

# Generics/Templates

Steven R. Bagley

# Code reuse

- Often we implement a class...
- Only to find that next week, we need to reuse it with just a slight tweak
- Collections, Containers are good examples

```
class CIntStack
{
public:
    CIntStack();
    ~CIntStack();

    void Push(int val);
    int Pop();

private:
    int m_stack[256];
    int *m_stackPtr;
};

void CIntStack::Push(int val)
{
    *--m_stackPtr = val;
}

int CIntStack::Pop()
{
    return *m_stackPtr++;
}
```

# Make mine a double...

- What if we need a stack of doubles?
- Code is virtually identical
- Replace all occurrences of int with double

```

class CIntStack
{
public:
    CIntStack();
    ~CIntStack();
    void Push(int double val);
    int double Pop();
private:
    int double m_stack[256];
    int double *m_stackPtr;
};

void CIntStack::Push(int double val)
{
    *--m_stackPtr = val;
}

int double CIntStack::Pop()
{
    return *m_stackPtr++;
}

```

# What more types?

- Doubles? I need objects stacking...
- Time to rewrite the code again...
- Isn't there an automatic way to handle it?

# Polymorphic Container

- Create a type that you store in the containers
- Everything else inherits from this type
- E.g. `java.lang.Object`
- Exactly what Java did (at first)
- Write once, can store anything

# Contains Anything

- Disadvantage, can store anything
- Mixed Collections
- Container can't type check what is stored
- Client code needs to be aware of this
- This behaviour can be useful in some circumstances

# Two Options

- Code containers that can store anything
- Hand-code containers to be fixed to a type, for each type we need
- Can we automate this last process?

# Auto Code Generation

- This is what Generics/Templates/Paramaterized classes do
- Generates a specialized class at compile-time automatically
- Identical to coding them by hand

# How it works

- Define the class in terms of a generic type
- Provide the implementation as normal
- But use the generic type rather than a specific type
- Tell the compiler the type we want it to use when we instantiate object

```

template<class T>
class CStack
{
public:
    CStack();
    ~CStack();

    void Push(T val);
    T Pop();

private:
    T m_stack[256];
    T *m_stackPtr;

};

```

```

void CStack::Push(T val)
{
    *--m_stackPtr = val;
}

T CStack::Pop()
{
    return *m_stackPtr++;
}

```

C++

# C++ implementation

- `template<...>` defines class as parameterized
- `class T` defines parameters
  - `class` means the type changes (doesn't have to be an object)
  - Can also provide other forms of parameterization
- Use the identifiers (`T`) to refer to the types

# Using Templates

- The easier bit to understand...
- Syntax is similar in Java and C++
- Class name is decorated with the type to create
- E.g. `CStack<int>` or `CStack<double>`
- Use in the same fashion as a hand coded one

```
CStack<int> *intStack = new CStack<int>;

intStack->Push(42);
intStack->Push(1963);
intStack->Push(10101010);

int i = intStack->Pop();
cout << i << endl;

CStack<string> *stringStack = new CStack<string>;

stringStack->Push("Hello");
stringStack->Push("World!");
stringStack->Push(42); /* won't compile, 42 is not
a string */
```

```
java.util.Stack myStk = new java.util.Stack();  
  
myStk.push(new Integer(42));  
myStk.push(new Integer(128));  
myStk.push("Hello World"); /* Will compile,  
everything is an object */
```

# Code Specialization

- When the compiler comes across:  
`new CStack<int>`
- Generates the code need to support a  
stack of `ints`
- Just as if we created it by hand
- And so on for any other types used in the  
program

```
template<class T>
class CStack
{
public:
    CStack();
    ~CStack();

    void Push(T val);
    T Pop();

private:
    T m_stack[256];
    T *m_stackPtr;

};
```

```
void CStack::Push(T val)
{
    *--m_stackPtr = val;
}

T CStack::Pop()
{
    return *m_stackPtr++;
}
```

C++

```
template<class T>
class CStack
{
public:
    CStack();
    ~CStack();

    void Push(int val);
    int Pop();

private:
    int m_stack[256];
    int *m_stackPtr;

};
```

```
void CStack::Push(int val)
{
    *--m_stackPtr = val;
}

int CStack::Pop()
{
    return *m_stackPtr++;
}
```

C++

# Code Specialization

- Strongly typed — different code to handle `ints` as `strings` or `objects`
- Can lead to code bloat
- Pointers to objects could be handled by identical code

# C++ Implementation

- Compiler needs access to template implementation when template used
- Very tricky to implement in the compiler
- Some compilers have had bugs

# Java Generics

- Available since Java 1.5
- Similar syntax  
`class<T>` instead of `template...`
- Still uses `T` in place of actual types

```
class CStack<T>
{
    public CStack();
    { ... }

    public void Push(T val) { ... }
    public T Pop() { ... }

    ...
};
```

Java

# Different Implementation

- Java does not use Code Specialization
- Java uses *Code Sharing*
- Only one instance of the code for a generic class created
- `T` gets translated to `Object`, when compiling this shared instance
- Type hiding

# Code Sharing

- All generic classes use the same implementation in Java — unlike C++
- Can do this because everything is derived from `java.lang.Object`
- I.e. identical to our polymorphic container
- Compiler ensures at compile-time that the type is correct

# Using Java Generics

- Type is ensured when the methods are called
- Compile-time error generated if not
- Cast back to type if returned
- Identical to how you'd do it if you used the polymorphic container
- But type checking is automatic

# Java vs C++

- Upside to Java approach, less code generated
- Downside, technically possible to use items of the wrong type
- Code generated is identical to old-style polymorphic container
- Can be cast to be an old-style container

```
ArrayList<Integer> il = new ArrayList<Integer>();
ArrayList al;
Iterator<Integer> it;
Object in;

for(int i = 0; i < 10; i++)
{
    il.add(new Integer(i));
}

al = (ArrayList)il;
al.add(new Double(3.14159));

System.out.println("Count"+il.size());
it = il.iterator();
do
{
    in = it.next();
    System.out.println(in);
}
while(it.hasNext());
```

# Conclusion

- Generics/Templates provide a way to avoid rewriting code
- Write generic, customize on use
- Can be tricky to use and implement