public:
    ...
  protected:
    void* data;
    long rc;
    DPtr source;
};
```

```
class DPtr {
  public:
    DPtr(void* data,
         TypeInfo* type) :
         ... { ... }
    ...
  protected:
    RefPtr* data;
    TypeInfo* type;
};
```

- Increase the reference counter when DPtr is copied, decrease on delete, e.g.

```
    DPtr::DPtr(DPtr& p) :
       data(p.data), type(p.type) { rc++; }
```

Application of smart pointers

- **Avoid:** Create two “DPtr”-s, which point to the same memory
- Returning pointers to aggregated parts leads to crashes

```
SPtr<char> get_line(SPtr<Image> image, int* y) {
  // Takes a pointer to the part of the image
  return &image->data[y*image->width];
}
```

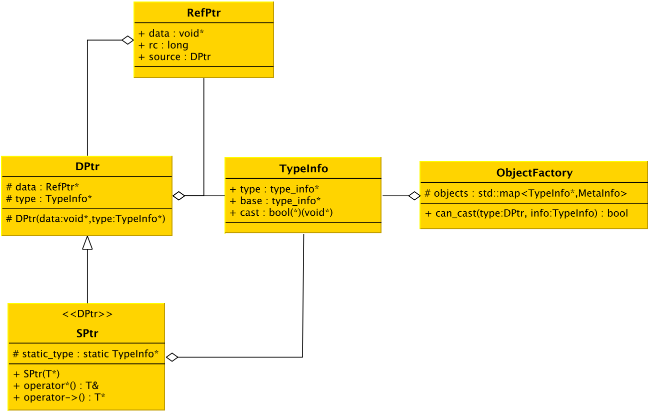
- For object-dependent pointers use the “SPtr”-constructor

```
SPtr::SPtr(T* t, DPtr source) : DPtr(T, get_static_type
    ()) { this->source = source; }
```

- Correct usage

```
SPtr<char> get_line(SPtr<Image> image, int* y) {
  // "image" is not destroyed before return value
  return SPtr(&image->data[y*image->width],image);
}
```

Class hierarchy



- More detailed information about the ContRap implementation available at

  http://contrap.sourceforge.net/html/classcrp_1_1DPtr.html

Summary

- "DPtr" for unified weak typed interfaces
- "SPtr" is a transparent replacement for regular pointers
- Casts are possible in both directions
- REGISTER_OBJECT macro for casted classes
- Pointers are reference counting (no delete necessary)
- A **valid** "DPtr" is a one with non-zero data
- "RefPtr" and "DPtr" are easily extended with timestamps
- Ordering on "DPtr"-s (**is_new**-replacement)

Thank you for the patience!