An Introduction to Generic Programming

Global Goals

- Become acquainted with the generic programming model
- Develop and use template code in C++
- Learn main functionality in STL (Standard Template Library)
- Use Generic Programming together with OOP

2

An Introduction to Generic Programming (GP)

- OOP is based on classes, instances and inheritance
- GP = data + algorithms
- The data is generic
- In C++, we call it template programming
- STL, Boost, your own template classes

3

What is OOP?

- Classes and inheritance
- Subtype polymorphism (virtual functions in base class)
- These are the interfaces requirements
- Most C++ programmers know and use this technique

4

What is GP?

- Separates data structure and algorithms
- Use abstract requirements specification
- We can use parametric polymorphism
- Similar to Abstract Data Type (ADT) behaviour
- Focus is on efficiency and compile-time behaviour

5

Shape Hierarchy

```
class Shape
{
public:
  virtual void draw() const = 0;
};

class Line : public Shape
{
public:
  void draw() const { cout << "Line" << endl; }
};

class Circle : public Shape
{
public:
  void draw() const { cout << "Circle" << endl; }
};
```

6

Subtype Polymorphism

```
void draw(const list<Shape*>& v)
{
  cout << "\nSubtype polymorphism\n";
  list<Shape*>::const_iterator it;

  for (it = v.begin(); it != v.end(); ++it)
  {
    (*it)->draw();
  }
}
```

7

Features of this Solution

- Traditional OO code
- Works with derived classes of Shape
- Run-time bahaviour (vtabl)
- Not highly reusable code

8

Valve Hierarchy

```
class Valve
{
public:
  virtual void draw() const = 0;
};

class BallValve: public Valve
{
public:
  void draw() const { cout << "Ball valve" << endl; }
};

class CheckValve: public Valve
{
public:
  void draw() const { cout << "Check valve" << endl; }
};
```

9

Parametric Polymorphism

```
template <class DrawableThing> void draw(const vector<DrawableThing>& v)
{
  cout << "\nParametric polymorphism\n";
  vector<DrawableThing>::const_iterator it;

  for (it = v.begin(); it != v.end(); ++it)
  {
    (*it)->draw(); // Make assumption on the type(pointer)
                   // of DrawableThing
  }
}
```

Assumption: DrawableThing object must have member function —> draw()

10

Features of this Solution

- Operates with any class that implements a draw function
- Simple example of a policy
- Compile-time behaviour
- Code is reusable, and it depends on standardised interface

11

Usage, Hierarchy I

```
cout << "\nNow printing Valves\n";
BallValve bv1;
CheckValve cv1;

vector<Valve*> network(2);
network[0] = &bv1;
network[1] = &cv1;

draw(network);
```

12

Usage, Hierarchy II

```
Line lin;
Circle cir;

cout << "\nNow printing shapes\n";
vector<Valve*> figure(2);
figure[0] = &lin;
figure[1] = &cir;

draw(figure);
```

13

Traditional Usage

```
Line lin;
Circle cir;

// Now the traditional OO approach
list<Shape*> figure2;
figure2.push_back(&lin);
figure2.push_back(&cir);

draw(figure2);
```

Using virtual function is slower than using template

14

Comparing OOP and GP

- Virtual functions are slower than function templates
- Run-time dispatching or compile-time dispatching?
- Risk of code bloat with template-based programs
- Reusability: too much inheritance limits it

Discovery and Documentation

- Need to switch mindset from traditional OOP
- Instead of inheritance hierarchies, think of contracts between ADTs
- Will formalise this process later
- Document using UML components diagrams

OOP or GP?

- Can simulate inheritance with generics
- Can simulate genericity with inheritance (common in GOF patterns)
- Can mix the two paradigms using inheritance, aggregation
- OOP support run-time subtype polymorphism, GP supports compile-time parametric polymorphism

17

Example
Generic Function

```
template <typename N>
N Max(const N& x, const N& y)
{ // N is generic type

  if (x > y) return x;

  return y;
}
```

18

Using Template Function

```
int d1, d2;
cout << "Give the first number: ";  cin >> d1;
cout << "Give the second number: "; cin >> d2;

char c;       // Character type
cout << "Which function a) Max() or b) Min()? ";
cin >> c;

if (c == 'a')
{
  cout << "Max value is: " << Max<int>(d1, d2) << endl;
}
else
{
  cout << "Min value is: " << Min<int>(d1, d2) << endl;
}
```


Remarks

- We can integrate OOP and GP in later modules
- Inheritance, aggregation and composition possible
- Most developers feel comfortable with OOP
- GP is necessary for the future