Template Variations

Version 1: Dr. Ofir Pele
Version 2: Dr. Erel Segal-Halevi

Template Variations

Can receive several arguments

```
template <typename T1, typename T2>
class Pair {
    T1 x;
    T2 y;
    bool operator> (Pair<T1,T2> other) {
      return (x > other.x || (x == other.x && y > other.y));
    }
};
int main()  {
    Pair<double, int> a; }
```

Template Variations (folder 1)

Can receive constant integral arguments

```
template<typename T,
         int Size>
class Buffer {
private:
   T m_values[Size];
};
Buffer<char,1024> Buff2;
Buffer<int, 256> Buff3;
```

Template Variations

Can set up default values for arguments

```
template<typename T= char,
         size_t Size= 1024>
class Buffer {
private:
   T m_values[Size];
};
Buffer<char> Buff1;
Buffer<> Buff2; // same as Buff1
Buffer<int, 256> Buff3;
```

Template and Types

- `Buffer` is not a type
- `Buffer<char,1024>` is a type
- `Buffer<char>, buffer<char,1024>` are two names for the same type
- `Buffer<char,256>` is a different type

Optimizations

Sometimes, when the size is small, the compiler will **unfold loops.** For example, a copy of Buffer<int,3> to another Buffer<int,3> will be unfolded to three assignments, that may even be parallelized**.**

```
#include <iostream>
using namespace std;

template<int n>
void print() {
    for (int iiii=0; iiii<n; iiii++) {
        cout << iiii;
    }
}

void print1(int n) {
    for (int iiii=0; iiii<n; iiii++) {
```

Template Variations

The **placeholder type** can be used in the types list:

```
template
<typename T, T default_value>
class Buffer { public:
   Buffer(int size) {
       p = new T[size];
       for (int i=0; i<size; ++i)
            p[i] = default_value;
   }
   void def()  { cout << default_value << endl; }
};
int main() {
   Buffer<int,7> a (100);
   Buffer<char,'h'> b (100);
   // Buffer<char,7.8> c;
```

```
output //
7
h
```

Template Variations

Can evaluate the type:

```
template <typename T>
void foo(T* p)
{
}
int main()
{
    int d;
    foo(&d); // foo(int *) will be expand
             // and called (T = int)
}
```

Template Variations

```
template <typename T>
void foo(LinkedList<T*> list) {...}
```

```
int main() {
 LinkedList<int*> l1;
 LinkedList<LinkedList<int>* > l2;
 foo(l1); // T = int
 foo(l2); // T = LinkedList<int>
}
```

Template Variations

```
template <typename T>
void foo(LinkedList<T*> list) {...}

template <typename T>
void zoo(LinkedList<LinkedList<T>*> list) {...}

int main() {
 LinkedList<int*> l1;
 LinkedList<LinkedList<int>* > l2;
 foo(l1); // T = int
 foo(l2); // T = LinkedList<int>
 zoo(l2); // T = int
}
```

Template Specialization

Template specialization

```
template <typename Type>
class Wrapper {
public:
    Type _data;
    Wrapper(const Type& data)
        : _data(data) { }
    bool isBigger
        (const Type& data) const;
};
template <typename Type>
bool Wrapper<Type>::isBigger
    (const Type& data) const {
    return data > _data;
}
```

```
int main()
{
    Wrapper<char*> a("hi");
    std::cout <<
    a.isBigger("bye");
}
```

Template specialization

```
template <typename Type>
class Wrapper {
public:
   Type _data;
   Wrapper(const Type& data)
      : _data(data) { }
   bool isBigger
      (const Type& data) const;
};
template <typename Type>
bool Wrapper<Type>::isBigger
   (const Type& data) const {
   return data > _data;
}
```

```
int main()
{
   Wrapper<int> ai(5);
   Wrapper<char*> as("hi");
   ai.isBigger(7);
   //generic isBigger()
   as.isBigger("bye");
   //specific isBigger()
}
```

```
template <>
bool Wrapper<char*>::isBigger(const char*& data) const {
   return strcmp(data, _data) > 0;
}
```

swap specialization – IntBufferSwap (folder 2)

```
template<> Pair<char*,char*>::operator> {

}
```

Template specialization: checking types at compile time:

```
#include <iostream>

template<typename T> struct is_numeric: std::false_type {};

template<>    struct is_numeric<int> : std::true_type {};
template<>    struct is_numeric<double> : std::true_type {};

int main() {
    std::cout << is_numeric<char>::value << '\n';
    std::cout << is_numeric<int>::value << '\n';
}

template<typename T> T plus(T a, T b) {
   if (is_numeric<T>::value)
       return a+b;
   else
       throw “not a number”;
}
```

Template specialization: checking types at compile time:

```
template <typename TNom,typename TDen> struct ErrorOnDivide
{
   enum { ProblemToDivideByZero= 1, NonDivideable = 1 };
};

template <> struct ErrorOnDivide<int,int>
{
   enum { ProblemToDivideByZero= 1, NonDivideable = 0 };
};

template <> struct ErrorOnDivide<double,double>
{
   enum { ProblemToDivideByZero= 0, NonDivideable = 0 };
};
```

Checking types at compile time (folder 3)

```
template <typename TNom, typename TDen, typename TRes >
void SafeDiv(const Tnom& nom, const Tden& den, TRes& res) {
   // static_assert only in c++11 (supported in current compilers)
   static_assert(
      ErrorOnDivide<TNom,TDen>::ProblemToDivideByZero==0 &&
      ErrorOnDivide<TNom,TDen>::NonDivideable==0,
      "Division not safe");
   res=nom/den;
}


int main() {
   double res;
   SafeDiv(10.0,3.0,res); //OK
   SafeDiv(10,3,res); //Compilation Error "division not safe"
}
```

Template specialization: memory efficiency (folder 4)

Example:

specializing `vector<T>` to `vector<bool>`
to reduce memory space – save 8 bools in one char.

Template Meta-Programming

Template Meta-Programming

```
// primary template computes 3 to the Nth
template<int N> class Pow3 {  public:
   enum { result=3*Pow3<N-1>::result };
};
// full specialization to end recursion
template<> class Pow3<0> {  public:
   enum { result = 1 };
};
int main(){
  cout << Pow3<1>::result<<"\n"; //3
  cout << Pow3<5>::result<<"\n"; //243
  return 0;
}
```

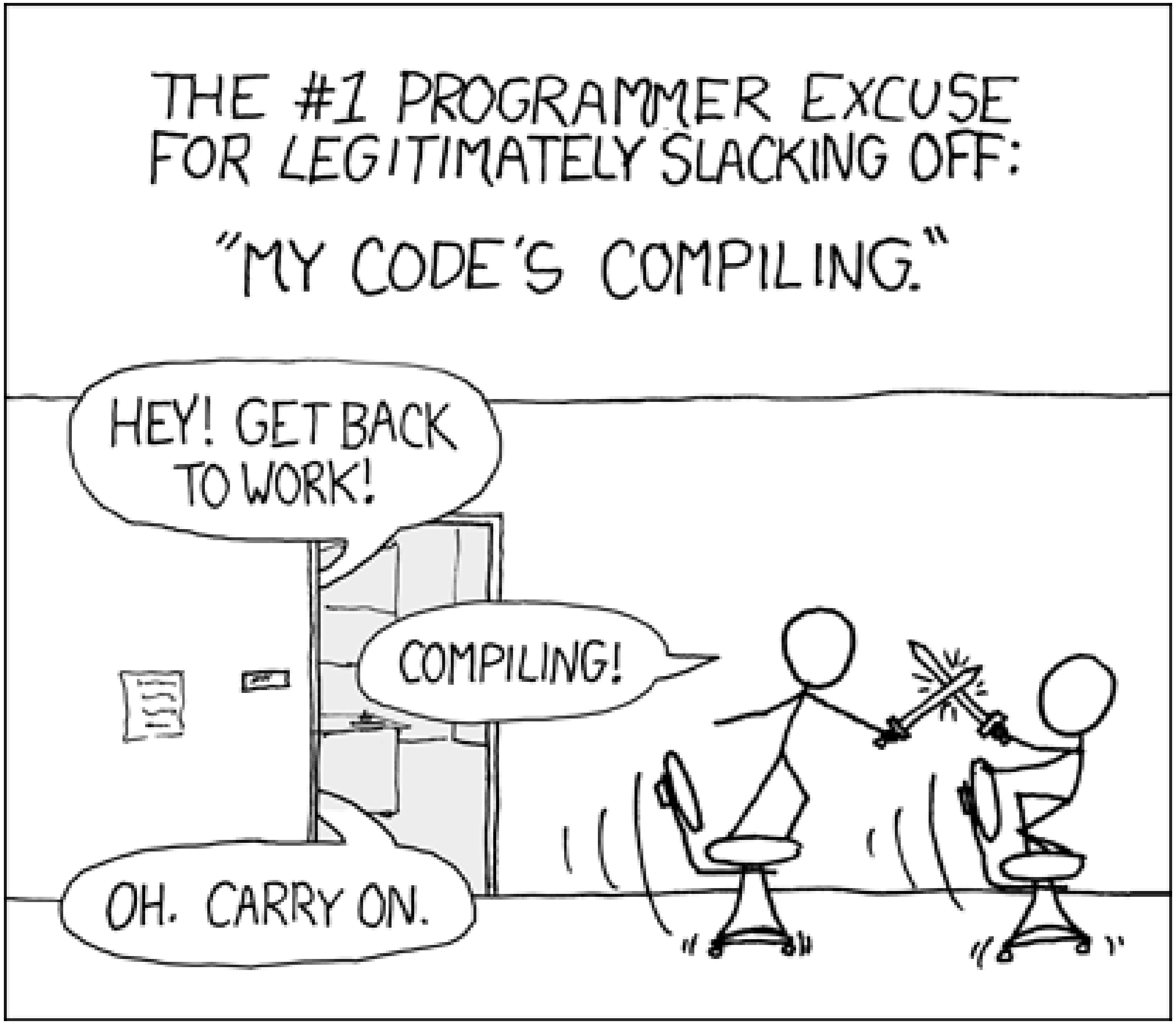
Template Meta-Programming (folder 5)

Numerically calculating and plotting the n-th derivative.

Templates - recap

1. Compile time polymorphism / meta programing – do whatever possible in compilation instead of run time.
2. Arguments are types or integral constants.
3. Efficient but large code.
4. Longer compilation time (but precompiled header can help)

Longer compilation time is not always a bad thing (from xkcd):



Polymorphism vs. Templates

- Templates compilation time is much longer than using inheritance.
- Using templates enlarge the code size.
- Compilation errors can be very confusing.

- Templates running time is much faster than using inheritance.
- Combined with inlining, templates can reduce runtime overhead to zero.

Polymorphism vs. Templates

- Templates allow static (compile-time) polymorphism, but not run-time polymorphism.
- You can actually combine them.
- As always – think of **your** task.

Summary

- Compile-time mechanism to polymorphism
- It might help to write & debug concrete example (e.g., intList) before generalizing to a template
- Understand iterators and other “helper” classes
- Foundation for C++ standard library