Introduction to the C++ Programming Language

Day 4

Jonas Rylund Glesaaen
jonas@glesaaen.com

October 1st 2015

What will we learn?

- ~~Basic C++ syntax~~
- ~~Control structures~~
- ~~Functions~~
- Structs and classes (Wednesday and today)
- Templates and STL (today and Friday)
- Exceptions (Friday)

Today's topics

1. Inheritance
2. Polymorphism
3. Templates
4. Programming Practices
5. Recap

Inheritance

Subtypes and supertypes

Inheritance lets you create new classes based on existing ones

Subtypes/children are more specialised than their supertypes/parents

Children inherit all the variables and methods of their parents

Simple Example

```
class Animal
{
public:
  void eat();
};

class Bird : public Animal
{
public:
  void fly();
};
```

Every animal can eat

Simple Example

```
class Animal
{
public:
  void eat();
};

class Bird : public Animal
{
public:
  void fly();
};
```

Every bird is an animal

→ Every bird can eat

Every bird can fly {well...}

Simple Example

```
class Animal
{
public:
  void eat();
};

class Bird : public Animal
{
public:
  void fly();
};
```

Not every animal is a bird
→ Not all animals can fly

Simple Example

```
#include"animals.hpp"

int main()
{
  Animal cow;
  cow.eat();

  Bird penguin;
  penguin.eat();
  penguin.fly();
}
```

Simple Example

```
#include"animals.hpp"

int main()
{
  Animal cow;
  cow.eat();
  cow.fly();                    Error: cows can't fly

  Bird penguin;
  penguin.eat();
  penguin.fly();
}
```

Inheritance and access

My { children } are not my friends

(they might be bastards)

Inheritance and access

My { siblings } are not my friends

(they are definitely up to no good)

Inheritance and access

My { parents } are not my friends

(they weren't chosen by me)

Inheritance and access

Inherited classes do not have access to the base class' private members

Here is where the protected access level enters

protected members are

- private to the outside
- public for subclasses

Inheritance and access

There is also an access level at the inheritance point

```
class Bird : public Animal {}
```

This one

Inheritance and access

This is the access level the inherited members will have in the new subclass

```
class Animal
{
public:
  void eat();
};

class Bird : public Animal {}
```

`eat()` **is a public method in** `Bird`

Inheritance and access

This is the access level the inherited members will have in the new subclass

```
class Animal
{
public:
  void eat();
};

class Bird : private Animal {}
```

`eat()` **is a private method in** `Bird`

Inheritance and access

Think of public inheritance as

{child} is a {parent}

Think of private inheritance as

{child} is implemented in terms of {parent}

Don't think too hard about protected inheritance

Constructors

Constructors are not inherited {not even the copy constructor}

But a parent's constructor can (and will) be called

Constructors

```
class Derived : public Base
{
public:
  Derived() {}

  Derived(int x)
    : var {x} {}

  Derived(int x)
    : Base {x} {}
};
```

Constructors

```
class Derived : public Base
{
public:
  Derived() {}

  Derived(int x)
    : var {x} {}

  Derived(int x)
    : Base {x} {}
};
```

Base default constructor called

Base default constructor called

Base constructor accepting an int called

Constructors

The copy constructor can be defined as

```
class Derived : public Base
{
public:
  Derived() {}

  Derived(const Derived & copy)
    : Base {copy} {}
};
```

We will get back to why this works

Destructors

The destructor is also not inherited

But it will also be called when the instance is deleted
{more on this later}

Inheritance vs composition

Inheritance is **great**,

but remember that all of it so far can also be accomplished with composition

Inheritance vs composition

```
class Bird
{
public:
  void eat()
  {
    base.eat();
  }

private:
  Animal base;
};
```

Inheritance vs composition

Always consider your options

Composition is used to describe

{A} is implemented in terms of {B}

{A} has a {B}

Friendship under inheritance

Friends of my { parents } are not my friends

(they might be creeps)

Friendship under inheritance

Friends of my { children } are not my friends

(they are all annoying)

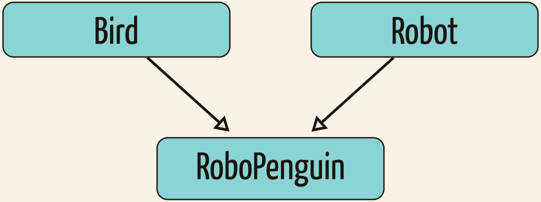
Friendship under inheritance

Friends of my { friends } are not my friends

(they are probably criminals)

Multiple inheritance

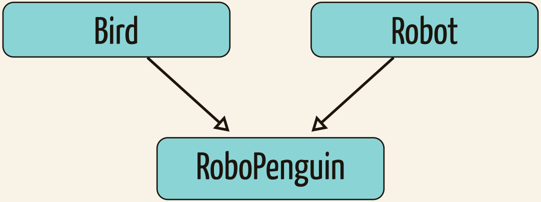
It is also possible to inherit from multiple classes




```
class RoboPenguin : public Bird, public Robot {};
```

Multiple inheritance

It is also possible to inherit from multiple classes



Note, the two base classes shouldn't declare the same members

Polymorphism

polymorphism -noun

/ˌpɒlɪˈmɔːfɪz(ə)m/

From ancient Greek

πολύς (polús), meaning "many" or "much", and

μορφή (morphé), meaning "form" or "shape"

a) **the condition of occurring in several different forms**

b) **the ability to assume different forms or shapes**

What is polymorphism?

A base class can always be replaced with
a child class instance

If given by-value:
  the derived class will be cast to the base class

If given by-reference:
  the derived class will keep its identity

Polymorphism - example

Base class

```
class Logger
{
public:
  virtual void log(std::string) = 0;
};
```

Polymorphism - example

Derived #1

```cpp
class ConsoleLogger : public Logger
{
public:
  virtual void log(std::string message) override
  {
    std::cout << message << std::endl;
  }
};
```

Derived #2

```cpp
class FileLogger : public Logger
{
private:
  std::ofstream ofs;

public:
  FileLogger(std::string filename)
    : ofs {filename} {}

  virtual void log(std::string message) override
  {
    ofs << message << std::endl;
  }
};
```

Polymorphism - example

Implementation

```
void complicatedOperation(/* args */, Logger & logger)
{
  // ...
  logger.log("Stuff happens");
}

int main()
{
  ConsoleLogger console_log;
  complicatedOperation(console_log);

  FileLogger file_log {"program.log"};
  complicatedOperation(file_log);
}
```

The virtual keyword

The virtual keyword means that the function in question can be overridden

If not present the base class' implementation will be called even when we have a by-reference instance

The virtual keyword

```
class Base
{
public:
  std::string name() const
  {
    return {"Base"};
  }
};
```

```
class Derived : public Base
{
public:
  std::string name() const
  {
    return {"Derived"};
  }
};
```

```
int main()
{
  Derived dclass;
  Base & bref = dclass;

  dclass.name();   "Derived"

  bref.name();     "Base"
}
```

The virtual keyword

```
class Base
{
public:
  virtual std::string name() const
  {
    return {"Base"};
  }
};
```

```
class Derived : public Base
{
public:
  virtual std::string name() const
  {
    return {"Derived"};
  }
};
```

```
int main()
{
  Derived dclass;
  Base & bref = dclass;

  dclass.name();
  bref.name();
}
```

dclass.name(); → "Derived"

bref.name(); → "Derived"

Overriding functions

Using polymorphism and overriding functions is a great way of writing more generalised code and decrease code duplication

Humans are awesome at abstraction, take advantage of that for your programming

The override keyword

virtual

tell the base class that a derived class can override the function

↕ Symmetry

override

tell the derived class that the function overrides a function from the base class

{C++11}

The override keyword

```
class Base
{
public:
  virtual std::string name() const
  {
    return {"Base"};
  }
};

class Derived : public Base
{
public:
  virtual std::string name()
  {
    return {"Derived"};
  }
};
```


{C++11}

The override keyword

```
class Base
{
public:
  virtual std::string name() const
  {
    return {"Base"};
  }
};

class Derived : public Base
{
public:
  virtual std::string name()
  {
    return {"Derived"};
  }
};
```

This will not override

{C++11}

The override keyword

```
class Base
{
public:
  virtual std::string name() const
  {
    return {"Base"};
  }
};

class Derived : public Base
{
public:
  virtual std::string name() override
  {
    return {"Derived"};
  }
};
```

Compile error

{C++11}

The override keyword

```
class Base
{
public:
  virtual std::string name() const
  {
    return {"Base"};
  }
};

class Derived : public Base
{
public:
  virtual std::string name() const override
  {
    return {"Derived"};
  }
};
```

OK

{C++11}

The final keyword

- The final keyword is used to disable inheritance
- It can be used on both classes and member functions

```
class Base
{
  virtual void finalMethod() final;
};

class Derived final : public Base
{
  virtual void finalMethod();
};

class Child : public Derived {};
```

Compile error

{C++11}

Pure virtual functions

It is possible to create an abstract class, which has virtual functions with no implementation

Pure virtual functions

It is possible to create an abstract class, which has virtual functions with no implementation

```
class Shape
{
public:
  virtual double area() const = 0;
};
```

Every shape has an area, but the area of a generic shape is undefinable

Interfaces

The sum of methods a class has is called its interface

It is common to define the topmost class of a hierarchy as a pure interface, or a pure abstract class

Destructors again

When making use of polymorphism it is important to mark all destructors as virtual

This way a derived class can be destructed from a base reference

```
class Base
{
public:
  ~Base() {...}
};

class Derived : public Base {...};

std::unique_ptr<Base> base_ptr = std::make_unique<Derived>();
```

Derived's destructor will not be called

Destructors again

When making use of polymorphism it is important to mark all destructors as virtual

This way a derived class can be destructed from a base reference

```
class Base
{
public:
  virtual ~Base() {...}
};

class Derived : public Base {...};

std::unique_ptr<Base> base_ptr = std::make_unique<Derived>();
```

`Derived`'s destructor will now be called

Casting

Casting from Derived to Base is called upcasting
This is always allowed for public inheritance

Casting from Base to Derived is called downcasting
Only allowed using an explicit cast operation

Dynamic casting

For downcasting we need a dynamic cast

```
int main()
{
  Derived derived_object;
  Base & bref = derived_object;

  Derived & dref = dynamic_cast<Derived&>(bref);
}
```

Dynamic cast checks at runtime whether the downcast is actually allowed

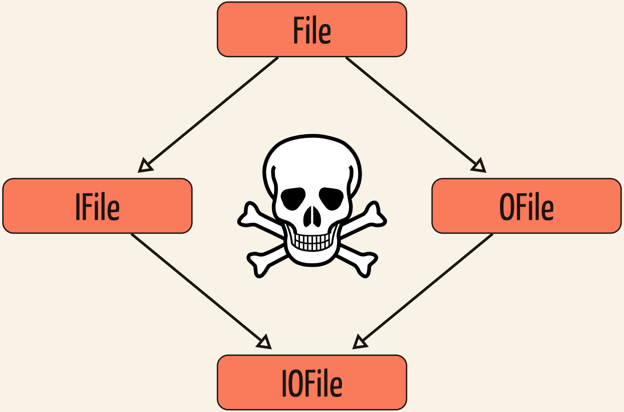
Slicing

Passing a derived class by value results in slicing

without the
compiler complaining

Upcasting happens automatically, and all the class specialisations are stripped away

The deadly diamond of death



The deadly diamond of death

There are multiple solutions available

- Using virtual inheritance

```
class IFile : public virtual File {};
```

- Have the base class be pure virtual
- Rewrite your class structure

Templates

Motivation

Say that you have overloaded a function to work for several types

```
void valueLog(int val, Logger & logger)
{
  logger << "[value]: " << val << std::endl;
}

void valueLog(double val, Logger & logger)
{
  logger << "[value]: " << val << std::endl;
}

// ...
```

Motivation

If the function bodies are identical {or could be made identical}
this is code duplication, and should be avoided

Can only be done with templates

Function templates

Can be replaced with `class`

```
template <typename ValueType>
void valueLog(ValueType val, Logger & logger)
{
  logger << "[value]: " << val << std::endl;
}
```

Compiles for every type that can be streamed to a Logger {which again could be a template function}

Function templates

Name of the undefined type

```
template <typename ValueType>
void valueLog(ValueType val, Logger & logger)
{
  logger << "[value]: " << val << std::endl;
}
```

Compiles for every type that can be streamed to a Logger {which again could be a template function}

Generic programming

Using templates is another great tool for abstraction

You can make assumptions on your template types, and the assumptions will be checked by the compiler for every instance

Generic programming

```
template <typename Container>
void print(const Container & container)
{
  std::cout << "{";
  for (auto i = 0; i < container.size(); ++i)
    std::cout << container[i] << ",";

  std::cout << "}";
}
```

Only compiles if

1. `Container` type has a `size()` member function

Generic programming

```
template <typename Container>
void print(const Container & container)
{
  std::cout << "{";
  for (auto i = 0; i < container.size(); ++i)
    std::cout << container[i] << ",";

  std::cout << "}";
}
```

Only compiles if

2. Type of `container.size()` is comparable to `int`

Generic programming

```
template <typename Container>
void print(const Container & container)
{
  std::cout << "{";
  for (auto i = 0; i < container.size(); ++i)
    std::cout << container[i] << ",";

  std::cout << "}";
}
```

Only compiles if

3. `Container` has overloaded the access operator `[]`

Generic programming

```
template <typename Container>
void print(const Container & container)
{
  std::cout << "{";
  for (auto i = 0; i < container.size(); ++i)
    std::cout << container[i] << ",";

  std::cout << "}";
}
```

Only compiles if

4. The type returned by `container[i]` can be streamed to `std::cout`

Generic programming

```
template <typename Container>
void print(const Container & container)
{
  std::cout << "{";
  for (auto i = 0; i < container.size(); ++i)
    std::cout << container[i] << ",";

  std::cout << "}";
}
```

Only compiles if

5. Both member functions `size()` and `[i]` are const

Return value templates

Can make functions that only differ in return type

```
template <typename Type>
Type null()
{
  return static_cast<Type>(0);
}
```

But you have to tell the compiler which function to call

```
auto zero = null<int>();
auto uzero = null<unsigned>();
auto first_char = null<char>();
```

Template specialisations

Changing the template function for a specific type is called template specialisation

```
template <>
char null()
{
  return 'a';
}
```

```
template <>
std::string null()
{
  return {""};
}
```

Disabling specialisations

Similarly to how one can disable the constructors and assignment operators in class declaration, one can disable templates for specific types

```
template <>
void* null() = delete;

template <>
std::ostream null() = delete;
```


{C++11}

Class templates

We can also make template classes

```
template <typename ValueType>
class Rational
{
private:
  ValueType num, den;
};

Rational<double> frac;
Rational< std::complex<double> > compl_frac;
```

Note: please don't accidentally write the stream operator

Partial specialisation

Class templates can also have partial template specialisations

```
template <typename Type>
class Foo {};

template <typename Type>
class Foo<Type*> {};
```

Specialisation for pointer types

Can’t have that for functions

(but you could always make function objects)

Use case: type traits

```
template <typename Type>
class type_trait
{
public:
  using value_type = typename Type::value_type;
};

template <typename Type>
class type_trait<Type*>
{
public:
  using value_type = Type;
};
```

Use case: type traits

```
template <typename Type>
class type_trait
{
public:
  using value_type = typename Type::value_type;
};

template <typename Type>
class type_trait<Type*>
{
public:
  using value_type = Type;
};
```

A **dependent** name

Separating declaration and definition

Separating template functions and classes into separate header and source files are unfortunately not that easy, and doing it the naive way will most likely result in linking errors

For a solution either have a look at the C++ Super FAQ (link on the homepage) or write all template functions as inline functions (which is what libraries do)

Template Meta Programming

It was discovered (by accident) that the C++ template system and pattern lookup is itself Turing complete

It is possible to program with the template system

Compile time factorials

```
template <unsigned N>
struct Factorial
{
  static constexpr unsigned value {N*Factorial<N-1>::value};
};

template <>
struct Factorial
{
  static constexpr unsigned value {0};
};

int main()
{
  std::cout << Factorial<10>::value << std::endl;
}
```

Compile time factorials

```
template <unsigned N>
struct Factorial
{
  static constexpr unsigned value {N*Factorial<N-1>::value};
};

template <>
struct Factorial
{
  static constexpr unsigned value {0};
};

int main()
{
  std::cout << Factorial<10>::value << std::endl;
}
```

Runtime constant

Programming Practices

Learn to love the compiler

Warnings are also great

```
g++ -pedantic -Wall -Wextra -Wcast-align -Wcast-qual
  -Wctor-dtor-privacy -Wdisabled-optimization -Wformat=2
  -Winit-self -Wlogical-op -Wmissing-declarations
  -Wmissing-include-dirs -Wnoexcept -Wold-style-cast
  -Woverloaded-virtual -Wredundant-decls -Wshadow
  -Wsign-conversion -Wsign-promo -Wstrict-null-sentinel
  -Wstrict-overflow=5 -Wswitch-default -Wundef -Werror
  -Wno-unused -o program code.cpp
```

Everything that helps you avoid runtime debugging is invaluable

Good Programming Practices

- Always make virtual destructors for classes meant for inheritance
- Use templates to avoid code duplication
- Abstractions is a tool for the educated mind

Fundamental theorem of software engineering

We can solve any problem by introducing an extra layer of indirection

Recap

Recap Day 4

- You can create new classes based on existing ones through inheritance
- Encapsulation is very strict, friendship is not transitive
- Public inheritance is like an ”is-a” relationship

Recap Day 4

- Polymorphism is when you let derived classes act as its base class
- Templates can be used to write type independent, generic programs
- Template metaprogramming is awesome