# Generic Programming

Different ways to achieve similar things

Thomas Madlener

Sep 10, 2021

## Generic programming

- Not going into too much detail here what is and what isn't *generic programming*
- Usually: Keeping algorithms and their implementations separate from their usage with dedicated types
- Here also: How to write code that can be easily reused
    - E.g. `sort`, `max`, `min` should work with different types of "lists"
- Mainly focussing on some techniques that can be used to achieve things that are useful for *generic programming*
    - Focus on C++ with some examples in python

## C++ techniques towards generic programming

- Template classes and functions
- Virtual inheritance / classic polymorphism
- `std::variant` / sum types
- “Type erasure”

## Template functions and classes

- In C++ classes and functions can be “templatized”
- Provide a template with a yet unspecified type and let the compiler fill that template with live (“instantiate”)
- E.g. `std::vector` is a template class, it is a dynamic array of elements of one type

# C++ templates vs. python

```cpp
// a template function
template<typename Animal>
std::string do_sound(const Animal& a) {
  return a.sound();
}

struct Duck {
  std::string sound() const { return "quack"; }
};

struct Goose {
  std::string sound() const { return "honk"; }
};

struct Dog {
  std::string sound() const { return "woof"; }
};

std::cout << do_sound(Duck()) << '\n';
std::cout << do_sound(Goose()) << '\n';
std::cout << do_sound(Dog()) << '\n';
```

← /vrGqoWdj6

```python
def do_sound(animal):
  return a.sound()

class Duck:
  def sound(self):
    return 'quack'

class Goose:
  def sound(self):
    return 'honk'

class Dog:
  def sound(self):
    return 'woof'

print(do_sound(Duck()))
print(do_sound(Goose()))
print(do_sound(Dog()))
```

- NOTE: **do_sound<Dog>** and **do_sound<Duck>** are two different functions even though the template only appears once

## Polymorphism using virtual inheritance

- Typically used when there is a hierarchy of classes that all have similar functionality
    - “Types have the same interface”
- The exact details of how this functionality is implemented might be different for specific types in this hierarchy
- Classical way of generic programming in object oriented programming

## C++ (virtual) inheritance vs. python

/GTTd4Ths3

```cpp
struct Animal {
  virtual std::string sound() const = 0;
};

struct Duck : public Animal {
  std::string sound() const override { return "quack"; }
};

struct Goose : public Animal {
  std::string sound() const override { return "honk"; }
};

struct Dog {
  std::string sound() const { return "woof"; }
};

// Use like this
std::vector<Animal*> animals = {new Duck(), new Goose()};
for (const auto* a : animals) {
  std::cout << a->sound() << '\n';
}

// This DOES NOT WORK (fails to compile)
// Dog is not part of the hierarchy!
std::vector<Animal*> animals = {new Dog()};
```

```python
class Animal:
  def sound(self):
    raise NotImplementedError

class Duck(Animal):
  def sound(self):
    return 'quack'

class Goose(Animal):
  def sound(self):
    return 'honk'

class Dog:
  def sound(self):
    return 'woof'

animals = [Duck(), Goose()]
for a in animals:
  print(a.sound())

# This DOES WORK
# Python's "duck typing" doesn't really care
# about type hierarchies
for a in [Duck(), Goose(), Dog()]:
  print(a.sound())
```

## Sum Types / `std::variant`

- Also known as: “tagged union”
- A sum type holds a value that will at any given time be **exactly one** of an arbitrary but **fixed set of types**
- `std::variant` has been standardized in C++17
    - Standard compliant implementations also exist

## Example with C++ `std::variant`

```cpp
struct Duck {
  std::string sound() const { return "quack"; }
};

struct Goose {
    std::string sound() const { return "honk"; }
};

using Animal = std::variant<Duck, Goose>;

std::string do_sound(const Animal& animal) {
 return std::visit([](const auto& a) {
   return a.sound();
 }, animal);
}

// Use like this
std::vector<Animal> animals = {Duck(), Goose()};
for (const auto& a : animals) {
  std::cout << do_sound(a) << '\n';
}
```

- Classes in a `std::variant` do not have to be in a class hierarchy
- They have to offer all the functionality that is used via the variant
    - `sound` in this case
- The canonical way to invoke any function on the variant is via `std::visit`
    - Takes a callable (*Visitor*) that can be called with all the types in the `std::variant`
- /156fqWKn9

## Type erasure (in C++)

- A possible technique to make value semantics possible in C++
- Used in e.g. in `std::function` and `std::any`

# Type erasure implementation

```cpp
class Animal {
  struct AnimalConcept {
    // see next slide
  };

  template<typename T>
  struct AnimalModel : public AnimalConcept {
   // see next slide
  };

public:
  std::string sound() const { return m_concept->sound(); }

  // Constructor from arbitrary types
  template<typename T>
  Animal(T&& t) :
    m_concept(new AnimalModel<T>(std::forward<T>(t))) {}

  ~Animal() { delete m_concept; }

private:
  AnimalConcept* m_concept;
}
```

- Type erasure works by defining the type erased class which uses two internal classes to which it delegates the work
- It combines virtual inheritance and templated classes
- Fully functional implementation needs a few more things!
  - Mainly for resource management
  - Make the type erased class behave more like a value

# Type erasure implementation - internal classes

```cpp
class Animal {
  struct AnimalConcept {
    virtual std::string sound() const = 0;
  };

  template<typename T>
  struct AnimalModel : public AnimalConcept {
    std::string sound() const override {
      return m_instance.sound()
    };

  template<typename U>
  AnimalModel(U&& u) :
    m_instance(std::forward<U>(u)) {}
  private:
    T m_instance;
  };

// the rest of the implementation from previous slide
};
```

- *Concept* defines the interface
- *Model* implements that interface and holds the actual value
- Have to "say it three times", because every function needs:
    - a declaration in the concept,
    - an implementation in the model,
    - and a call in the type erased class

# Type erasure usage

```cpp
struct Duck {
  std::string sound() const { return "quack"; }
};

struct Goose {
  std::string sound() const { return "honk"; }
};

struct Car {
  std::string sound() const { return "wroom"; }
};

// Now the only thing that really matters is
// whether all expected functionality is there
std::vector<Animal> = {Duck(), Goose(), Car()};
for (const auto& a : animals) {
  std::cout << a.sound() << '\n';
}
```

- Once everything is in place usage is almost python like ;)
- Still checked at compile time if all the types that are used actually fulfill all the functionality
  - Could restrict the constructor to make this more strict if necessary
- /94x7nYaje