C++: Templates

Miro Jurišić
meeroh@meeroh.org

Why templates

- Compile-time invariance over types
- Contrast with runtime invariance over specialization (classes)
- Example: array of integers, array of characters
- Example: max (a, b)
- Parameterized types: class templates
- Function templates

Life before templates – class 'templates'

```
class Array {
    public:
        Array (int size);

    void
        Insert (void* item);

    void*
        ItemAt (
            int index);

    private:
        void** items;
};
```

- Unsafe: no compile-time type checking

Life before templates – function 'templates'

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

- Macros are unsafe: no type checking, side effects

Basic template syntax

```
template <typename T1, ...>     // Template parameters
class Foo {
                                // Use parameters in the class declaration
};

template <typename T1, ...>     // Template parameters
...                             // Use parameters in the function return type
functionName (...)              // Use parameters in the function argument list
{
                                // Use parameters in the function body
};
```

Simple example: homogeneous pair

```
template <typename T>
class HomPair {
    public:
            HomPair (
                const T&    inElt1,
                const T&    inElt2);

        T
            Element1 () const;

        T
            Element2 () const;

    private:
        T       mElt1;
        T       mElt2;
};
```

```
template <typename T>
HomPair <T>::HomPair (
    const T&    inElt1,
    const T&    inElt2):
    mElt1 (inElt1),
    mElt2 (inElt2)
{
}

template <typename T>
T
HomPair <T>::Element1 () const
{
    return mElt1;
}

template <typename T>
T
HomPair <T>::Element2 () const
{
    return mElt2;
}
```

```
HomPair <int>           intPair (1, 2);

Point   pt1;
Point   pt2;
HomPair <Point>         pointPair (pt1, pt2);

HomPair <HomPair <int> >    intIntPair1 (HomPair <int> (1, 2), HomPair <int> (3, 4));

typedef HomPair <int>       IntPair;
HomPair <IntPair>           intIntPair2 (IntPair (1, 2), IntPair (3, 4));
```

Simple example: max (a, b)

```
template <typename T>
T
max (
    const T& inLeft,
    const T& inRight)
{
    if (inLeft < inRight) {
        return inRight;
    } else {
        return inLeft;
    }
}

int     xi = 1;
int     yi = 2;
int     maxi = max <int> (xi, yi);
```

```
double  xd = 1.0;
double  yd = 2.0;
double  maxd = max <double> (xd, yd);
```

Default template parameters

```
template <typename T1, typename T2 = T1>
class HetPair {
    public:
            HetPair (
                const T1&   inElt1,
                const T2&   inElt2);

        T1
            Element1 () const;

        T2
            Element2 () const;

    private:
        T1      mElt1;
        T2      mElt2;
};
```

```
HetPair <int, double>                pair1 (1, 2.0);
HetPair <int, HetPair <int, int> >   pair2 (1, HetPair <int, int> (2, 3));

HetPair <int>                        pair3 (1, 2);
HetPair <int, HetPair <int> >        pair4 (1, HetPair <int> (2, 3));
```

Function template parameter deduction

```
template <typename T>
T
max (
    const T& inLeft,
    const T& inRight)
{
    if (inLeft < inRight) {
        return inRight;
    } else {
        return inLeft;
    }
}

int     xi = 1;
int     yi = 2;
int     maxi = max (xi, yi);
```

```
double  xd = 1.0;
double  yd = 2.0;
double  maxd = max (xd, yd);
```

```
template <typename T1, typename T2>
HetPair <T1, T2>
MakeHetPair (
    const T1&       inElt1,
    const T2&       inElt2)
{
    return HetPair <T1, T2> (inElt1, inElt2);
}
```

```
int      x = 1;
int      y = 2;
float    z = 3.0;
HetPair <int>            pair1 = MakeHetPair <int, int> (x, y);
HetPair <int>            pair2 = MakeHetPair <int> (x, y);
HetPair <int>            pair3 = MakeHetPair (x, y);

HetPair <int, float>     pair4 = MakeHetPair <int, float> (x, z);
HetPair <int, float>     pair5 = MakeHetPair <int> (x, z);
HetPair <int, float>     pair6 = MakeHetPair (x, z);

HetPair <float>          pair7 = MakeHetPair <float> (x, z);
```

Integer template parameters

```
template <int N>
class IntArray {
    public:
        int ItemAt (int inIndex) const;

    private:
        int     mItems [N];
};
```

Function template specialization

```
template <typename T>
T
ConstructDefault ()
{
    return T;
}

template <>
int
ConstructDefault <int> ()
{
    return 42;
}

int     x = ConstructDefault <int> ();
string  x = ConstructDefault <string> ();
```

Class template specialization

```
template <>
class HomPair <bool> {
    public:
            HomPair (
                bool        inElt1,
                bool        inElt2);

        bool
            Element1 () const;

        bool
            Element2 () const;

    private:
        char        mElts;
};
```

```
template <>
HomPair <bool>::HomPair (
    bool        inElt1,
    bool        inElt2):
    mElts (inElt1 | (inElt2 << 1))
{
}

template <>
bool
HomPair <bool>::Element1 () const
{
    return mElts & 0x1;
}

template <>
bool
HomPair <bool>::Element2 () const
{
    return mElts & 0x2;
}
```

Parameter-dependent types

```
template <typename T1, typename T2 = T1>
class HetPair {
    public:
        typedef     HetPair <T2, T1>        ReversePair;

    // ...
};

template <typename T1, typename T2>
typename HetPair <T1, T2>::ReversePair
Reverse (
    const HetPair <T1, T2>&     inPair)
{
    return typename HetPair <T1, T2>::ReversePair (inPair.Element2 (), inPair.Element1 ());
}

HetPair <int, HetPair <int> >           pair1 (1, HetPair <int> (2, 3));
HetPair <HetPair <int>, int>            pair2 = Reverse (pair1);
```

Member templates

```
class DiskStorage {
    public:
        template <typename T>
        std::size_t
            StorageSize (
                const T&        inObject);
};

DiskStorage storage;
Class c;
size_t size = storage.StorageSize (c);
```

```
template <typename T1>
class SmartPointer {
    public:
            SmartPointer (
                T1*                             inPointer);

        T1*
            Get () const;

        template <typename T2>
            SmartPointer (
                const SmartPointer <T2>&        inSmartPointer);

    private:
        T1*     mPointer;
        int*    mRefCount;
};

SmartPointer <Base>         p1 (new Base ());
SmartPointer <Derived>      p2 (new Derived ());
SmartPointer <Base>         p3 (p2);
```

```
template <typename T1>
template <typename T2>
SmartPointer <T1>::SmartPointer (
    const SmartPointer <T2>&        inSmartPointer):
    mPointer (inSmartPointer.Get ())
{
    ++(*mRefCount);
}
```

Pros and cons

- Less repetition
- Type safety
- Compiler writes the code for you

- More typing, use typedefs
- Compiler writes the code for you – code bloat