Diet Templates

Jonathan Wakely

The Myth of Bloat

“Template Bloat”

Ten years ago it was often claimed that C++ templates caused code bloat.

At one time I was told to avoid using the STL because it would lead to fatter, slower code.

To <T> or not to <T>?

C++ had just been standardized and the STL was supposed to be a Good Thing.

But if it's the Standard *Template* Library, won't it cause code bloat?

Will my code really be slower if it uses templates?!

Understanding Templates

So I read CUJ, Dr. Dobbs, Overload etc.

I learnt about the history and design of the STL.

I played with the compiler.

I decided template code is no bigger than “normal” code.

Instantiated as needed

Instantiating `std::vector<int>` and `std::vector<char>` results in no more code than hand-written `vector_int` and `vector_char` classes would.

Often *less* code because unused member functions of class templates won't be instantiated.

Reuse, reuse, reuse

Write your own templates once and reuse them for any number of types.

For every instantiation you use you've saved the time of modifying existing code to work with the new type.

The more you instantiate the more you save!

There are several advantages to using templates.

They support and encourage powerful programming styles.

Not using them would be a mistake!

So I did.

The Reality of Bloat

Missed the bloat

The truth isn't quite as simple.

Some template-heavy code *is* bloated.

Where does this come from?

Does it matter?

Compiler support

It took some time for toolchains to implement templates efficiently.

Template definitions must be in visible to be instantiated, so get compiled in every file using them

- Either can't use separate compilation,
- or must manage explicit instantiations,

At least one compiler supported an instantiation model where all template code had static linkage and every instantiation was duplicated in every file that used it.

Compiler support

Early implementations of the Standard Library varied in quality and completeness.

Features such as partial specialization and member templates weren't widely available.

Bloated object files

Template instantiations are compiled multiple times, so compilation takes longer.

Template instantiations appear in multiple object files

... but the linker only keeps one copy

Symbol names might grow due to including template parameters

But those are compile-time problems.

Excessive inlining

Complete definitions of templates are usually in headers.

This allows compilers to statically resolve functions calls and perform aggressive inlining across function boundaries.

This can be a huge advantage for optimization!

But sometimes this is undesirable – and can be hard to control because everything is in headers.

Don't Repeat Yourself

A single template might by used to generate dozens of distinct instantiations.

The generated code might be very similar at the instruction level, maybe even identical.

e.g. std::vector<char*> and std::vector<int*>

This repetition isn't apparent in the code, because you've only written the template once.

So what?

Aren't memory and disks always getting cheaper, bigger and faster?

- CPU caches improve more slowly
- Some programs **must** be *as fast as possible*

The problems are not inherent in templates

If we're aware of the problems we can try to avoid them

Bloat in detail

Duplicate code

Can happen in several ways

- Separate instantiations for types with the same representation (e.g. `char*` and `int*`)
- Non-generic functionality which doesn't need to be in generic code
- Unnecessary dependency on template parameters

```
template<typename T>
  class Vec {
    T* ptr;
    size_t size;
    size_t capacity;
  public:
    // ...
    void push_back(const T& t)
    {
      if (size == capacity)
        reallocate(size+1);
      ptr[size++] = t;
    }
  };
```

```
_ZN3VecIPiE9push_backERKS0_:
.LFB2:
        .cfi_startproc
        .cfi_personality 0x3,__gxx_personality_v0
        movq    %rbx, -16(%rsp)
        movq    %rbp, -8(%rsp)
        subq    $24, %rsp
        .cfi_def_cfa_offset 32
        .cfi_offset 6, -16
        .cfi_offset 3, -24
        movq    %rdi, %rbx
        movq    %rsi, %rbp
        movq    8(%rdi), %rsi
        cmpq    16(%rdi), %rsi
        jne     .L2
        addq    $1, %rsi
        call    _ZN3VecIPiE10reallocateEm
.L2:
        movq    8(%rbx), %rax
        movq    (%rbx), %rdx
        movq    0(%rbp), %rcx
        movq    %rcx, (%rdx,%rax,8)
        addq    $1, %rax
        movq    %rax, 8(%rbx)
        movq    8(%rsp), %rbx
        movq    16(%rsp), %rbp
        addq    $24, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

```
_ZN3VecIPcE9push_backERKS0_:
.LFB2:
        .cfi_startproc
        .cfi_personality 0x3,__gxx_personality_v0
        movq    %rbx, -16(%rsp)
        movq    %rbp, -8(%rsp)
        subq    $24, %rsp
        .cfi_def_cfa_offset 32
        .cfi_offset 6, -16
        .cfi_offset 3, -24
        movq    %rdi, %rbx
        movq    %rsi, %rbp
        movq    8(%rdi), %rsi
        cmpq    16(%rdi), %rsi
        jne     .L2
        addq    $1, %rsi
        call    _ZN3VecIPcE10reallocateEm
.L2:
        movq    8(%rbx), %rax
        movq    (%rbx), %rdx
        movq    0(%rbp), %rcx
        movq    %rcx, (%rdx,%rax,8)
        addq    $1, %rax
        movq    %rax, 8(%rbx)
        movq    8(%rsp), %rbx
        movq    16(%rsp), %rbp
        addq    $24, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

Microsoft's linker uses heuristics to determine when two template instantiations produce the same code and can use a single version of the code.

Manual techniques for avoiding the duplication can produce even better results

- *“Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat”* Bourdev & Järvi, 2006

Template hoisting

Provide a common implementation for all instantiations which have the same behaviour and representation

Move all the code into that common implementation

Defer all real work to that common implementation

```
class Vec_ptr {
  void** ptr;
  size_t size;
  size_t capacity;
protected:
  // ...
  void P_push_back(void* t)
  {
    if (size == capacity)
      reallocate(size+1);
    ptr[size++] = t;
  }
};
template<typename T>
  class Vec<T*> : public Vec_ptr {
  public:
    void push_back(T* const& t)
    { this->P_push_back(t);}
  };
```

This means writing more code: the original class template, the base class and a partial specialization of the template

But the executable doesn't have duplicate instructions for all the instantiations that store pointers

... except when it does! The compiler might still inline the member functions of the base class into the derived class template

But we have more control over what must be defined inline in the header and what can be separately compiled

For `Vec<T*>` we can see that hoisting might be useful as soon as we write the class template

In other templates, hoisting might only become necessary as the code evolves

A piece of code might be suitable for writing as a template, but have extra non-generic functionality added later, which should be factored out to non-template code.

A class template that holds **N** bytes and an ID field:

```
template<unsigned N>
  struct DataBuffer<N> {
    unsigned char data[N];
    unsigned short typeId;
    // ...
  };
```

Which can be written to a socket:

```
template<unsigned N>
  void send(int sock, const DataBuffer<N>& buf)
  {
    write(sock, htons(buf.typeId), 2);
    write(sock, htonl(N), 4);
    write(sock, &buf.data, N);
  }
```

Over time the function gets extended:

```
template<unsigned N>
  void send(int sock, const DataBuffer<N>& buf)
  {
    LOG(net, DEBUG) << “Sending ” << N << “ bytes, type: “
      << buf.typeId;
    write(sock, htons(buf.typeId), 2);
    write(sock, htonl(N), 4);
    write(sock, &buf.data, N);
    if (isCheckedType(buf.typeId)
    {
      // calculate checksum
      // write checksum to socket
    }
  }
```

This code is not very dependent on the template parameter:

```
template<unsigned N>
  void send(int sock, const DataBuffer<N>& buf)
  {
    LOG(net, DEBUG) << "Sending " << N << " bytes, type: "
      << buf.typeId;
    write(sock, htons(buf.typeId), 2);
    write(sock, htonl(N), 4);
    write(sock, &buf.data, N);
    if (isCheckedType(buf.typeId)
    {
      // calculate checksum
      // write checksum to socket
    }
  }
```

This will generate nearly identical instructions for every different value of `N` used in the program

All the actual work can be hoisted out of the template, turning compile-time template parameters into run-time function paramters::

```
void send2(int sock, unsigned short type, void* buf,
           unsigned buflen);

template<unsigned N>
  inline void send(int sock, const DataBuffer<N>& buf)
  {
    send2(sock, buf.typeId, &buf.data, N);
  }
```

The function template can be inlined now without any negative consequences in code size

Any possible bloat due to `send()` is not *because* it's a template

As with `vector_int` and `vector_char`, just as much code would be generated if you wrote several functions without using a function template:

```
  send1() send2() send4() send16() ...
```

But noone would do that! Especially not in if they care about code size

Template hoisting is just the application of good practice for avoiding repetition, not coding via copy'n'paste, and creating a new class or function to replace similar code that appears in more than one place

You wouldn't duplicate all that code by hand, so try not to do it with templates!

The tricky part is that the repetition doesn't appear in the source code with templates

Be vigilant!

Usually a function template's behaviour really does depend on a template parameter, that's why it's written as a template

```
template<typename F>
  void Node::apply(F f)
  {
    f(*this);
    for (Node n : children)
      if (n.leaf())
        f(n);
    for (Node n : children)
      if (!n.leaf())
        n.apply(f);
  }
```

We want to be able to apply many types of function

```
struct PruneNode {
  void operator()(Node&) const;
};

void  printNode(Node& n) { std::cout << n.label(); }

RootNode root = fooNodes();

unsigned count = 0;
root.apply( [&count](Node const&) { ++count; } );

if (count > limit)
  root.apply( PruneNode() );

root.apply( PrintNode{std::cout} );
```

Each call instantiates the function template differently

As well as the functions themselves, the executable will contain different instructions for all those instantiations:

```
void Node::apply<void(*)(Node const&)>;
void Node::apply<λ>;
void Node::apply<PruneNode>;
```

C++0x to the rescue:

```
~~template<typename F>~~
  void Node::apply(std::function<void(Node&)> f)
  {
    f(*this);
    for (Node n : children)
      if (n.leaf())
        f(n);
    for (Node n : children)
      if (!n.leaf())
        n.apply(f);
  }
```

By using a function wrapper we support all the same types of function, without having to define `Node::apply()` as a function template

Does this change in code size really matter? Does it help?

Caches are tiny and the various function types `F` must be doing something, which will need to load `F`'s instructions into the I-cache for every call anyway

It might not matter. It might not help.

But there will only be one set of instructions for the apply function and so there's a better chance they will be kept in L2 or L3 cache between calls

SCARY assignments

```
std::set<int> s1;
std::set<int, std::greater<int>> s2;
std::set<int, MyCmp, MyAlloc> s3;
auto it = s1.begin();
it = s2.begin();
it = s3.begin();
```

Is this OK?!

SCARY assignments

Seemingly erroneous (appearing **C**onstrained by conflicting generic parameters), but **A**ctually work with the **R**ight implementation (unconstrained b**Y** the conflict due to minimized dependencies).

From "*Minimizing Dependenies within Generic Classes for Faster and Smaller Programs*" , OOPSLA'09, Tsafrir, Wisniewski, Bacon, Stroustrup

Standard set container:

```
template <class Key, class Compare = less<Key>,
             class Allocator = allocator<Key> >
class set {
public:
  // types:
  typedef Key                               key_type;
  typedef Key                               value_type;
  typedef Compare                           key_compare;
  typedef Compare                           value_compare;
  typedef Allocator                         allocator_type;

  typedef implementation defined            iterator;
  typedef implementation defined            const_iterator;
```

So are the SCARY assignments OK?!?!

Let's assume the implementation defines them as nested structs:

```
template <class Key, class Compare = less<Key>,
             class Allocator = allocator<Key> >
class set {
public:
  // types:
  typedef Key                                       key_type;
  typedef Key                                       value_type;
  typedef Compare                                   key_compare;
  typedef Compare                                   value_compare;
  typedef Allocator                                 allocator_type;

  struct iterator;
  struct const_iterator;
```

SCARY assignments

```
std::set<int> s1;
std::set<int, std::greater<int>> s2;
std::set<int, MyCmp, MyAlloc> s3;

// std::set<int, std::less<int>, std::allocator<int>>::iterator
auto it = s1.begin();

// std::set<int, std::greater<int>, std::allocator<int>>::iterator
it = s2.begin();  // ERROR!

// std::set<int, MyCmp, MyAllo>::iterator
it = s3.begin();  // ERROR!
```

This is definitely not OK!

```
std::set<int> s1;
count_if(s1.begin(), s1.end(), Pred())

// count_if<set<int, less<int>, allocator<int>>::iterator, Pred>

std::set<int, std::greater<int>> s2;
count_if(s2.begin(), s2.end(), Pred())

// count_if<set<int, greater<int>, allocator<int>>::iterator, Pred>

std::set<int, MyCmp, MyAlloc> s3;
count_if(s3.begin(), s3.end(), Pred())

// count_if<set<int, MyCmp, MyAlloc>::iterator, Pred>
```

BLOAT!

But if the implementation defines the iterator separately:

```
template <class T>
class tree_iterator { ... };


template <class Key, class Compare = less<Key>,
             class Allocator = allocator<Key> >
class set {
public:
  // types:
  typedef Key                              key_type;
  typedef Key                              value_type;
  typedef Compare                          key_compare;
  typedef Compare                          value_compare;
  typedef Allocator                        allocator_type;

  typedef tree_iterator<Key> iterator;
```

```
std::set<int> s1;
count_if(s1.begin(), s1.end(), Pred())

// count_if<tree_iterator<int>, Pred>

std::set<int, std::greater<int>> s2;
count_if(s2.begin(), s2.end(), Pred())

// count_if<tree_iterator<int>, Pred>

std::set<int, MyCmp, MyAlloc> s3;
count_if(s3.begin(), s3.end(), Pred())

// count_if<tree_iterator<int>, Pred>
```

SCARY assignments

```
std::set<int> s1;
std::set<int, std::greater<int>> s2;
std::set<int, MyCmp, MyAlloc> s3;

// tree_iterator<int>
auto it = s1.begin();

// tree_iterator<int>
it = s2.begin();  // OK

// tree_iterator<int>
it = s3.begin();  // OK
```

This is now OK

Seemingly erroneous (appearing **C**onstrained by conflicting generic parameters), but **A**ctually work with the **R**ight implementation (unconstrained b**Y** the conflict due to minimized dependencies).

The SCARY paper reports an experiment converting a iterator-based application to SCARY (unconstrained) iterators and obtaining performance improvements between 1.2 and 2.0 times over alternative iterator implementations