An application for private inheritance?

Lightning Talk for MUC++

Matthäus Brandl

2018-05-17

Wait, what?

What is private inheritance

```
1 class Derived : private Base
2 {};
```

- all public and protected members of `Base` are accessible as private members of the derived class
- private members of the base are never accessible unless friended
- instances of `Derived` cannot be cast to `Base` outside of `Derived` as this relationship is inaccessible

Wait, what?

What is private inheritance

```
1 class Derived : private Base
2 {};
```

- all public and protected members of `Base` are accessible as private members of the derived class
- private members of the base are never accessible unless friended
- instances of `Derived` cannot be cast to `Base` outside of `Derived` as this relationship is inaccessible

Consequently this models HAS-A instead of IS-A.
But HAS-A is better modelled by using a member variable because this causes less coupling (*favor composition over inheritance*).

Exceptions to the rule

Private inheritance should be used if one does not want to model IS-A but

Exceptions to the rule

Private inheritance should be used if one does not want to model IS-A but

- needs to override a virtual function

Exceptions to the rule

Private inheritance should be used if one does not want to model IS-A but

- needs to override a virtual function
- needs access to a protected member

Exceptions to the rule

Private inheritance should be used if one does not want to model IS-A but

- needs to override a virtual function
- needs access to a protected member
- needs to create an object before another base or destroy it after another base

Exceptions to the rule

Private inheritance should be used if one does not want to model IS-A but

- needs to override a virtual function
- needs access to a protected member
- needs to create an object before another base or destroy it after another base
- needs to share a virtual base or needs to control the construction of a virtual base

Exceptions to the rule

Private inheritance should be used if one does not want to model IS-A but

- needs to override a virtual function
- needs access to a protected member
- needs to create an object before another base or destroy it after another base
- needs to share a virtual base or needs to control the construction of a virtual base
- wants to make use of the Empty Base Optimization, e.g. with policy-based design

Exceptions to the rule

Private inheritance should be used if one does not want to model IS-A but

- needs to override a virtual function
- needs access to a protected member
- needs to create an object before another base or destroy it after another base
- needs to share a virtual base or needs to control the construction of a virtual base
- wants to make use of the Empty Base Optimization, e.g. with policy-based design

Also see the C++ FAQ or cppreference.com.

The problem

Suppose the following C API:

```
1  typedef struct
2  {
3      /* raw owning pointer, it's C after all */
4      char const * name;
5
6      /* more variables that need resources ... */
7  } Widget;
8
9  int createWidget(Widget const ** widget);
10
11 void freeWidget(Widget const * widget);
```

Your job now is to implement this API using C++.

The C-ish approach

Tedious but effective

```
1  int createWidget(Widget const ** const widget)
2  {
3    auto const newWidget = (Widget *) std::malloc(sizeof(
         Widget));
4    if (!newWidget)
5    {
6      return 0;
7    }
8    *widget = Widget{}; // zero initialize
9
10   std::string const name = getNameFromSomewhere(/* ... */);
11   if (name.empty())
12   {
13     freeWidget(newWidget);
14     return 0;
15   }
16   newWidget->name = strdup(name.c_str());
17
18   /* ... */
19
20   *widget = newWidget;
21   return 1;
22 }
```

The C-ish approach

Tedious but effective

```
1  int createWidget(Widget const ** const widget)
2  {
3    auto const newWidget = (Widget *) std::malloc(sizeof(
         Widget));
4    if (!newWidget)
5    {
6      return 0;
7    }
8    *widget = Widget{}; // zero initialize
9
10   std::string const name = getNameFromSomewhere(/* ... */);
11   if (name.empty())
12   {
13     freeWidget(newWidget);
14     return 0;
15   }
16   newWidget->name = strdup(name.c_str());
17
18   /* ... */
19
20   *widget = newWidget;
21   return 1;
22 }
```

```
1  void freeWidget(Widget const * const widget)
2  {
3    std::free(widget->name);
4
5    /* ... */
6
7    std::free(widget);
8  }
```

The C-ish approach

Advantages:

- straightforward to implement
- easy to understand
- low complexity

The C-ish approach

Advantages:

- straightforward to implement
- easy to understand
- low complexity

Disadvantages:

- manual ressource management
- error prone
- hard to get right
- tedious

Automating ressource management

Using the power of C++

Automating ressource management

Using the power of C++

```
1  class WidgetWrapper : public Widget
2  {
3  public:
4    explicit WidgetWrapper()
5          : Widget() // Default initialization!
6    {}
7
8    void setName(std::string value)
9    {
10     m_name = std::move(value);
11     name = m_name.c_str();
12   }
13
14   // More setters...
15
16 private:
17   std::string m_name;
18 };
```

Automating ressource management

Using the power of C++

```
class WidgetWrapper : public Widget
{
public:
  explicit WidgetWrapper()
        : Widget() // Default initialization!
  {}

  void setName(std::string value)
  {
    m_name = std::move(value);
    name = m_name.c_str();
  }

  // More setters...

private:
  std::string m_name;
};
```

```
int createWidget(Widget const ** const widget)
try
{
  auto newWidget = std::make_unique<WidgetWrapper>();

  std::string name = getNameFromSomewhere(/* ... */);
  if (name.empty())
  {
    return 0;
  }
  newWidget->setName(std::move(name));

  // More setters

  *widget = newWidget.release();
  return 1;
}
catch (std::bad_alloc const &)
{
  return 0;
}
```

Automating ressource management

Using the power of C++

```
class WidgetWrapper : public Widget
{
public:
  explicit WidgetWrapper()
        : Widget() // Default initialization!
  {}

  void setName(std::string value)
  {
    m_name = std::move(value);
    name = m_name.c_str();
  }

  // More setters...

private:
  std::string m_name;
};
```

```
int createWidget(Widget const ** const widget)
try
{
  auto newWidget = std::make_unique<WidgetWrapper>();

  std::string name = getNameFromSomewhere(/* ... */);
  if (name.empty())
  {
    return 0;
  }
  newWidget->setName(std::move(name));

  // More setters

  *widget = newWidget.release();
  return 1;
}
catch (std::bad_alloc const &)
{
  return 0;
}
```

```
void freeWidget(Widget const * const widget)
{
  delete static_cast<WidgetWrapper const *>(widget);
}
```

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation
- easier `freeWidget()` implementation
- encapsulated "conversion" to C types

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation
- easier `freeWidget()` implementation
- encapsulated "conversion" to C types

Disadvantages:

Potential for resource leaks:

- `static_cast` can be forgotten during deletion
- implementers can still access `Widget` members and use them wrongly (e.g. assign raw owning pointers)

Enter private inheritance

Making `Wrapper` members inaccessible

Enter private inheritance

Making `Wrapper` members inaccessible

```
1  class WidgetWrapper : private Widget // private!
2  {
3    // As before
4
5  public:
6    Widget const * toWidget() const
7    {
8      return static_cast<Widget const *>(this);
9    }
10
11   static void deleteWidget(Widget const * const widget)
12   {
13     delete static_cast<WidgetWrapper const *>(widget);
14   }
15 };
```

Enter private inheritance

Making `Wrapper` members inaccessible

```
1  class WidgetWrapper : private Widget // private!
2  {
3    // As before
4
5  public:
6    Widget const * toWidget() const
7    {
8      return static_cast<Widget const *>(this);
9    }
10
11   static void deleteWidget(Widget const * const widget)
12   {
13     delete static_cast<WidgetWrapper const *>(widget);
14   }
15 };
```

```
1  int createWidget(Widget const ** const widget)
2  try
3  {
4    // As before
5
6    *widget = newWidget->toWidget();
7    newWidget.release();
8    return 1;
9  }
10 catch /* as before */
```

```
1  void freeWidget(Widget const * const widget)
2  {
3    WidgetWrapper::deleteWidget(widget);
4  }
```

Enter private inheritance

Advantages:

- `Widget` members not public anymore in `WidgetWrapper` context
- Easy to use right, hard to use wrong

Enter private inheritance

Advantages:

- `Widget` members not public anymore in `WidgetWrapper` context
- Easy to use right, hard to use wrong

Disadvantages:

- still possible to delete a `WidgetWrapper` via a pointer to `Widget`
  (but easier to remember the function than the `static_cast`)
- increased complexity, two additional functions necessary
- uses private inheritance for an IS-A relationship

Alternatives?

- Do not introduce private inheritance and trust in that no one will use the `Widget` wrongly

Alternatives?

- Do not introduce private inheritance and trust in that no one will use the `Widget` wrongly
- Use aggregation and pass the pointer to the member to the client
  But now shared state between `createWidget()` and `freeWidget()` is necessary to find the correct `WidgetWrapper` instance for the given `Widget` pointer

Alternatives?

- Do not introduce private inheritance and trust in that no one will use the `Widget` wrongly
- Use aggregation and pass the pointer to the member to the client
  But now shared state between `createWidget()` and `freeWidget()` is necessary to find the correct `WidgetWrapper` instance for the given `Widget` pointer

Please share your opinion and ideas

There is a working example on Coliru