Observable

Geração de observáveis através da metaprogramação

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*
  - Modo arcaíco: classes base *Observable* e *Observer*

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*
  - Modo arcaíco: classes base *Observable* e *Observer*
    - Poliformismo dinâmico

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*
  - Modo arcaíco: classes base *Observable* e *Observer*
    - Poliformismo dinâmico
    - Notificação abrangente (não sei exatamente o que mudou)

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*
  - Modo arcaíco: classes base *Observable* e *Observer*
    - Poliformismo dinâmico
    - Notificação abrangente (não sei exatamente o que mudou)
  - Modo moderno: sinais e slots

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*
  - Modo arcaíco: classes base *Observable* e *Observer*
    - Poliformismo dinâmico
    - Notificação abrangente (não sei exatamente o que mudou)
  - Modo moderno: sinais e slots
    - Poliformismo estático

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*
  - Modo arcaíco: classes base *Observable* e *Observer*
    - Poliformismo dinâmico
    - Notificação abrangente (não sei exatamente o que mudou)
  - Modo moderno: sinais e slots
    - Poliformismo estático
    - Não é tão intrusivo

Problema

Observar a alteração de estado de objetos C++

- *Observer pattern (GoF)*
  - Modo arcaíco: classes base *Observable* e *Observer*
    - Poliformismo dinâmico
    - Notificação abrangente (não sei exatamente o que mudou)
  - Modo moderno: sinais e slots
    - Poliformismo estático
    - Não é tão intrusivo
    - Notificações específicas (sinais customizados)

Observed

Classe que é observada

```
using skills_t = std::map<
  std::size_t, /*skill level*/
  std::string  /*skill name*/
>;

struct person_t {
  std::string name;
  std::size_t age;
  skills_t skills;
};
```

Boost.Signals2

```
struct person_t {
   std::string name() const noexcept
   { return _name; }

   void name(std::string val)
   {
      _name = val;
      _name_changed(_name);
   }
private:
   std::string _name;
   boost::signals2::signal<void(std::string)>
      _name_changed;
};
```

Boost.Signals2

```
struct person_t {
  std::string name() const noexcept
  { return _name; }

  void name(std::string val)
  {
    _name = val;
    _name_changed(_name);
  }

  std::size_t age() const noexcept
  { return _age; }

  void age(std::size_t val)
  {
    _age = val;
    _age_changed(_age);
  }
private:
  std::string _name;
  std::size_t _age;

  boost::signals2::signal<void(std::string)> _name_changed;
  boost::signals2::signal<void(std::size_t)> _age_changed;
};
```

Boost.Signals2

```
struct person_t {
  std::string name() const noexcept
  { return _name; }

  void name(std::string val)
  {
    _name = val;
    _name_changed(_name);
    _any_changed();
  }

  std::size_t age() const noexcept
  { return _age; }

  void age(std::size_t val)
  {
    _age = val;
    _age_changed(_age);
    _any_changed();
  }
private:
  std::string _name;
  std::size_t _age;

  boost::signals2::signal<void(std::string)> _name_changed;
  boost::signals2::signal<void(std::size_t)> _age_changed;
  boost::signals2::signal<void()> _any_changed;
};
```

Boost.Signals2

```
struct person_t {
  //... setters e getters para name e age ...
  skills_t::iterator insertSkill(std::size_t level, std::string name)
  {
   auto it = _skills.emplace(level, name).first;
   _skill_inserted(it);
   return it;
  }
  //erase, find, size e etc
private:
  std::string _name;
  std::size_t _age;
  skills_t _skills;
  boost::signals2::signal<void(std::string)> _name_changed;
  boost::signals2::signal<void(std::size_t)> _age_changed;
  boost::signals2::signal<void()> _any_changed;
  boost::signals2::signal<void(skills_t::iterator)> _skill_inserted;
};
```

Boost.Signals2

```
struct person_t {
  //... setters e getters para name e age ...
  skills_t::iterator insertSkill(std::size_t level, std::string name)
  {
   auto it = _skills.emplace(level, name).first;
   _skill_inserted(it);
   return it;
  }
  //erase, find, size e etc
private:
  std::string _name;
  std::size_t _age;
  skills_t _skills;
  boost::signals2::signal<void(std::string)> _name_changed;
  boost::signals2::signal<void(std::size_t)> _age_changed;
  boost::signals2::signal<void()> _any_changed;
  boost::signals2::signal<void(skills_t::iterator)> _skill_inserted;
};
```

- API proprietária para acessar o container

Boost.Signals2

```
struct person_t {
  //... setters e getters para name e age ...
  skills_t::iterator insertSkill(std::size_t level, std::string name)
  {
   auto it = _skills.emplace(level, name).first;
   _skill_inserted(it);
   return it;
  }
  //erase, find, size e etc
private:
  std::string _name;
  std::size_t _age;
  skills_t _skills;
  boost::signals2::signal<void(std::string)> _name_changed;
  boost::signals2::signal<void(std::size_t)> _age_changed;
  boost::signals2::signal<void()> _any_changed;
  boost::signals2::signal<void(skills_t::iterator)> _skill_inserted;
};
```

- API proprietária para acessar o container
- Existem 6 sobrecargas para unordered_map::insert() em C++11!

Boost.Signals2

```
struct person_t {
  //... setters e getters para name e age ...
  skills_t::iterator insertSkill(std::size_t level, std::string name)
  {
   auto it = _skills.emplace(level, name).first;
   _skill_inserted(it);
   return it;
  }
  //erase, find, size e etc
private:
  std::string _name;
  std::size_t _age;
  skills_t _skills;
  boost::signals2::signal<void(std::string)> _name_changed;
  boost::signals2::signal<void(std::size_t)> _age_changed;
  boost::signals2::signal<void()> _any_changed;
  boost::signals2::signal<void(skills_t::iterator)>_skill_inserted;
};
```

- API proprietária para acessar o container
- Existem 6 sobrecargas para unordered_map::insert() em C++11!
- Alteração no elemento notifica o dono do container? (person_t)

Boost.Signals2

```
struct person_t {
  //... setters e getters para name e age ...
  skills_t::iterator insertSkill(std::size_t level, std::string name)
  {
   auto it = _skills.emplace(level, name).first;
   _skill_inserted(it);
   return it;
  }
  //erase, find, size e etc
private:
  std::string _name;
  std::size_t _age;
  skills_t _skills;
  boost::signals2::signal<void(std::string)> _name_changed;
  boost::signals2::signal<void(std::size_t)> _age_changed;
  boost::signals2::signal<void()> _any_changed;
  boost::signals2::signal<void(skills_t::iterator)>_skill_inserted;
};
```

- API proprietária para acessar o container
- Existem 6 sobrecargas para unordered_map::insert() em C++11!
- Alteração no elemento notifica o dono do container? (person_t)
- Elementos observáveis?

Observable

Contrói *Observables* em tempos de compilação usando sinais e slots

- Elimina código *boilerplate* para a construção do *Observable*

Observable

Contrói *Observables* em tempos de compilação usando sinais e slots

- Elimina código *boilerplate* para a construção do *Observable*
  - Agilidade na implementação

Observable

Contrói *Observables* em tempos de compilação usando sinais e slots

- Elimina código *boilerplate* para a construção do *Observable*
  - Agilidade na implementação
  - Evita erros de uma solução *handwritten*

Observable

Contrói *Observables* em tempos de compilação usando sinais e slots

- Elimina código *boilerplate* para a construção do *Observable*
  - Agilidade na implementação
  - Evita erros de uma solução *handwritten*
- Não intrusivo. *Observed* está separado do *Observable*

Observable

Contrói *Observables* em tempos de compilação usando sinais e slots

- Elimina código *boilerplate* para a construção do *Observable*
  - Agilidade na implementação
  - Evita erros de uma solução *handwritten*
- Não intrusivo. *Observed* está separado do *Observable*
- Suporte a containers da STL

Observable - person_t

```
OBSERVABLE_CLASS_GEN(
    observable_person,
    person_t,
    ((std::string, name))
    ((std::size_t, age))
    ((skills_t, skills))
)
```

Observable - person_t

```
OBSERVABLE_CLASS_GEN(
  observable_person,
  person_t,
  ((std::string, name))
  ((std::size_t, age))
  ((skills_t, skills))
)

inline observable_person
observable_factory(person_t& observed) {
  return observable_person(observed,
                           observed.name,
                           observed.age,
                           observed.skills);
}
```

Observable - person_t

```
person_t observed{"maria", 26};
```

Observable - person_t

```
person_t observed{"maria", 26};

auto operson = observable_factory(observed);
```

Observable - person_t

```
person_t observed{"maria", 26};

auto operson = observable_factory(observed);

operson.on_change<name>(
    [](std::string name)
    {
      std::cout << "name_has_changed_to_" << name;
    });
```

Observable - person_t

```
person_t observed{"maria", 26};

auto operson = observable_factory(observed);

operson.on_change<name>(
    [](std::string name)
    {
      std::cout << "name_has_changed_to_" << name;
    });

operson.get<name>() = "MARIA";
```

Observable - person_t

```
auto& oskills = operson.get<skills>();
```

Observable - person_t

```
auto& oskills = operson.get<skills>();

oskills.on_insert(
    [](const skills_t&,
       skills_t::const_iterator it)
    {
      std::cout << "new_skill_was_added:_"
                << it->second << "_with_level_"
                << it->first;
    });
```

Observable - person_t

```
auto& oskills = operson.get<skills>();

oskills.on_insert(
    [](const skills_t&,
       skills_t::const_iterator it)
    {
      std::cout << "new_skill_was_added:_"
                << it->second << "_with_level_"
                << it->first;
    });

oskills.emplace(8, "woodworking");
```

Tipos de Observables

- Containers da STL
  - map, unordered_map, unordered_set e vector

Tipos de Observables

- Containers da STL
  - map, unordered_map, unordered_set e vector
  - Os elementos são *observables*

Tipos de Observables

- Containers da STL
  - map, unordered_map, unordered_set e vector
  - Os elementos são *observables*
  - Não guardam *observables*

Tipos de Observables

- Containers da STL
  - map, unordered_map, unordered_set e vector
  - Os elementos são *observables*
  - Não guardam *observables*
- Classes
  - Os membros são *observables*

Tipos de Observables

- Containers da STL
  - map, unordered_map, unordered_set e vector
  - Os elementos são *observables*
  - Não guardam *observables*
- Classes
  - Os membros são *observables*
- Variant
  - boost::Variant

Tipos de Observables

- Containers da STL
  - map, unordered_map, unordered_set e vector
  - Os elementos são *observables*
  - Não guardam *observables*
- Classes
  - Os membros são *observables*
- Variant
  - boost::Variant
  - Visitor visita *observable* do elemento

Tipos de Observables

- Containers da STL
  - map, unordered_map, unordered_set e vector
  - Os elementos são *observables*
  - Não guardam *observables*
- Classes
  - Os membros são *observables*
- Variant
  - boost::Variant
  - Visitor visita *observable* do elemento
- Valor
  - Qualquer objeto que modela o concept *Assignable*

Sinais

- on_change
  - Alteração em qualquer parte do objeto
  - Disponível em qualquer *observable*

Sinais

- on_change
  - Alteração em qualquer parte do objeto
  - Disponível em qualquer *observable*
- on_insert e on_erase
  - Inserção e remoção de elementos de containers

Sinais

- on_change
  - Alteração em qualquer parte do objeto
  - Disponível em qualquer *observable*
- on_insert e on_erase
  - Inserção e remoção de elementos de containers
- on_value_change
  - Alteração no estado de elementos de containers e variants

Ownership dos observables

- Classes
  - *Ownership* do programador

Ownership dos observables

- Classes
  - *Ownership* do programador
- Membros de classes
  - *Ownership* da classe

Ownership dos observables

- Classes
  - *Ownership* do programador
- Membros de classes
  - *Ownership* da classe
- Elementos de containers e variant
  - *Ownership* compartilhado

Observed público ou privado

Livre escolha em disponibilizar ou não alterações diretas em *Observed*

- Exemplo para esconder:

```
struct operson_t : observable_person {
  operson_t(person_t p)
    : observable_person(observable_factory(_observed))
    , _observed(std::move(p))
  {}
private:
  person_t _observed;
};
```

Operações básicas

- Obter *Observed* a partir de *Observable*
  - const Observed& observable.get()

Operações básicas

- Obter *Observed* a partir de *Observable*
  - const Observed& observable.get()
- Obter *Observable* de um membro de classe observável
  - auto& omember = observable.get<member>()

Operações básicas

- Obter *Observed* a partir de *Observable*
  - const Observed& observable.get()
- Obter *Observable* de um membro de classe observável
  - auto& omember = observable.get<member>()
    - Retorna uma referência

Operações básicas

- Obter *Observed* a partir de *Observable*
  - const Observed& observable.get()
- Obter *Observable* de um membro de classe observável
  - auto& omember = observable.get<member>()
    - Retorna uma referência
- Obter *Observable* do elemento de um container ou variant
  - auto omember = ocontainer.at(key)

Operações básicas

- Obter *Observed* a partir de *Observable*
  - const Observed& observable.get()
- Obter *Observable* de um membro de classe observável
  - auto& omember = observable.get<member>()
    - Retorna uma referência
- Obter *Observable* do elemento de um container ou variant
  - auto omember = ocontainer.at(key)
    - Retorna um std::shared_ptr para o observável do elemento

Operações básicas

- Obter *Observed* a partir de *Observable*
  - const Observed& observable.get()
- Obter *Observable* de um membro de classe observável
  - auto& omember = observable.get<member>()
    - Retorna uma referência
- Obter *Observable* do elemento de um container ou variant
  - auto omember = ocontainer.at(key)
    - Retorna um std::shared_ptr para o observável do elemento
- Obter tipo de um *Observable*
  - using OFoo = observable::observable_of_t<Foo>

Operações básicas

- Obter *Observed* a partir de *Observable*
  - const Observed& observable.get()
- Obter *Observable* de um membro de classe observável
  - auto& omember = observable.get<member>()
    - Retorna uma referência
- Obter *Observable* do elemento de um container ou variant
  - auto omember = ocontainer.at(key)
    - Retorna um std::shared_ptr para o observável do elemento
- Obter tipo de um *Observable*
  - using OFoo = observable::observable_of_t<Foo>
    - Útil para passar *Observables* para funções ou definir visitors de observáveis