An application for private inheritance?

Lightning Talk for MUC++

Matthäus Brandl

2018-05-17

Wait, what?

What is private inheritance

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

- all public and protected members of `Base` accessible as private members of `Derived`
- private members of `Base` never accessible (unless friended)

Wait, what?

What is private inheritance

```cpp
class Derived : private Base
{};
```

- all public and protected members of `Base` accessible as private members of `Derived`
- private members of `Base` never accessible (unless friended)
- inheritance relationship not accessible outside of `Derived`, not `static_cast`-able

Wait, what?

What is private inheritance

```cpp
class Derived : private Base
{};
```

- all public and protected members of `Base` accessible as private members of `Derived`
- private members of `Base` never accessible (unless friended)
- inheritance relationship not accessible outside of `Derived`, not `static_cast`-able
- models HAS-A instead of IS-A

Wait, what?

What is private inheritance

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

- all public and protected members of `Base` accessible as private members of `Derived`
- private members of `Base` never accessible (unless friended)
- inheritance relationship not accessible outside of `Derived`, not `static_cast`-able
- models HAS-A instead of IS-A

However HAS-A is usually better modelled by using a member variable because this causes less coupling (*favor composition over inheritance*).

Typical use cases for private inheritance

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

- needs to override a virtual function

Typical use cases for private inheritance

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

Typical use cases for private inheritance

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
- wants to make use of the Empty Base Optimization (e.g. with policy-based design)

Typical use cases for private inheritance

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
- wants to make use of the Empty Base Optimization (e.g. with policy-based design)

There are also other use cases, see the C++ FAQ or cppreference.com.

The problem

Suppose a dynamic library with the following C API:

```c
typedef struct
{
    /* pointer to a resource, e.g., a C string */
    char const * foo;

    /* more variables that need resources ... */

} Widget;

int createWidget(Widget const ** widget);

void freeWidget(Widget const * widget);
```

We want to implement this API using C++...

The C-ish approach

Tedious but effective

```cpp
int createWidget(Widget const ** widget)
{
  Widget * newWidget =
      (Widget *) std::malloc(sizeof(Widget));
  if (!newWidget)
  {
    return OUT_OF_MEMORY;
  }
  *widget = {}; // zero initialize

  std::string foo = frobnicate(/* ... */);
  if (foo.empty())
  {
    freeWidget(newWidget);
    return FROBNICATE_FAILED;
  }
  newWidget->foo = strdup(foo.c_str());

  /* ... */

  *widget = newWidget;
  return SUCCESS;
}
```

```cpp
void freeWidget(Widget const * widget)
{
  std::free(widget->foo);

  /* ... */

  std::free(widget);
}
```

The C-ish approach

malloc() & free() galore

Advantages:

The C-ish approach

`malloc()` & `free()` galore

Advantages:

- straightforward to implement
- easy to understand
- low complexity

The C-ish approach

`malloc()` & `free()` galore

Advantages:

- straightforward to implement
- easy to understand
- low complexity

Disadvantages:

The C-ish approach

`malloc()` & `free()` galore

Advantages:

- straightforward to implement
- easy to understand
- low complexity

Disadvantages:

- manual ressource management
- raw owning pointers
- error prone
- hard to get right
- tedious

Automating ressource management

Using the power of C++

Idea:

- introduce class `ResourcedWidget`

Automating ressource management

Using the power of C++

Idea:

- introduce class `ResourcedWidget`

```
class ResourcedWidget
{
};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class `ResourcedWidget`
- derive from C struct `Widget`

```cpp
class ResourcedWidget
{
```

```cpp
};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class `ResourcedWidget`
- derive from C struct `Widget`

```
class ResourcedWidget : public Widget
{
};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class `ResourcedWidget`
- derive from C struct `Widget`
- default initialize the base C struct

```
class ResourcedWidget : public Widget
{
};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class `ResourcedWidget`
- derive from C struct `Widget`
- default initialize the base C struct

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



};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class ResourcedWidget
- derive from C struct Widget
- default initialize the base C struct
- for every resource in Widget
  - add managing member (std::string, std::unique_ptr, std::vector, ...) to ResourcedWidget

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

};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class ResourcedWidget
- derive from C struct Widget
- default initialize the base C struct
- for every resource in Widget
    - add managing member (std::string, std::unique_ptr, std::vector, ...) to ResourcedWidget

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

```

```
private:
  std::string m_foo;
};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class ResourcedWidget
- derive from C struct Widget
- default initialize the base C struct
- for every resource in Widget
  - add managing member (std::string, std::unique_ptr, std::vector, ...) to ResourcedWidget
  - add setter to assign resource to member and assign correct pointer to C struct

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

```

```cpp
private:
  std::string m_foo;
};
```

Automating ressource management

Using the power of C++

Idea:

- introduce class ResourcedWidget
- derive from C struct Widget
- default initialize the base C struct
- for every resource in Widget
  - add managing member (std::string, std::unique_ptr, std::vector, ...) to ResourcedWidget
  - add setter to assign resource to member and assign correct pointer to C struct

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

  void setFoo(std::string const & value)
  {
    m_foo = value;
    foo = m_foo.c_str();
  }

  // more setters...

private:
  std::string m_foo;
};
```

Automating ressource management

Using the power of C++

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

```

Automating ressource management

Using the power of C++

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

    std::string foo = frobnicate(/* ... */);
    if (foo.empty())
    {
      return FROBNICATE_FAILED;
    }
    newWidget->setFoo(foo);

    // more setters...

```

Automating ressource management

Using the power of C++

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

    std::string foo = frobnicate(/* ... */);
    if (foo.empty())
    {
      return FROBNICATE_FAILED;
    }
    newWidget->setFoo(foo);

    // more setters...

    *widget = newWidget.release();
    return SUCCESS;
  }
```

Automating ressource management

Using the power of C++

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

    std::string foo = frobnicate(/* ... */);
    if (foo.empty())
    {
      return FROBNICATE_FAILED;
    }
    newWidget->setFoo(foo);

    // more setters...

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

Automating ressource management

Using the power of C++

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

    std::string foo = frobnicate(/* ... */);
    if (foo.empty())
    {
      return FROBNICATE_FAILED;
    }
    newWidget->setFoo(foo);

    // more setters...

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

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

Automating ressource management

Advantages:

Easier usage:

Automating ressource management

Advantages:

Easier usage:

- automated resource management

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation
- easier `freeWidget()` implementation

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation
- easier `freeWidget()` implementation
- feels like a C++ class

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation
- easier `freeWidget()` implementation
- feels like a C++ class

Disadvantages:

Potential for resource leaks:

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation
- easier `freeWidget()` implementation
- feels like a C++ class

Disadvantages:

Potential for resource leaks:

- `static_cast` can be forgotten during deletion

Automating ressource management

Advantages:

Easier usage:

- automated resource management
- `Widget` members are default initialized
- easier `createWidget()` implementation
- easier `freeWidget()` implementation
- feels like a C++ class

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 `Widget` members inaccessible

Enter private inheritance

Making Widget members inaccessible

```
class ResourcedWidget : private Widget // private!
{
```

Enter private inheritance

Making Widget members inaccessible

```
class ResourcedWidget : private Widget // private!
{
  // as before...

```

Enter private inheritance

Making Widget members inaccessible

```
class ResourcedWidget : private Widget // private!
{
  // as before...

public:
  Widget const * toWidget() const
  {
    return static_cast<Widget const *>(this);
  }

```

Enter private inheritance

Making Widget members inaccessible

```
class ResourcedWidget : private Widget // private!
{
  // as before...

public:
  Widget const * toWidget() const
  {
    return static_cast<Widget const *>(this);
  }

  static void deleteWidget(Widget const * widget)
  {
    delete static_cast<ResourcedWidget const *>(widget);
  }
};
```

Enter private inheritance

Making Widget members inaccessible

```cpp
class ResourcedWidget : private Widget // private!
{
  // as before...

public:
  Widget const * toWidget() const
  {
    return static_cast<Widget const *>(this);
  }

  static void deleteWidget(Widget const * widget)
  {
    delete static_cast<ResourcedWidget const *>(widget);
  }
};
```

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

  // as before...

  *widget = newWidget->toWidget();
  newWidget.release();
  return SUCCESS;
}
catch /* as before */
```

Enter private inheritance

Making Widget members inaccessible

```cpp
class ResourcedWidget : private Widget // private!
{
  // as before...

public:
  Widget const * toWidget() const
  {
    return static_cast<Widget const *>(this);
  }

  static void deleteWidget(Widget const * widget)
  {
    delete static_cast<ResourcedWidget const *>(widget);
  }
};
```

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

  // as before...

  *widget = newWidget->toWidget();
  newWidget.release();
  return SUCCESS;
}
catch /* as before */
```

```cpp
void freeWidget(Widget const * widget)
{
  ResourcedWidget::deleteWidget(widget);
}
```

Enter private inheritance

Advantages:

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

Enter private inheritance

Advantages:

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

Disadvantages:

- still possible to delete a `ResourcedWidget` 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?

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 `ResourcedWidget` 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 `ResourcedWidget` instance for the given `Widget` pointer
- Leave the type system with `reinterpret_cast`

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 `ResourcedWidget` instance for the given `Widget` pointer
- Leave the type system with `reinterpret_cast`
- Use a handle approach, to access a data member you pass the handle to a dedicated function

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 `ResourcedWidget` instance for the given `Widget` pointer
- Leave the type system with `reinterpret_cast`
- Use a handle approach, to access a data member you pass the handle to a dedicated function

Please share your opinion and ideas (brandl.matthaeus@gmail.com)

There is a working example on Coliru