Templates

Before we dive in

- Preprocessing
- Compilation
- Linkage

Motivation

A useful routine to have is

```
void swap( int& a, int &b )
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Example

What happens if we want to swap a double ? or a string?

For each one, we need different function:

```
void swap( double& a, double &b )
{
    double tmp = a; a = b; b = tmp;
}
void swap( string& a, string &b )
{
    string tmp = a; a = b; b = tmp;
}
```

Generics Using void*

C approach:

```
void swap( void *a, void *b, size_t size )
{
    for (size_t i=0; i<size; i++) {
        char t = *(char *)(a+i);
        *(char *)(a+i) = *(char*)(b+i);
        *(char*)(b+i) = t;
    }
    // or can be done using malloc and memcpy
}
```

Generic Programming

All these versions of Swap are “isomorphic”:

```
// code for swap for arbitrary type T
void swap( T& a, T &b )
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Can we somehow tell the **compiler (not preprocessor)** to use swap with any type T?

Function Templates

The `template` keyword defines “templates”

Piece of code that will be regenerated with different arguments each time

```
template<typename T> // T is a
                     //"type argument"
void swap( T& a, T& b )
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

These two definitions are exactly the same, if we have both, we will have a compilation error:

```
template <typename T>
void swap(T &a, T &b)
{
   T c = a;
   a = b;
   b = c;
}
```

```
template <class P>
void swap(P &a, P &b)
{
   P c = a;
   a = b;
   b = c;
}
```

Template Instantiation

```
int main()
{
    int a = 2;
    int b = 3;
    swap( a, b ); // requires swap(int&, int&)
}
```

The compiler encounters `swap(T&, T&)`

It **instantiates** the template `swap` with `T = int`
and compiles the code defined by it.

Template Instantiation

Different instantiations of a template can be generated by the same program

```
int a = 2;
int b = 3;
cout << "Before " << a << " " << b << "\n";
swap( a, b ); // Compiler generates the code for
              // swap(int&,int&)
cout << "After " << a << " " << b << "\n";
double x = 0.1;
double y = 1.0;
cout << "Before " << x << " " << y << "\n";
swap( x, y ); // Compiler generates the code for
              // swap(double&,double&)
cout << "After " << x << " " << y << "\n";
```

Template Instantiation

Explicit

```
double d1 = 4.5, d2 = 6.7;
swap<double>(d1, d2);
```

Implicit

```
double d1 = 4.5, d2 = 6.7;
swap(d1, d2);
```

Template assumptions

```
// example of a uncopyable class
class Cookie
{
private:
   // private copy operations
   Cookie(const Cookie&);
   Cookie & operator=(const Cookie&);
public:
   Cookie() { };
};

...
 Cookie van, choc;
 swap(van, choc); /* compiler will try generate code for
swap(Cookie&,Cookie&), but will fail, claiming an error
somewhere at the declaration of swap*/
```

Templates & Compilation

- A template is a **declaration**
- The compiler performs functions/operators calls checks only when a template is instantiated with specific arguments - then the generated code is compiled.

Implications:

1. Template **code** has to be visible by the code that uses it (i.e., appear in header *.h/.hpp* file)
2. Compilation errors can occur only in a specific instance

Templates & Compilation

- A template is a **declaration**
- The compiler performs functions/operators calls checks only when a template is instantiated with specific arguments - then the generated code is compiled.

Implications:

1. Template **code** has to be available to the code that uses it (i.e., appear in header *.h/.hpp* file)
2. Compilation errors can occur only in a specific instance

Not just the declaration

Another Example

```
// Inefficient generic sort
template< typename T >
void sort( T* begin, T* end )
{
    for( ; begin != end; begin++ )
        for( T* q = begin+1; q != end; q++ )
        {
            if( *q < *begin )
                swap( *q, *begin );
        }
}
```

What are the template assumptions?

Another Example

```
// Inefficient generic sort
template< typename T >
void sort( T* begin, T* end )
{
    for( ; begin != end; begin++ )
        for( T* q = begin+1; q != end; q++ )
        {
            if( *q < *begin )
                swap( *q, *begin );
        }
}
```

What are the template assumptions?

Usage

Suppose we want to avoid writing operator != for new classes

```
template <typename T>
bool operator!= (T const& lhs, T const& rhs)
{
   return !(lhs == rhs);
}
```

When is this template used?

Usage

```
class MyClass
{
public:
   bool operator==
   (MyClass const & rhs) const;
};
int a, b;
if( a != b ) // uses built in
             // operator!=(int,int)
...
MyClass x,y;
if( x != y ) // uses template with
             // T= MyClass
```

When Templates are Used? – overloads, again

When the compiler encounters

```
...
f( a, b )
...
```

1. Look for all functions named f.
2. Creates instances of all templates named f according to parameters.
3. Order them by matching quality (1..4).
4. **Within** same quality, prefer non-template over template.

Example 1

```
#include <iostream>
#include <typeinfo>
void foo(int x) {
   std::cout << "foo(int)\n";
}
void foo(double x) {
   std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.2);
   foo("abcdef");
   return 0;
}
```

Example 1

```
#include <iostream>
#include <typeinfo>
void foo(int x) {
   std::cout << "foo(int)\n";
}
void foo(double x) {
   std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.2);
   foo("abcdef");
   return 0;
}
```

```
foo(int)
foo(double)
foo<char>(T*)
Press any key to continue . . . _
```

Example 3

```
#include <iostream>
#include <typeinfo>
//void foo(int x) {
//   std::cout << "foo(int)\n";
//}
void foo(double x) {
   std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.2);
   foo("abcdef");
   return 0;
}
```

Example 3

```
#include <iostream>
#include <typeinfo>
//void foo(int x) {
//   std::cout << "foo(int)\n";
//}
void foo(double x) {
   std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.2);
   foo("abcdef");
   return 0;
}
```

```
foo(double)
foo(double)
foo<char>(T*)
Press any key to continue . . .
```

Example 2

```
#include <iostream>
#include <typeinfo>
void foo(int x) {
   std::cout << "foo(int)\n";
}
//void foo(double x) {
//  std::cout << "foo(double)\n";
//}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.2);
   foo("abcdef");
   return 0;
}
```

Example 2

```
#include <iostream>
#include <typeinfo>
void foo(int x) {
   std::cout << "foo(int)\n";
}
//void foo(double x) {
//  std::cout << "foo(double)\n";
//}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.2);
   foo("abcdef");
   return 0;
}
```

```
foo(int)
foo(int)
foo<char>(T*)
Press any key to continue . . .
```

Example 3

```
#include <iostream>
#include <typeinfo>
void foo(int x) {
   std::cout << "foo(int)\n";
}
//void foo(double x) {
//  std::cout << "foo(double)\n";
//}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.2);
   foo("abcdef");
   return 0;
}
```

Won't compile with the flag:
-Wconversion

Example 4

```
#include <iostream>
#include <typeinfo>
//void foo(int x) {
//   std::cout << "foo(int)\n";
//}
void foo(double x) {
   std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
   std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main()
{
   foo(42);
   foo(42.0);
   foo("abcdef");
   return 0;
}
```

Will compile with the flag:
-Wconversion

Example 5

```
template<typename T>
void f(T x, T y)
{
     cout << "Template" << endl;
}

void f(int w, int z)
{
     cout << "Non-template" << endl;
}

int main()
{
    f( 1 ,  2 );
    f('a', 'b');
    f( 1 , 'b');
}
```

```
Non-template
Template
Non-template
Press any key to continue . . .
```

Example 6

```
template<typename T>
void f(T x, T y)
{
     cout << "Template" << endl;
}

void f(int w, int z)
{
     cout << "Non-template" << endl;
}

int main()
{
     f( 2 , 1.0);
     f( 2.0 , 1.0);
}
```

```
Non-template
Template
Press any key to continue . . .
```

Example 7

```
template <typename T> void f(T) { cout << "Less specialized");}
template <typename T> void f(T*) { cout<< "More specialized");}
int main() {
   int i =0;
   int *pi = &i;
   f(i); // Calls less specialized function.
   f(pi); // Calls more specialized function.
}
```

1

2

Is there a type that fits 1 and will not fit 2? If so, 2 is more specialized and should be preferred.

Interim summary

- We saw template method
  - Declaration + Definition
  - Instantiation

- Now we will see a template class

String Stack

```
class StrStk {
public:
  StrStk():m_first(nullptr) { }
  ~StrStk() { while (!isEmpty()) pop(); }
  void push (string const& s )  {m_first=new Node(s,m_first);}
  bool isEmpty() const {return m_first==nullptr;}
  const string& top () const {return m_first->m_value;}
  void pop ()
  {Node *n=m_first; m_first=m_first->m_next; delete n;}
private:
  StrStk(StrStk const& rhs);    StrStk& operator=(StrStk const& rhs);
  struct Node {
    string m_value;
    Node* m_next;
    Node(string const& v ,Node* n):m_value(v),m_next(n) { }
  };
  Node* m_first;
};
```

Generic Classes

- The actual code for maintaining the stack has nothing to do with the particulars of the string type.
- Can we have a generic implementation of stack?

Generic Stack (Stk.hpp)

```
template <typename T> class Stk {
public:
  Stk():m_first(nullptr) { }
  ~Stk() { while (!isEmpty()) pop(); }
  void push (T const& s )  {m_first=new Node(s,m_first);}
  bool isEmpty() const {return m_first==nullptr;}
  const T& top () const {return m_first->m_value;}
  void pop ()
  {Node *n=m_first; m_first=m_first->m_next; delete n;}
private:
  Stk(Stk const& rhs); Stk& operator=(Stk const& rhs);
  struct Node {
    T m_value;
    Node* m_next;
    Node(T const& v ,Node* n):m_value(v),m_next(n) { }
  };
  Node* m_first;
};
```

Class Templates

```
template<typename T>
class Stk
{
   ...
};

Stk<int> intList; // T = int
Stk<string> stringList; // T = string
```

Class Templates

The code is similar to non-template code, but:

- Add `template<…>` statement before the class definition
- Use template argument as type in class definition
- To implement methods outside the class definition (but still in header: .h.hpp, not in a cpp file!):

```
template <typename T>
bool Stk<T>::isEmpty() const
{
  return m_first==nullptr;
}
```

Example of generic programming - Iterators

Constructing a List

- We want to initialize a stack from an array

- We can write a method that receives a pointer to the array, and a **size argument**

```
int arr[] = { 1, 2, 3, 4, 5, 6 };
Stk<int> stk;
stk.push(arr, sizeof(arr)/sizeof(*arr) );
```

- Alternative: use a pointer to initial position and **one to the position after the last**:

```
stk.push(arr, arr + sizeof(arr)/sizeof(*arr) );
```

This form is more flexible (as we shall see)

Constructing a List

```
// Fancy copy from array
template< typename T >
Stk<T>::push(const T* begin, const T* end)
{
   for(const T* p=begin; p!=end; ++p)
   {
      push(*p);
   }
}
```

Constructing a List

```
// Fancy copy from array
template< typename T >
Stk<T>::push(const T* begin, const T* end)
{
   for(; begin!=end; ++begin)
   {
      push(*begin);
   }
}
```

Pointer Paradigm

Code like:

```
const T* begin=arr;
const T* end= arr+sizeof(arr)/sizeof(*arr);
for(; begin!=end; ++begin)
{
    // Do something with *begin
}
```

- Applies to all elements in [begin,end-1]
- Common in C/C++ programs
- Can we extend it to other containers?

Iterator

- **Object that behaves just like a pointer (or "weak" pointer)**
- Allows to iterate over elements of a container

Example:

```
Stk<int> L;
...
Stk<int>::iterator i;
for( i = L.begin(); i != L.end(); i++ )
    cout << " " << *i << "\n";
```

Iterators

To emulate pointers, we need:

1. copy constructor
2. `operator= (copy)`
3. `operator== (compare)`
4. `operator* (access value)`
5. `operator++ (increment)`

And maybe:

1. `operator[] (random access)`
2. `operator+= / -= (random jump)`
3. ...

Stk<T> iterator

Create an inner class, keep a pointer to a node.

```
class iterator
{
private:
   Node *m_pointer;
};
```

Provides encapsulation, since through such an iterator we cannot change the structure of the list

Stk.hpp

Initializing a Stk

We now want to initialize a stack from using parts of another stack. Something like:

```
Stk(iterator begin, iterator end) {
    for(; begin!=end; ++begin) {
        push(*begin);
    }
}
```

Initializing a Stk

Compare:

```
Stk(iterator begin, iterator end) {
   for(; begin!=end; ++begin) {
      push(*begin);
   }
}
```

To:

```
template< typename T >
Stk<T>::push(const T* begin, const T* end) {
   for(; begin!=end; ++begin) {
      push(*begin);
   }
}
```

Generic Constructor

The code for copying using

- T*
- Stk<T>::iterator

are essentially identical on purpose --- iterators mimic pointers!

Can we write the code once?

Yes: Stk.hpp

Template Variations

Template Variations

Can receive several arguments

```
template <typename T1, typename T2>
class A
{
    T1 _d;
    T2 _c;
};
int main()
{
    A<double, int> a;
}
```

Template Variations

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, int 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

Template Variations

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

```
template
<typename T, T val>
class A
{
public:
   void foo()
   {
      std::cout << val
      << std::endl;
   }
};
```

```
int main()
{
   A<int,7> a;
   A<char,'h'> b;
   // A<char,7.8> c;
   // compilation error
   a.foo(); b.foo();
}
```

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

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

Template specialization

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

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

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

Template specialization

```
template <typename Type>
class A {
publi
    Ty
    A
    bo
};
temp
bool
    (
    re
}
```

```
int main()
```

```
i");
;
gger()
ye");
igger()
```

Although this is a classic example of template specialization, it is not clear that this is a good example as we change the behavior of the function!

An example of a problem with these kinds of template specialization: vector<bool> in the standard library.

A better example will be that we just change the time complexity.

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

swap specialization

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 };
};
```

Template specialization: checking types at compile time:

```
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 Meta-Programming

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

The typename keyword - reminder (1)

```
template <class T>
class A
{
public:
   T _data;
   A(T data)
   :_data(data) { }
};
```

```
template <typename T>
class A
{
public:
   T _data;
   A(T data)
   : _data(data) { }
};
```

Another keyword to define type parameters in templates

The typename keyword (2)

Resolve "ambiguity":

```
template<typename T>
class X
{
    T::Y * _y;
};
int main()
{
}
// sometimes compilation error
```

The typename keyword (3)

Resolve "ambiguity":

```
template<typename T>
class X
{
   typename T::Y * _y; // treat Y as a type
};
int main()
{
}
// OK
```

The typename keyword (4)

```
template <typename T>
T inner_product (const vector<T>& v1,
                 const vector<T>& v2) {
   T res= 0;
   typename vector<T>::const_iterator iter1;
   typename vector<T>::const_iterator iter2;
   for (iter1= v1.begin(), iter2= v2.begin();
        iter1!=v1.end();
        ++iter1,++iter2) {
      assert(iter2!=v2.end());
      res+= (*iter1) * (*iter2);
   }
   assert(iter2==v2.end());
   return res;
}
```

C++11 no need for typename

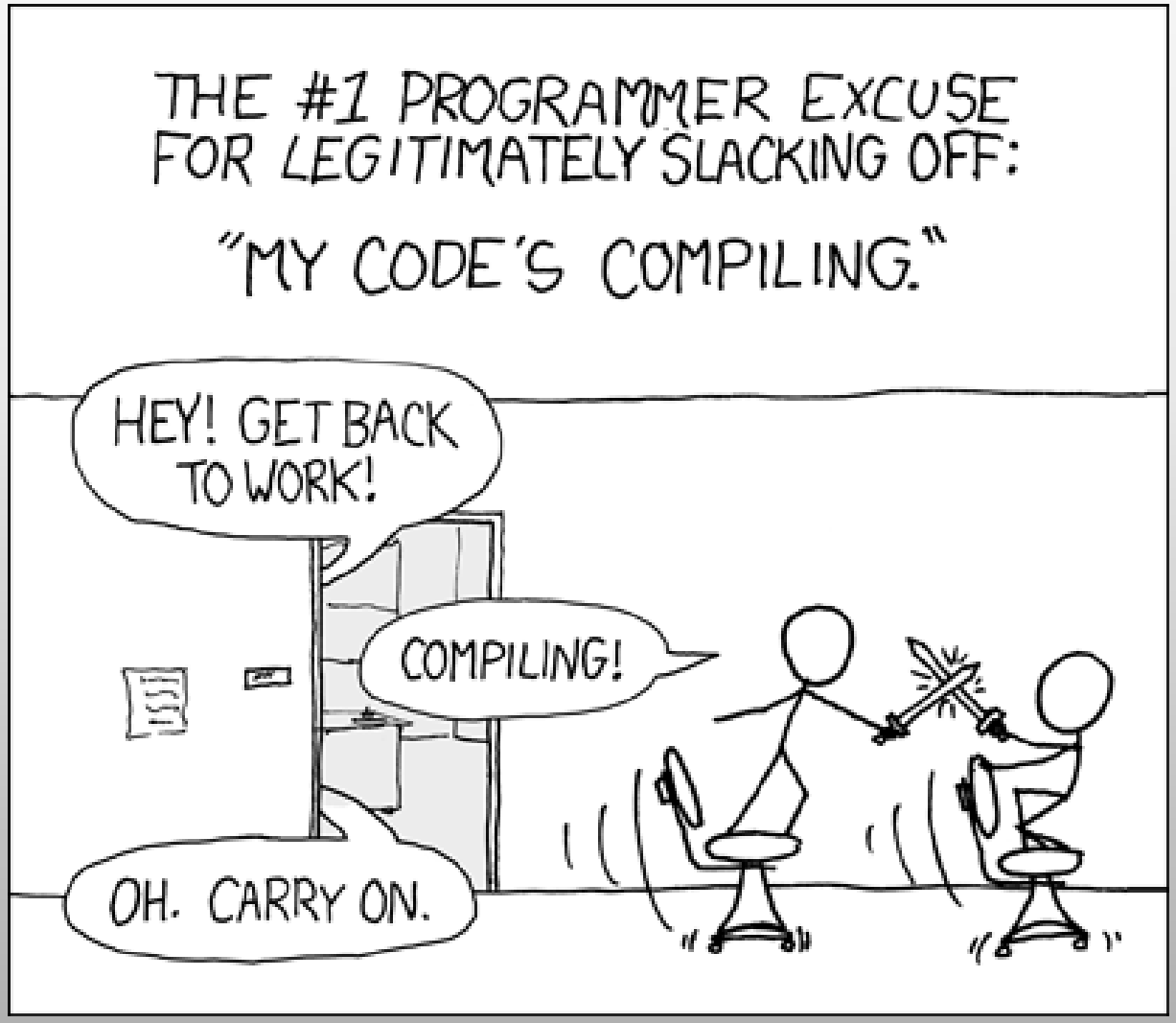
```
template <typename T>
T inner_product (const vector<T>& v1,
                 const vector<T>& v2) {
   T res= 0;
   auto iter1= v1.cbegin();
   auto iter2= v2.cbegin();
   for ( ; iter1!=v1.cend(); ++iter1,++iter2) {
      assert(iter2!=v2.cend());
      res+= (*iter1) * (*iter2);
   }
   assert(iter2==v2.cend());
   return res;
}
```

Inner product is actually in the standard library and is more general…

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