Praxisorientierte Einführung in C++
Lektion: "Runtime-Type-Identification (RTTI) in C++"

Christof Elbrechter

Neuroinformatics Group, CITEC

June 18, 2014

Table of Contents

- RTTI
- typeid
- Beispiel
- Anmerkungen
- RTTI: Do-it-yourself ...
- RTTI und Polymorphismus

RTTI

- C++ ist eine statisch-Typisierte Sprache
- Typen stehen immer zur Übersetzungs-Zeit fest
- Daher: Eigentlich keine Laufzeit-Typinformationen notwendig
- Machmal wären Laufzeit-TypInformationen aber doch sehr nützlich
  - Um z.B. tatsächlichen Typ eines polymorphen Objektes (z.B. Implementation eines Interfaces) zur Laufzeit herauszufinden
- Header <typeinfo> stellt Klasse `std::type_info` zur Verfügung
- `std::type_info` ist sehr stark in die Sprache eingebaut

Der 'typeid'-Operator

- Es gibt sogar einen extra Operator namens `typeid(Objekt oder Typname)`
- `typeid` erzeugt eine `std::type_info`-Instanz, welche den Typ von `Objekt` bzw. `Typname` beschreibt
- `std::type_info` enthalten dennoch nur wenige verlässliche Informationen
  - `std::type_info` Instanzen können verglichen werden
  - Es gibt eine Methode `name()` welche einen dynamisch generierten Typnamen zurückgibt

Wichtig

- `typeid(...).name()` erzeugt eine **Kompiler-abhängige** String-Repräsentation eines Typs.
- Nur für Typ-Vergleich innerhalb eines Programmes

Beispiel

- Man kann z.B. für bessere Handhabbarkeit ein nettes Template damit implementieren

```
#include <typeinfo>
#include <string>

template<class T>
inline std::string get_type_name(const T &t=T()){
   return typeid(t).name();
}
```

- Template kann ebenfalls mit Objekt-Instanz und mit Typnamen aufgerufen werden

```
std::string intName = get_type_name<int>();
float f = 7.3;
std::string floatName = get_type_name(f);
```

Anmerkungen

- Da `typeid` ein C++-Operator ist, werden solche Ausdrücke falls möglich zur Übersetzungszeit ausgewertet und damit erheblich beschleunigt

```
if(typeid(myVar)==typeid(int)){
  // ...
}
```

- Für polymorphe Typen[1] wird allerdings die *vtable* ausgewertet, was erheblich länger dauert
- Damit aber erst wirkliche Laufzeit-Typ-Identifikation

[1] implementiert oder beerbt mindestens eine virtuelle Funktion

RTTI: Do-it-yourself ...

- In Zeitkritischen Fällen kann auf einfache Weise eine Laufzeit-Typ-Identifikation selbst implementiert werden
- **Nachteil:** Mehr Schreibarbeit, insbes. bei Hinzufügen von neuen Typen
- **Vorteil:** Schneller und geht auch für nicht-polymorphe Typen

Event.h

```
#include <iostream>

struct Event{
  enum Type{
    MouseEvent, KeyEvent, UpdateEvent, unknownEvent//, ...
  };
  Event(Type type):type(type){}
  Type getType() const { return type; }

  protected:
  Type type;
};
```

Fortsetzung ...

test_events.h

```
#include <Event.h>
struct MouseEvent : public Event{
  MouseEvent():Event(Event::MouseEvent){}
};
struct KeyEvent : public Event{
  KeyEvent():Event(Event::KeyEvent){}
};
struct UpdateEvent : public Event{
  UpdateEvent():Event(Event::UpdateEvent){}
};

void handle(const Event &e){
  switch(e.getType()){
    case Event::MouseEvent:
      handle_mouse_event(reinterpret_cast<const MouseEvent&>(e));
      break;
      // ...
  }
}
int main(){
  handle(MouseEvent());
  handle(KeyEvent());
}
```

Typ-Identifikation via String

- Verwende `std::string` anstatt `enum`-Wert
- Langsamer
- Dynamischer → es muss kein `enum` erweitert werden, wenn ein neuer Typ hinzugefügt wird

Weitere Speziellere Techniken

- Explizite Kodierung eines Vererbungs-graphen
- Damit kann man z.B. sowas wie `is_a` implementieren
- Dann allerdings fraglich, ob schneller als C++-RTTI

RTTI und Polymorphismus

- Wie geht `typeid` mit polymorphen Typen um?
- Wird immer der speziellste oder der allgemeinste Typ zurückgegeben?

Experiment

```
#include <typeinfo>
#include <string>
#include <iostream>

template<class T> inline std::string get_type_name(const T &t=T()) {  return typeid(t).name(); }

#define SHOW(X) std::cout << #X << ":" << X << std::endl;

struct Base {};
struct Derived : public Base {};

int main(){
  Base base;
  Derived derived;
  Base &derived2 = derived;

  SHOW(get_type_name(base));
  SHOW(get_type_name(derived));
  SHOW(get_type_name(derived2));
}
```

Output

Output des Programms

```
get_type_name(base):4Base
get_type_name(derived):7Derived
get_type_name(derived2):4Base
```

- Bei derived2 wird nicht der tatsächliche Typ zurückgegeben, sondern der Compile-Time Typ der Referenz

Bei polymorphen Typen ...

- Aber was passiert, wenn wir polymorphe Typen haben

Anpassung des Beispiels

```
...
struct Base {
  virtual ~Base() {} // Base ist nun polymorph
};
struct Derived : public Base {}; // Derived dadurch automatisch auch
...
```

Output des angepassten Programms

```
get_type_name(base):4Base
get_type_name(derived):7Derived
get_type_name(derived2):7Derived
```

- Nun funktioniert `typeinfo` wie erwartet