Overload issue 54 april 2003

contents

credits & contacts

Editor:
John Merrells,
merrells@acm.org
241 Heartwood Lane,
Mountain View,
CA 94041-11836, U.S.A

Advertising:
Pete Goodliffe, Chris Lowe
ads@accu.org

Membership:
David Hodge,
membership@accu.org
31 Egerton Road
Bexhill-on-Sea, East Sussex
TN39 3HJ, UK

Readers:
Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Website: http://www.accu.org/

Membership fees and how to join:

Basic (C Vu only): £15
Full (C Vu and Overload): £25
Corporate: £80
Students: half normal rate
ISDF fee (optional) to support Standards work: £21
There are 6 journals of each type produced every year.
Join on the web at www.accu.org with a debit/credit card, T/Polo shirts available.
Want to use cheque and post - email `membership@accu.org` for an application form.
Any questions - just email `membership@accu.org`.

Editorial - Digital Identity

For this editorial I am returning to a topic that I have often written about in these pages: digital identity. It is a topic of fundamental importance to the future of the Internet, but one that the mainstream computing press rarely tackles directly. I am currently securing an online application, and being lazy (in a good way in this case) I am trying to reuse as much existing code as possible. As there are many enterprise and Internet application developers in the Overload audience I thought my exploration into this area would be of general interest.

By securing an application I mean that only the right people can perform the right actions against the right things, without being observed. Let's use the application I'm working on as an example. It's a simple client-server document storage service that could be deployed within an enterprise, as part of a website implementation, or as a service on the Internet in its own right. The client connects over a TCP/IP connection and performs operations to manipulate the stored documents. The service must only be available to the users authorized by the system administrator, and each user can read and write their own documents, but can only read documents they don't own.

Let us break the security problem down into sub-problems and discuss each and its possible solutions in turn.

Identity

Identity is hard to define, but for our discussion let us say that an identity is a name within a namespace, and an associated set of attributes. The name identifies the identity, and the attributes provide us with information about the identity. When we refer to an identity we use the name of the identity.

My example application could maintain its own list of users, each of whom is assigned a username. But then each user is going to have to remember yet another username, and the administrators have yet another identity namespace to manage. A more sophisticated approach would be to reuse an existing identity management service, such as the Unix password file through the Unix API, or Unix Yellow Pages/NIS/NIS+, or a directory server over LDAP [1].

Authentication

Authentication is the process by which we prove we are who we say we are. This is done by offering up credentials that can be verified against the attributes of the identity. My example application could request both a username and a password. The username is the name of the identity, and the password is the credential than can be checked against the password stored as an attribute of the identity. But, this is yet another password for the user to remember, so it'd be better to reuse an existing authentication mechanism. We can use one of the above identity management interfaces to perform the password test for us.

Authorization

After a client has authenticated it performs some actions against the application, each of which must be authorized. Authorization is the process of deciding whether a client is permitted to perform the action it is attempting. My example application currently allows anyone to perform any action to any document, in other words there is no access control. Fundamentally there is a three dimensional Boolean array with the axes being: identity, action, and document. Obviously an array this size is impossible for an administrator to manage, so a more manageable representation must be used. The Unix solution is to break the identity axis down into user, system, and group, and to limit the actions to read, write, and execute, and then to have these permissions associated with each file in the file system. This isn't expressive enough, or granular enough for most applications. Directory servers on the other hand usually have very rich and expressive access control languages. Each identity is modeled as an entry, actions are usually modeled as attributes, and resources are usually modeled as entries. Other directory features such as groups, roles, and its hierarchical nature can be used to reduce the number of access control lists, and their complexity.

Confidentiality

When an application is deployed on a public network, within a website, or even within an enterprise, we may need to ensure that snooping of the network traffic is fruitless. The solution is to secure the network channel using cryptography, in the form of SSL/TLS [2]. (The latest version of the Secure Sockets Layer specification has been named the Transport Layer Security 1.0 specification.)

Solution #1

So far we have come to the conclusion that my sample application can be secured by using SSL/TLS for confidentiality and by using a directory server for identity management, authentication, and authorization.

Conveniently, high quality open source implementations of both are available in OpenSSL [3] and OpenLDAP [4].

This solution might normally be sufficient for most applications, but I want to consider this set of problems further.

Authentication

Password based systems are vulnerable to social issues. Users select poor passwords, share them with their friends, and tape them inside their desk drawer. A password policy is therefore required for enforcing good password selection. Directory services do support password policies, but they have no standard way of defining them. Other, more secure, authentication mechanisms could be used instead, such as public key cryptography, but implementing them is challenging.

My example application is currently behaving as an authentication proxy. The client presents the username and password to the server and the server passes them on to the directory server for authentication. When using more sophisticated authentication mechanisms the server is unable to proxy the authentication. The server must either perform the authentication itself, or the client must authenticate directly with the directory server and provide the server with some evidence that this authentication was successful.

Unfortunately LDAP does not provide a mechanism for the directory server to provide a token to the client so that it can prove to a third party that it has successfully authenticated. This leaves us with implementing authentication within the server application itself. Fortunately there is a standard framework defined for authentication mechanisms, called the Simple Authentication and Security Layer, or SASL [5]. SASL provides a framework for authentication mechanisms, so an application protocol need only support the framework in order to support all the authentication mechanisms that have been defined for SASL. Both the LDAP and IMAP protocol specifications defer to SASL for authentication, which suggests that the specification is of a high quality. Also, there is an open source implementation of SASL available from Carnegie Mellon University called Cyrus SASL [6].

Although SASL provides many standard authentication mechanisms developers still have to design their application protocols to carry the authentication messages back and forth. Every Internet protocol has a verb for the client to authenticate itself with the server, and they are all different. HTTP, POP, LDAP, IMAP, and FTP, amongst others, all have their own verbs for authenticating to the server. Couldn't this commonality have been factored out and pushed down the network stack so that all application protocols, whether Internet standard application protocols, or homegrown application protocols, could share this functionality? This is exactly the thinking that led Marshall Rose to define BEEP [7]. Marshall worked on many of the Internet standards for system management, messaging systems, and directory services, so knows how to design a good application protocol. BEEP provides a framework upon which an application protocol can be defined. HTTP is commonly being used in this way right now, but it offers very little, other than not being blocked by corporate firewalls. BEEP, in contrast, offers built in authentication mechanisms via SASL, framing of the application protocol verbs, multiple channels within a connection, connection confidentiality via SSL/TLS, negotiation of the confidentiality level, permits channels to be turned around, and allows TLS to be started and stopped during a connection.

Solution #2

So, after further consideration, we can upgrade my example application to support more sophisticated authentication mechanisms, by layering the application protocol over BEEP. Authorization has moved into the application, but we still make use of the directory server for identity management and authorization. An open source implementation of BEEP is available [8] that incorporates both OpenSSL and Cyrus SASL.

But, something interesting is going on; a whole new generation of digital identity development is occurring within the XML standards forums. In the next issue of Overload I plan to write about the standards being defined by the W3C and OASIS for application protocols, authentication, authentication assertions, access control, and single-sign-on. Perhaps it's possible there's a simpler solution to online application security out there.

John Merrells
merrells@acm.org

References

[1] LDAP - http://www.ietf.org/html.charters/ldapbis-charter.html
[2] TLS - http://www.ietf.org/rfc/rfc2246.txt
[3] OpenSSL - http://www.openssl.org/
[4] OpenLDAP - http://www.openldap.org/
[5] SASL - http://www.ietf.org/rfc/rfc2222.txt
[6] Cyrus SASL - http://asg.web.cmu.edu/sasl/
[7] BEEP - http://www.ietf.org/html.charters/beep-charter.html
[8] beepcore - http://www.beepcore.org/beepcore/home.jsp

Copy Deadlines

All articles intended for publication in *Overload 55* should be submitted to the editor by May 1st, and for *Overload 52* by July 1st.

A C++ Petri Net Framework For Multithreaded Programming

by David L. Nadle

One of the pitfalls of multithreaded programming is deadlock, a situation where each thread exclusively holds one resource while waiting for another's resource. Every non-trivial multithreaded program must contend with deadlocks. One strategy is to detect a deadlock at runtime and take some action to remove it (e.g. send a quit signal to one of the threads). Another approach is to design the program carefully to avoid deadlocks. In practice, this can be a difficult task. The Petri net framework presented in this article supports a hybrid approach, combining careful design with runtime checks to create a deadlock-free process.

Like UML activity diagrams, Petri nets are graphical representations of processes providing a state-oriented view and an activity-oriented view. In other words, Petri nets simultaneously represent the state of a system and what the system is doing. What makes Petri nets powerful is their semantics; formal mathematical analysis techniques can be used to determine characteristics of a given net. For example, it can be proven that a particular net with the right initial conditions will not reach a deadlocked state.

This article briefly discusses the properties of Petri nets and presents a demonstration of a (intentionally) poorly designed application using a C++ framework with Win32 synchronization objects. The framework supports rapid implementation of a process that has been described with a Petri net and is capable of runtime or post-build testing for deadlocks.

Petri Nets

The Petri Net is named after C.A. Petri, who developed the concept as part of his doctoral thesis in 1962. In mathematical terms, a Petri net is a directed bipartite graph. The two types of vertices are called places and transitions. The directed edges, known as arcs, connect places to transitions, and transitions to places.

A place is a container that holds zero or more tokens. The set of places and the number of tokens in each represents the state of the system. This set is called a marking. A transition represents an activity of the system. The arcs that point towards a transition are input arcs, and those that point towards a place are output arcs. Each arc has an associated arc expression, which indicates how many tokens will be added to or removed from a place when the transition executes. The arc expression is usually 1, in which case it is omitted from drawings.

A transition is considered enabled when enough tokens exist in the places connected to its input arcs for all input arc expressions to be satisfied. Only enabled transitions can execute. After an enabled transition has completed, tokens are added to the places connected to its output arcs according to the output arc expressions. The net now has a new marking, and a new set of enabled transitions exists. A dead marking, or deadlocked state, is one where the set of enabled transitions is empty.

Figure 1 is a Petri net representation of a theoretical, and flawed, file printing application with its initial marking displayed. The application has a hold-and-wait problem and will deadlock. Places `P0`, `P3`, `P6`, and `P7` contain one token each. The tokens at `P0` and `P3` represent the number of threads which can execute in the left chain of execution and the right chain, respectively. The tokens at `P6` and `P7` represent a lock on a file or printer resource, respectively. A single token in each resource indicates that the locks are exclusive.

Concurrency and Conflict

The initial marking `M0` of the example Petri net can also be described as the set of enabled transitions; `M0 = { T0, T3 }`. If `T0` fires first, `T3` is still enabled, and vice versa. `T0` and `T3` are enabled concurrently. By systematically tracing execution of the Petri net and the evolution of its markings we build what is called the occurrence graph, the graph of all markings reachable from the initial marking. Firing `T0` gives `M1 = { T1, T3 }`. If `T1` fires first, `T3` is disabled, and vice versa. This situation is called a conflict. `T1` and `T3` are enabled, but not concurrently.

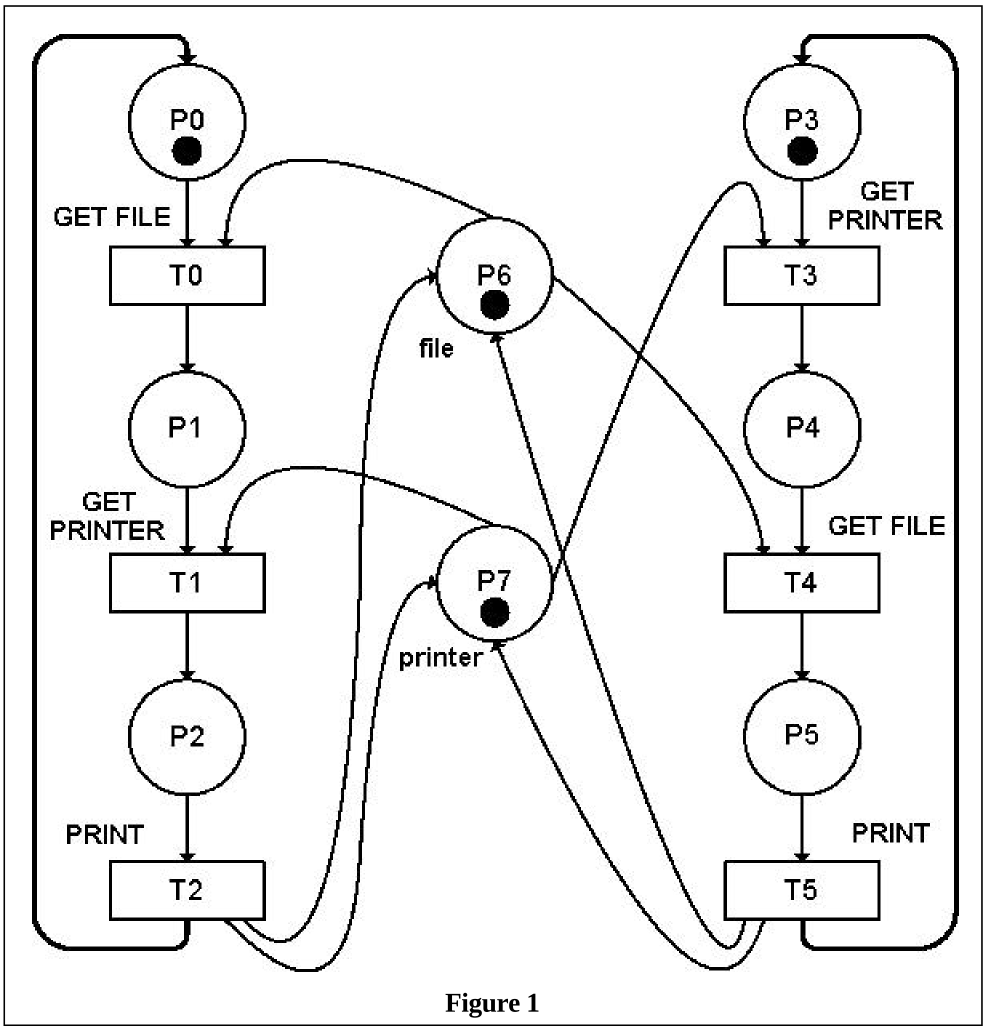

Figure 1

The most efficient multithreaded applications would maximize concurrent states and minimize conflicted states, and Petri net analysis can help in their design.

Continuing our systematic execution, `T1` gives `M2 = { T2 }`, and `T2` gives `M0` again. No problems yet. Let's return to `M1`, but this time `T3` fires first, giving `M3 = { }`. Deadlock. If through some intervention we were to give `T0` and `T1` priority over `T3`, we will have created a live-lock situation. The chain on the right (`T3`, `T4`, `T5`) would never execute.

The Framework

The Petri net framework is anchored by the `CPetriNet` class, which aggregates `Place`, `Transition`, `Arc`, and `Thread`. Listing 1 shows how a net is constructed and operated in a console application `main()` function. Both `Place` and `Transition` inherit from `Node`, which keeps track of arc connections. For details on these classes see the full source archive.

Places 0-5 in the example are generic Petri net places, which provide valuable state information but do not represent resources. `Resource` classes inherit from `Place` and implement an additional interface to the resource.

`Transition` is an abstract class. Users implement the `Execute()` method in subclasses. Each transition is executed by the next available thread. It's important not to think of the execution chains in Figure 1 as cycles for a single thread. Resource interfaces in classes inheriting from `Place` must not use per-thread synchronization or locking mechanisms. A properly constructed net provides the necessary locking.

`Execute()` methods use the Visitor pattern (Gamma et al., 1995) to access `Place` resources. Classes derived from `Place` define an `Accept()` method for each client class derived from `Transition`. `Execute()` methods call `Accept()` on each connected place in some defined order, as shown below.

```
void DerivedTransition::Execute() {
  arcInput_[1]->
            GetPlace()->Accept(*this);
  // visit other places. . .
}
```

One consequence of Visitor is that the base class `Place` must declare `Accept()` methods for all derived `Transition` types. To preserve the framework-like design of `Place`, a `PlaceNodeVisitor` interface class defines empty `Accept()` methods. The `Place` class uses multiple inheritance to expose a modifiable `PlaceNodeVisitor` interface without requiring changes to `Place`'s definition.

Another consequence of Visitor is that `Transition` classes become stateful, essentially duplicating data from a `Place` to be used by `Accept()` calls to subsequent `Place`s. An alternative design might use runtime type information (RTTI) to access connected derived `Place`s with `dynamic_cast`:

```
void DerivedTransition::Execute() {
  DerivedPlace* p =
      dynamic_cast<DerivedPlace*>(
         arcInput_[1]->GetPlace() );
  assert(p != 0);
  // use DerivedPlace's methods. . .
}
```

```
int _tmain(int argc, _TCHAR* argv[]) {
  CPetriNet net;

  // places 0-5 are generic
  for(int k = 0; k < 6; ++k) {
    net.AddPlace(new CPlace);
  }

  // places 6, 7 are specific
  net.AddPlace(new CPlaceFile);
  net.AddPlace(new CPlacePrinter);

  // transitions 0-5
  net.AddTransition(new CTransitionGetFile);
  net.AddTransition(
               new CTransitionGetPrinter);
  net.AddTransition(new CTransitionPrint);
  net.AddTransition(
               new CTransitionGetPrinter);
  net.AddTransition(new CTransitionGetFile);
  net.AddTransition(new CTransitionPrint);

  // string describing all arc connections
  string strArcs =
    "p0t0t0p1p1t1t1p2p2t2t2p0p3t3t3p4p4t4" \
    "t4p5p5t5t5p3p6t0p6t4t2p6t5p6p7t1p7t3" \
    "t2p7t5p7;";
  net.MakeConnections(strArcs);

  // set initial marking
  net.GetPlace(0)->AddTokens(1);
  net.GetPlace(3)->AddTokens(1);
  net.GetPlace(6)->AddTokens(1);
  net.GetPlace(7)->AddTokens(1);

  // create two threads
  net.AddThread(new CThread);
  net.AddThread(new CThread);
  if(!net.Test()) {
    cout << "ERROR: deadlock state exists"
         << endl;
    return 1;
  }
  net.Start();

  // run demonstration for 30 seconds
  Sleep(30000);
  net.Stop();
  return 0;
}
```

Listing 1

The RTTI design does away with `PlaceNodeVisitor` and allows purely behavioural derived `Transition` classes to use multiple derived `Place`s within a single function body.

Both designs tie the identity of a specific resource to the order in which it is connected by the `MakeConnections()` function, determined by the ordering of the string describing the list of

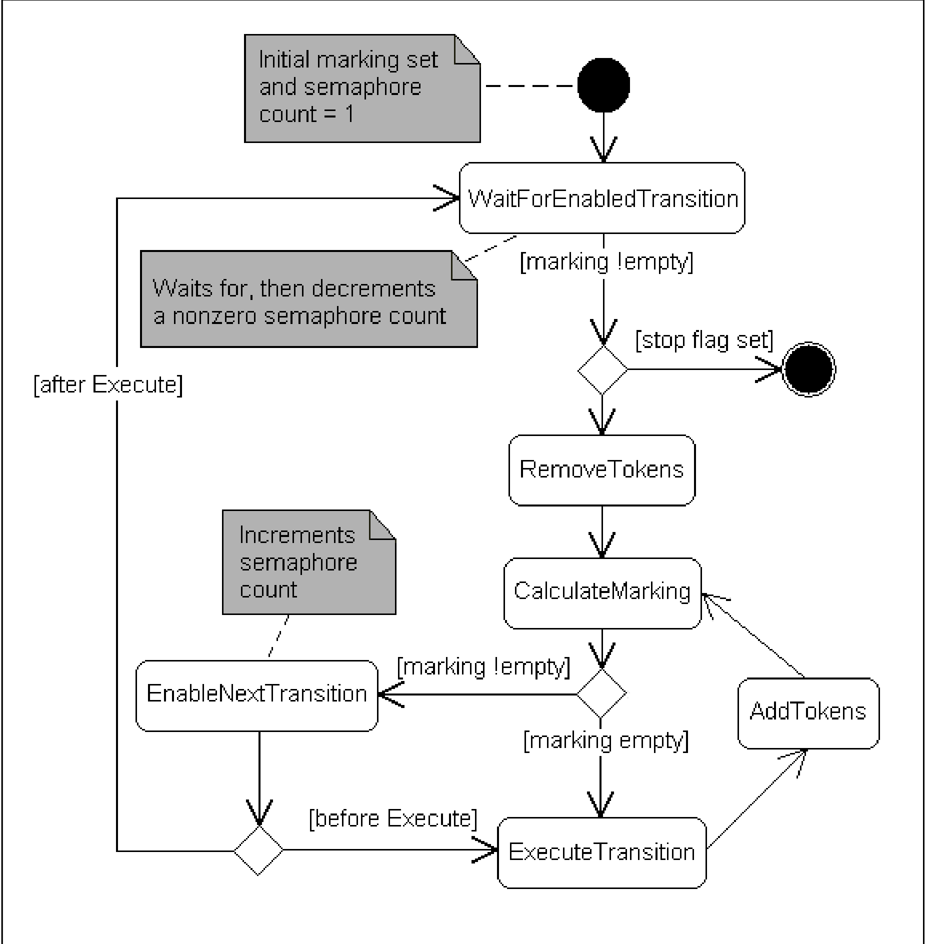

Figure 2

A new marking is calculated twice per loop, the first time after removing tokens from input places. This marking may be empty due to a conflict or because only one transition was enabled in the previous marking, but this does not produce deadlock. If the new marking calculated after removing tokens still has enabled transitions, the semaphore is released, enabling any waiting thread to process a remaining transition concurrently.

After executing the transition and adding tokens to output places, a new marking is calculated again. To prevent a live-lock, the order of the calculated marking is shuffled randomly so that the same transition in a conflicted pair is not picked first every time. If the marking is empty the semaphore is not released and the system is deadlocked.

Post-build Deadlock Testing

The `PetriNet::Test()` method builds the occurrence graph by calculating every marking reachable from the initial marking (without the random shuffle). Intermediate (post remove, pre add) markings are not considered here. If an empty marking is found the test fails and the function returns `false`. The test algorithm in pseudo-code looks like this:

connections. In the RTTI design, asserting that the cast pointer is not null is a good debug check.

The largest number of concurrently enabled transitions in any marking in the occurrence graph determines the maximum number of threads that can process the net. Presently this would be set during design, but a function could feasibly be written to create the appropriate number of threads at runtime.

Net Processing

Figure 2 is a state chart of `Thread` processing of the net. Processing is started by a call to `PetriNet::Start()`. This calculates the set of enabled transitions from the initial token placement and creates a Win32 semaphore with an initial count of 1 and a maximum count equal to the number of threads. A Win32 semaphore is a synchronization object that decrements its count for each thread it allows to pass and blocks threads while its count is zero. When the thread releases the semaphore its count is incremented.

Operations that change the internal state of a Petri net (e.g. adding or removing tokens) must be performed atomically. `PetriNet` contains a Win32 mutex for this purpose. A mutex allows a single thread to pass and blocks all other threads until it is released. Mutex is short for mutual exclusion. A thread can gain exclusive access to the net by creating a `ScopedLock` object. `ScopedLock`'s constructor accepts a `PetriNet` reference and is a `friend` class to `PetriNet`, so it can acquire a lock on `PetriNet`'s mutex. When the created lock object goes out of scope its destructor releases `PetriNet`'s mutex.

```
Calculate set of enabled transitions from
                 initial token placement.
If set is empty declare failure.
Name the initial set, count it as
          unvisited and add it to a list.
Call the initial set the current set.
While there are unvisited sets:
  Take the first unvisited transition in
                         the current set.
  Push the transition and the name of the
                current set onto a stack.
  Remove tokens from the places connected
              to the transition's inputs.
  Add tokens to the places connected to
              the transition's outputs.
  Mark this transition in this set
                                 visited.
  Calculate the new set.
  If set is empty declare failure.
  Else if set not in list:
    Name new set.
    Add it to the list.
    Mark it unvisited.
    Make it the current set.
  End
  If all transitions in the current set
                      have been visited:
    Declare the set visited.
```

```
      Undo the transition token move at the
                                 top of the stack.
      Make the set at the top of the stack
                                   the current set.
    End
  End
```

In the example application `Test()` is called prior to `Start()` to prevent a deadlocked net from running. `Test()` works without executing any of the net's transitions. A practical application could be executed with a command line switch that causes `main()` to call `Test()` and return an exit code. This could be performed as a post-build step in development. This feature would be helpful if the structure of the net or the number of resources were undergoing design.

Fixing the Demo Application

Suppose instead of a printer resource we constructed a print queue resource that took file submissions in an atomic operation. With this change to the resource we would obtain not an exclusive lock on a printer but a lock on a queue location. Adding a second token to the printer resource in the initial marking and building the occurrence graph proves that a two-position printer queue would prevent deadlock. The markings are as follows:

```
  M0 = { T0, T3 }
  M1 = { T1, T3 }
  M2 = { T2, T3 }
  M3 = { T2 }
  M4 = { T0, T4 } (conflict)
  M5 = { T1 }
  M6 = { T5 }
```

Figure 3 shows the output of the demo application with two initial printer tokens. The results of each marking calculation are printed as well, with the token placement following the set of enabled transitions.

Future Work

There is a lot of room for future improvement of the framework. A feature of Petri nets not implemented yet is the inhibitor arc, in which the presence of tokens in the connected place inhibits the connected transition. Concepts from higher-level Petri nets would add powerful functionality. For example, colored Petri nets allow tokens to have a type (or color) property. This enables the use of complex arc expressions involving type and quantity and makes possible decision branches as part of the net structure.

Conclusion

A multithreaded process designed using Petri net analysis might be deployed rapidly enough using this framework to justify the added runtime costs. For more information on Petri nets two references are listed below [1, 2].

David Nadle
david@nadle.com

References

[1] Marsan, M. Ajmone et al. 1995. *Modelling with Generalized Stochastic Petri Nets*. Chichester: John Wiley & Sons.
[2] Jensen, Kurt. 1996. *Colored Petri Nets, vol. 1*. 2nd ed. Berlin: Springer-Verlag.
[3] Gamma, Erich et al. 1995. *Design Patterns*. Reading: Addison-Wesley.

```
start
{ 0 3 } [ 1 0 0 1 0 0 1 2 ]
{ 0 } [ 1 0 0 0 0 0 1 1 ]
thread 3548 gets printer
{ } [ 0 0 0 0 0 0 0 1 ]
{ } [ 0 0 0 0 1 0 0 1 ]
thread 3648 gets file
{ 1 } [ 0 1 0 0 1 0 0 1 ]
{ } [ 0 0 0 0 1 0 0 0 ]
thread 3548 gets printer
{ 2 } [ 0 0 1 0 1 0 0 0 ]
{ } [ 0 0 0 0 1 0 0 0 ]
thread 3648 PRINTING: Hello World!
{ 0 4 } [ 1 0 0 0 1 0 1 1 ]
{ } [ 1 0 0 0 0 0 0 1 ]
thread 3548 gets file
{ 5 } [ 1 0 0 0 0 1 0 1 ]
{ } [ 1 0 0 0 0 0 0 1 ]
thread 3648 PRINTING: Hello World!
{ 3 0 } [ 1 0 0 1 0 0 1 2 ]
{ 3 } [ 0 0 0 1 0 0 0 2 ]
thread 3548 gets file
{ } [ 0 0 0 0 0 0 0 1 ]
thread 3648 gets printer
{ 1 } [ 0 1 0 0 0 0 0 1 ]
{ 1 } [ 0 1 0 0 1 0 0 1 ]
{ } [ 0 0 0 0 1 0 0 0 ]
thread 3548 gets printer
{ 2 } [ 0 0 1 0 1 0 0 0 ]
{ } [ 0 0 0 0 1 0 0 0 ]
thread 3648 PRINTING: Hello World!
{ 4 0 } [ 1 0 0 0 1 0 1 1 ]
{ } [ 0 0 0 0 1 0 0 1 ]
thread 3548 gets file
{ 1 } [ 0 1 0 0 1 0 0 1 ]
{ } [ 0 0 0 0 1 0 0 0 ]
thread 3648 gets printer
{ 2 } [ 0 0 1 0 1 0 0 0 ]
{ } [ 0 0 0 0 1 0 0 0 ]
thread 3548 PRINTING: Hello World!
{ 0 4 } [ 1 0 0 0 1 0 1 1 ]
{ } [ 1 0 0 0 0 0 0 1 ]
thread 3648 gets file
{ 5 } [ 1 0 0 0 0 1 0 1 ]
{ } [ 1 0 0 0 0 0 0 1 ]
thread 3548 PRINTING: Hello World!
{ 3 0 } [ 1 0 0 1 0 0 1 2 ]
. . .continues. . .
```

Figure 3

Implementing the Bridge pattern using templates with Microsoft Visual C++ 6.0

by Chris Main

I wonder whether you had a similar experience to me. You read with excitement Andrei Alexandrescu's Modern C++ Design [1] (the author's remark "Truly there is beauty in computer engineering" could be applied to his own book). Then you came up against Microsoft Visual C++ at your place of work and found you couldn't try much of it out. Do we have to be content with using all that fun template stuff on our home computers, where we get to choose the compiler, or can we use some of the techniques even with Visual C++?

The main limitation of Visual C++ 6.0 is its lack of support for partial specialisation of class templates. That eliminates using Alexandrescu's `TypeList`s, upon which a significant portion of his book depends. Another limitation is that it does not permit template parameters that are themselves class templates. This is less of a problem since the inferior alternative of member function templates is supported (though the implementation of such functions has to be placed in the class definition, "inline" style). Within the boundaries set by these limitations there are some valuable techniques available to us as I hope to demonstrate.

The example I have chosen is to implement the Bridge pattern [2] for a class by making the class of its implementation member variable a policy class, passed as a template parameter. This application of templates was mentioned by Nicolai Josuttis in his talk at the 2001 ACCU Conference (a talk described in C Vu as "solid but uninspiring". Well, I was inspired by it!). I am also indebted in what follows to Lois Goldthwaite's stimulating presentation of polymorphism using templates in Overload [3].

The example class

To be specific, I am going to develop a Blackjack `Hand` class, as suggested by Code Critique 14 [4]. For card names I will use a simple enum:

```
// card.h (include guard not shown)
namespace Blackjack {
  enum Card { ace, king, queen, jack, ten,
              nine, eight, seven, six,
              five, four, three, two };
}
```

I will also use an encapsulated lookup table of card values:

```
// card_values.h (include guard not shown)
#include <map>
#include <exception>
#include "card.h"

namespace Blackjack {
  class Card_Values {
  public:
    explicit Card_Values(unsigned int
                         ace_value = 1);
    // copy constructor, assignment, swap,
    // destructor not shown
    class Card_Has_No_Value
              : public std::exception {};
    unsigned int lookup(Card card) const
                 throw(Card_Has_No_Value);
  private:
    typedef std::map<Card, unsigned int>
                              Card_Lookup;
    Card_Lookup card_lookup;
  };
}
```

For a first pass, we build a non-template version of the `Hand` class:

```
// hand.h (include guard not shown)
#include "card.h"
#include <map>
#include <exception>

namespace Blackjack {
  class Hand {
  public:
    // default constructor, copy constructor,
    // assignment, swap and destructor not shown
    class Four_Of_That_Card_Already
              : public std::exception {};
    class Five_Cards_Already
              : public std::exception {};
    class Fewer_Than_Two_Cards
              : public std::exception {};
    Hand& add(Card card)
          throw(Four_Of_That_Card_Already,
                Five_Cards_Already);
    unsigned int value() const
          throw(Fewer_Than_Two_Cards);
  private:
    struct Card_Data {
      unsigned int count;
    };
    typedef std::map<Card, Card_Data>
                          Card_Container;
    Card_Container cards;
    unsigned int number_of_cards;
    class Accumulate_Card_Value;
  };
}

// hand.cpp
#include "hand.h"
#include "card_values.h"
#include <numeric>

namespace {
  const Blackjack::Card_Values&
                       get_card_values() {
    static const Blackjack::Card_Values
                             card_values;
    return card_values;
  }
}
```

```
class Blackjack::Hand::Accumulate_Card_Value {
  public:
    Accumulate_Card_Value(
            const Card_Values& card_values)
            : values(card_values) {}
    unsigned int operator()(
            unsigned int accumulated_value,
            const Card_Container::value_type&
                                  card_data) {
      return accumulated_value +=
             (card_data.second.count *
              values.lookup(card_data.first));
    }
  private:
    const Card_Values& values;
};

// default constructor, copy constructor,
// assignment, swap and destructor not shown

Blackjack::Hand::AddStatus
Blackjack::Hand::add(Card card)
           throw(Four_Of_That_Card_Already,
                 Five_Cards_Already) {
  if(number_of_cards == 5) {
    throw Five_Cards_Already();
  }
  Card_Data& card_data = cards[card];
  if(card_data.count == 4) {
    throw Four_Of_That_Card_Already();
  }
  ++card_data.count;
  ++number_of_cards;
  return *this;
}

unsigned int Blackjack::Hand::value() const
           throw(Fewer_Than_Two_Cards) {
  if(number_of_cards < 2) {
    throw Fewer_Than_Two_Cards();
  }
  unsigned int hand_value =
     std::accumulate(cards.begin(),
                     cards.end(),
                     0,
                     Accumulate_Card_Value(
                         get_card_values()));
  if( (hand_value < 12) &&
      (cards.find(ace) != cards.end()) ) {
    hand_value += 10;
  }
  return hand_value;
}
```

Converting the example class to a template

In the non-template version, the implementation of hand is hard coded to be a `std::map<Card, Card_Data>` together with a cached value `number_of_cards`. Our goal is to replace this implementation with a single member variable.The class of this member variable will be a template parameter. To reach this goal we need to identify all the places in the implementation of `Hand` that will depend upon that member variable. To simplify matters we make the reasonable assumption that the class of the member variable will always have the properties of an STL container, i.e. we can assume things like `size()` and iterators are always available. In `add()` we additionally need to obtain non-const references to the count for a given card and to the total number of cards in the hand. In `value()` we need to determine the total number of cards in the hand and whether the container contains an ace. We can turn these four requirements into helper function templates. Function templates are particularly useful because of the way C++ deduces the instantiation required from the function arguments, avoiding the need for explicit instantiations.

```
// hand_implementation.h (include guard not
// shown)
#include "card.h"

namespace Blackjack {
  // Assume that Card_Container is usually a
  // std::map   If something else is used,
  // e.g. std::vector, these function
  // templates would need to be specialised to
  // use std::find

  template< class Card_Container >
  unsigned int& hand_implementation_count(
            Card card,
            Card_Container& card_container) {
    return card_container[card].count;
  }

  template< class Card_Container >
  typename Card_Container::const_iterator
       hand_implementation_find(
           Card card,
           const Card_Container&
                           card_container) {
    return card_container.find(card);
  }

  // non-const reference version of
  // number_of_cards
  template< class Card_Container >
  unsigned int&
      hand_implementation_number_of_cards
         (Card_Container& card_container) {
    return card_container.number_of_cards;
  }

  // const version of number_of_cards
  template< class Card_Container >
  unsigned int
      hand_implementation_number_of_cards
         (const Card_Container&
                          card_container) {
    return card_container.number_of_cards;
  }
}
```

Less obviously, perhaps, `Accumulate_Card_Value` depends upon the type of values contained by the container, so is indirectly dependent upon the container. We therefore make it a nested class of the container like so:

```
// card_count_container.h (include guard not
// shown)
#include <map>
#include "card.h"
#include "card_values.h"

namespace Blackjack {
  struct Card_Count {
    unsigned int count;
    Card_Count() : count(0) {}
  };

  class Card_Count_Container
        : public std::map<Card, Card_Count> {
  public:
    unsigned int number_of_cards;
                // cached value as before
    class Accumulate_Card_Value {
      // same as before
    };
  };
}
```

We are now in a position to re-implement `Hand` as a class template. I follow the convention described by Dietmar Kuehl at the 2002 ACCU Conference of placing the implementation of the class template in a `.tpp` file. For this article I am going to put all the instantiations in a `.cpp` file in which this `.tpp` file is included. The alternative is to let clients include the `.tpp` file and perform the instantiations themselves. The `.tpp` file serves to keep both of these alternatives available to us.

```
// hand_type.h (include guard not shown)
#include "card.h"
#include "card_count_container.h"

namespace Blackjack {

  // Exceptions moved out of class into
  // namespace so that all instantiations
  // can share the exceptions
  class Four_Of_That_Card_Already
                : public std::exception {};
  class Five_Cards_Already
                : public std::exception {};
  class Fewer_Than_Two_Cards
                : public std::exception {};

  template< class Card_Container >
  class Hand_Type {
  public:
    // identical to public section of
    // non-template Hand class except
    // for exceptions as noted above
  protected:
    // get_card_values() is moved here in
    // case we want to provide the
    // implementation of this class template
    // in a header file
    const Card_Values& get_card_values() const;

  private:
    Card_Container cards;
  };

  // declare the valid instantiations
  typedef Hand_Type<Card_Count_Container>
                      Hand_No_Cached_Values;
}

// hand_type.tpp
#include "card_values.h"
#include "hand_implementation.h"
#include <numeric>

// default constructor, copy constructor,
// assignment, swap and destructor not
// shown

template< class Card_Container >
Blackjack::Hand_Type<Card_Container>::Hand_Type&
Blackjack::Hand_Type<Card_Container>::add(
                                  Card card)
        throw(Four_Of_That_Card_Already,
              Five_Cards_Already) {
  // use the helper to access the number of
  // cards
  unsigned int& number_of_cards =
      hand_implementation_number_of_cards(
                                     cards);
  if(number_of_cards == 5) {
    throw Five_Cards_Already();
  }
  // use the helper to access the count
  unsigned int& count =
      hand_implementation_count(card, cards);
  if(count == 4) {
    throw Four_Of_That_Card_Already();
  }
  ++count;
  ++number_of_cards;
  return *this;
}

template< class Card_Container >
unsigned int Blackjack::Hand_Type<
           Card_Container>::value() const
        throw(Fewer_Than_Two_Cards) {
  // use the helper to check the number of
  // cards
  if(hand_implementation_number_of_cards(
                             cards) < 2) {
    throw Fewer_Than_Two_Cards();
  }
  unsigned int hand_value =
    std::accumulate(
        cards.begin(),
        cards.end(),
        0,
        typename Card_Container::
              Accumulate_Card_Value(
                          get_card_values()));
  if( (hand_value < 12) &&
      // use the helper to check for an ace
      (hand_implementation_find(ace, cards)
              != cards.end()) ) {
    hand_value += 10;
  }
  return hand_value;
}

template< class Card_Container >
const Blackjack::Card_Values&
    Blackjack::Hand<Card_Container>::
                     get_card_values() const {
  static const Card_Values card_values;
  return card_values;
}

// hand_type.cpp
#include "hand_type.h"
#include "hand_type.tpp"
#include "card_count_container.h"

// instantiate the valid instantiations
template
Blackjack::Hand_Type<
             Blackjack::Card_Count_Container>;
```

Creating a different instantiation of the class template

So far so good, but we only have one instantiation at the moment. It may appear, therefore, that we haven't gained very much. In fact we have significantly improved the testability of the `Hand` class, a point to which I will return later.

Let's try and build a variant of `Hand` that looks up the value of a card when it is added to the hand, and caches that value. For this we need an additional helper function template that will set the value for a card. We need to specialise this new function template to do nothing when there is nowhere to cache the value (which is the case for our first instantiation). Hence:

```
// hand_implementation.h (include guard not
// shown)
#include "card.h"
#include "card_count_container.h"

namespace Blackjack {
  // hand_implementation_count(),
  // hand_implementation_find()
  // and
  // hand_implementation_number_of_cards()
  // as before
  template< class Card_Container >
  void hand_implementation_set_value(
              Card card,
              unsigned int value,
              Card_Container& card_container) {
    card_container[card].value = value;
  }
  // specialisation to do nothing when there
  // is nowhere to cache the value
  template<>
  void hand_implementation_set_value(
         Card card,
         unsigned int value,
         Card_Count_Container& card_container);
}

// hand_implementation.cpp
#include "hand_implementation.h"

template <>
void Blackjack::hand_implementation_set_value
    (Card card,
     unsigned int value,
     Card_Count_Container& card_container) {}
```

The container class is:

```
// card_count_with_value_container.h
// (include guard not shown)
#include "card_count_container.h"

namespace Blackjack {
  struct Card_Count_With_Value
             : public Card_Count {
    unsigned int value;
    Card_Count_With_Value() : value(0){}
  };
  class Card_Count_With_Value_Container
      : public std::map<Card,
                    Card_Count_With_Value> {
  public:
    unsigned int number_of_cards; // as before
    class Accumulate_Card_Value {
    public:
      Accumulate_Card_Value(
           const Card_Values& /* unused */) {}
      unsigned int operator()
        (unsigned int accumulated_value,
         const std::map<Card,
          Card_Count_With_Value>::value_type&
                                 card_data) {
        return accumulated_value +=
             (card_data.second.count *
              card_data.second.value);
      }
    };
  };
}
```

Notice that we have adjusted `Accumulate_Card_Value operator()` to take advantage of the cached values; the signature of its constructor is preserved, even though the `Card_Values` argument is unused, so that it will work with the Hand_Type we have already. We then modify Hand_Type::add() to call hand_implementation_set_value() at the appropriate point:

```
template< class Card_Container >
Blackjack::Hand_Type<Card_Container>::Hand_Type&
  Blackjack::Hand_Type<Card_Container>::add(
                                  Card card)
       throw(Four_Of_That_Card_Already,
             Five_Cards_Already) {
  unsigned int& number_of_cards =
    hand_implementation_number_of_cards(cards);
  if(number_of_cards == 5) {
    throw Five_Cards_Already();
  }
  unsigned int& count =
       hand_implementation_count(card, cards);
  if(count == 4) {
    throw Four_Of_That_Card_Already();
  }
  if(count == 0) {
    // use the helper to cache the card value
    hand_implementation_set_value(card,
               get_card_values().lookup(card),
               cards);
  }
  ++count;
  ++number_of_cards;
  return *this;
}
```

We add the new instantiations:

```
// in hand_type.h
typedef
    Hand_Type<Card_Count_With_Value_Container>
    Hand_With_Cached_Values;
// in hand_type.cpp
template Blackjack::Hand_Type<
  Blackjack::Card_Count_With_Value_Container>;
```

I have used long names for the instantiation typedefs in an attempt to document the characteristics of each variant. I am assuming that any particular client is likely to want only one variant, and will further typedef the variant required to a shorter name like Hand. (I have used Hand_Type for the class template so that the simple name Hand is available for clients to use).

Making the implementation member variable a pointer

The classic Bridge pattern, of course, has the implementation member variable as a pointer. This allows us to change the implementation without forcing clients to recompile. We cannot simply instantiate our existing Hand_Type with a pointer because its implementation does not dereference the member variable cards. The ideal solution would be to partially specialise Hand_Type for all instantiations taking a pointer as the template parameter, but this is not supported by Visual C++ 6.0. The workaround is to define another class template which I will name Hand_Bridge.

Bjarne Stroustrup [5] writes that this was the approach he tried before deciding upon partial specialisation. He comments that he abandoned it because even good programmers forgot to use the templates designed to be instantiated with pointers. That problem does not arise in our case, though, because the compiler prevents us from instantiating Hand_Type with a pointer.

For this class template I will assume the existence of a suitable smart pointer (deep copy is appropriate - see Alexandrescu [1] for a full discussion of smart pointers):

```
// hand_bridge.h (include guard not shown)
#include "card.h"
#include "smart_pointer.h"

namespace Blackjack {
  template< class Card_Container >
  class Hand_Bridge {
  public:
    // identical to public section of
    // Hand_Type class
  private:
    Smart_Pointer<Card_Container> cards;
  };
  // Forward declaration is now sufficient in
  // the header
  class Card_Container_Implementation;
  // declare the valid instantiation
  typedef Hand_Bridge<
        Card_Container_Implementation> Hand;
}

// hand_bridge.tpp
#include "card_values.h"
#include "hand_implementation.h"
#include <numeric>
// default constructor, copy constructor,
// assignment, swap and destructor not shown
template< class Card_Container >
Blackjack::Hand_Bridge<Card_Container>::
         Hand_Bridge& Blackjack::Hand_Bridge<
         Card_Container>::add(Card card)
         throw(Four_Of_That_Card_Already,
               Five_Cards_Already) {
  unsigned int& number_of_cards =
      hand_implementation_number_of_cards(
                     cards->implementation());
  if(number_of_cards == 5) {
    throw Five_Cards_Already();
  }
  unsigned int& count = hand_implementation_count(
          card, cards->implementation());
  if(count == 4) {
    throw Four_Of_That_Card_Already();
  }
  if(count == 0) {
    hand_implementation_set_value(card,
               get_card_values().lookup(card),
               cards->implementation());
  }
  ++count;
  ++number_of_cards;
  return *this;
}
```

```
template< class Card_Container >
unsigned int Blackjack::Hand_Bridge<
               Card_Container>::value() const
               throw (Fewer_Than_Two_Cards) {
  if(hand_implementation_number_of_cards(
         cards->const_implementation()) < 2) {
    throw Fewer_Than_Two_Cards();
  }
  unsigned int hand_value =
     std::accumulate(cards->begin(),
        cards->end(), 0, typename
        Card_Container::Accumulate_Card_Value(
                          get_card_values()));
  if( (hand_value < 12) &&
      (hand_implementation_find(ace,
           cards->const_implementation()) !=
                              cards->end()) ) {
    hand_value += 10;
  }
  return hand_value;
}

//hand_bridge.cpp
#include "hand_bridge.h"
#include "hand_bridge.tpp"
#include "card_count_container.h"
#include "card_count_with_value_container.h"

namespace {
  template< class Card_Container >
  class Implementation : public Card_Container {
  public:
    Card_Container& implementation() {
      return *this;
    }
    const Card_Container&
                const_implementation() const {
      return *this;
    }
  };
}

#ifdef NO_CACHED_CARD_VALUES

class Blackjack::Card_Container_Implementation
    : public Implementation<
        Blackjack::Card_Count_Container> {};
#else // use the implementation with cached
      // card values
class Blackjack::Card_Container_Implementation
    : public Implementation<Blackjack::
        Card_Count_With_Value_Container> {};
#endif // NO_CACHED_CARD_VALUES

// instantiate the valid instantiation template
Blackjack::Hand_Bridge<
    Blackjack::Card_Container_Implementation>;
```

We can switch the implementation of `Hand` simply by recompiling `hand_bridge.cpp` with or without `NO_CACHED_CARD_VALUES` being defined; clients of `Hand` do not need to be recompiled.

You will no doubt be wondering why I have found it necessary to derive `Card_Container_Implementation` from the helper class template `Implementation`. The reason is so that the specialisations of the helper function templates are used. Take for example `hand_implementation_set_value()`. This has a specialisation for `Card_Count_Container`. If we simply called `hand_implementation_set_value(*cards)` the type of `*cards` is `Card_Container_Implementation` for which there is no specialisation (nor can there be since this is the class that changes depending upon our compilation settings); the compiler tries to use the unspecialised function template and fails to compile if `NO_CACHED_CARD_VALUES` is defined. In contrast, when we call `hand_implementation_set_value(cards->implementation())` the type of `cards->implementation()` is `Card_Count_Container` (if `NO_CACHED_CARD_VALUES` is defined) and the compiler uses the specialised function template as required.

Testing using the Dependency Inversion Principle revisited

Implementing the Bridge pattern in the way I have described has the benefit of facilitating unit testing. It realises the Dependency Inversion Principle [6] because classes depend upon other classes only via template parameters. It is therefore easy to use stub versions of classes that a class under test depends upon: the class under test is simply instantiated with the stub. I believe that this realisation of the Dependency Inversion Principle by means of templates is an improvement upon the realisation using abstract base classes [7] for a number of reasons:

- we are not limited to using classes derived from specific abstract base classes (specialisation allows us to get round this limitation)
- we avoid the overhead of virtual function tables
- we can even avoid using pointers altogether (if the recompilation cost incurred is a relatively insignificant factor);
- we avoid the need for supporting class factories (typedefs are sufficient).

In conclusion, the lack of support for partial specialisation in Visual C++ 6.0 is a serious inconvenience. However we should not underestimate the power and usefulness of full specialisation which it does support.

Chris Main

References

[1] Andrei Alexandrescu, *Modern C++ Design*, Addison Wesley C++ In Depth Series, 2001

[2] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995

[3] Lois Goldthwaite, “Programming With Interfaces In C++: A New Approach,” *Overload 40* (December 2000)

[4] “Student Code Critique 14,” *C Vu 14.1* (February 2002)

[5] Bjarne Stroustrup, *The C++ Programming Language*, 3rd Edition, Addison Wesley, 1997

[6] `http://www.objectmentor.com/resources/articles/dip.pdf`

[7] Chris Main, “OOD and Testing using the Dependency Inversion Principle,” *C Vu 12.6* (December 2000)

EXPR_TYPE – An Implementation of typeof Using Current Standard C++

by Anthony Williams

typeof is a much-sought-after facility that is lacking from current C++; it is the ability to declare a variable to have the same type as the result of a given expression, or make a function have the same type as an expression. The general idea is that typeof(some-expression) would be usable anywhere a type name could normally be used. This article describes a way of providing this ability within the realms of current C++.

Introduction

Imagine you're writing a template function that adds its parameters, such as the following:

```
template<typename T,typename U>
SomeResultType addValues(const T&t,
                         const U&u) {
  return t+u;
}
```

What type should the return value have? Obviously, the ideal choice would be "the type of t+u", but how can the compiler determine that type? The rules for the type of an additive expression involving only builtin types are not trivial – consider pointer arithmetic, integral type conversions, and integer conversion to floating-point type – so how *do* we do it? The Standard C++ Library only deals with the case where both function parameters have the same type, and forces the result to be the same type – e.g. std::plus, and std::min. Another possible solution is to hard code the rules for this particular operation, as Andrei Alexandrescu does for min and max [1].

The solution provided by some compilers as an extension, and widely considered to be a prime candidate for inclusion in the next version of the C++ standard is the typeof operator. The idea of typeof is that it takes an expression, and determines its type at compile time, without evaluating it, in much the same way as sizeof determines the size of the result of an expression without evaluating it. typeof could then be used anywhere a normal type name could be used, so we could declare addValues as:

```
template<typename T,typename U>
typeof(T()+U()) addValues(const T&t,
                          const U&u);
```

The function parameters t and u aren't in scope at the point of declaration of addValues, so we use the default constructors for the types to generate values to add – the values used are unimportant, as the expression is never evaluated[1].

This is good – the return type of addValues is now correct for all combinations of types, but we now have the problem that we've used a non-standard extension, so the code is not portable. One compiler may use the name typeof, another __typeof__, and a third may not provide the extension at all. We therefore need a portable solution, that uses only Standard C++.

1 There are better ways to generate suitable expressions, that don't rely on the type having a default constructor, but they will be dealt with later

Current Facilities

What facilities are there in Standard C++ that we can use? Firstly, we need a means of obtaining information about an expression without actually evaluating it. The only Standard facility for this is sizeof[2]. We therefore need a mechanism to ensure that expressions of different types yield different values for sizeof, so we can tell them apart, and we need a mechanism for converting a size into a type.

Another Standard C++ facility we can use to obtain information about a type is *function template argument type deduction*. By writing a function template which has a return type dependent on the argument type, we can encode information from the argument type into the return type any way we choose – given:

```
template<typename T>
struct TypeToHelperInfo{};

template<typename T>
TypeToHelperInfo<T> typeOfHelper(const
                                   T& t);
```

we can then write the TypeToHelperInfo class template to provide any necessary information.

Bringing it Together

Converting a value into a type is easy – just create a class template with a value template parameter to accept the value which the type has been assigned. Then, specialize this template for each value/type combination, as in listing 1 – Size1 and Size2 are constants, and are the unique size values that relate to Type1 and Type2, respectively, rather than sizeof(Type1) or sizeof(Type2).

```
template<std::size_t size>
struct SizeToType {};

template<>
struct SizeToType<Size1> {
  typedef Type1 Type;
};

template<>
struct SizeToType<Size2> {
  typedef Type2 Type;
};
```

Listing 1: Converting a size to a type

Converting a type into a size value is a bit more complex. If we have an expression expr, of type T, then typeOfHelper(expr) is of type TypeToHelperInfo<T>, if we use the signature of typeOfHelper from above. We can then specialize TypeToHelperInfo, so it has a distinct size for each distinct type. Unfortunately, it is not that simple – the compiler is free to add padding to structs to ensure they get aligned properly, so we cannot portably control the size of a struct. The only construct

2 For non-polymorphic types, typeid also identifies the type of its operand at compile-time, without evaluating it, but the return type of typeid is very little use, as it is not guaranteed to have the same value for the same type, just an equivalent value.

in C++ which has a precisely-defined size is an array, the size of which is the number of elements multiplied by the size of each element. Given that `sizeof(char)==1`, the size of an array of `char` is equal to the number of elements in that array, which is precisely what we need. We can now specialize `TypeToHelperInfo` for each type, to contain an appropriately-sized array of `char`, as in listing 2.

```
template<>
struct TypeToHelperInfo<Type1> {
  char array[Size1];
};

template<>
struct TypeToHelperInfo<Type2> {
  char array[Size2];
};
```

Listing 2: Getting appropriately-sized arrays for each type

We can now simulate `typeof(expr)` with `SizeToType<sizeof(typeOfHelper(expr).array)>::Type`. In templates, we probably need to precede this with `typename`, in case `expr` depends on the template parameters. To ease the use, we can define an `EXPR_TYPE` macro that does this for us:

```
#define EXPR_TYPE(expr)
SizeToType<sizeof(
         typeOfHelper(expr).array)>::Type
```

We also need to declare the appropriate specializations of `SizeToType` and `TypeToHelperInfo` for each type we wish to detect, so we define a `REGISTER_EXPR_TYPE` macro to assist with this, as in listing 3.

```
#define REGISTER_EXPR_TYPE(type,value)\
template<>\
struct TypeToHelperInfo<type>{\
  char array[value];\
};\
template<>\
struct SizeToType<value>{\
  typedef type Type;\
};
```

Listing 3: The `REGISTER_EXPR_TYPE` macro.

We can then declare the necessary specializations for all the basic types, and pointers to them, so our users don't have to do this themselves.

`const` Qualification

As it stands, with only the one `typeOfHelper` function template, `const` qualifications are lost. This may not be a problem, as `const`-qualification doesn't always have much significance with *value* types. However, this is a problem we can overcome[3] by providing two overloads of `typeOfHelper` instead of just the one:

```
template<typename T>
TypeToHelperInfo<T> typeOfHelper(T& t);
template<typename T>
TypeToHelperInfo<const T>
typeOfHelper(const T& t);
```

We can then specialize the class templates for each distinct cv-qualified type – most easily done by modifying the `REGISTER_EXPR_TYPE` macro to register all four cv-qualified variants of each type with distinct values. Note that volatile qualification is automatically picked up correctly, because `T` will then be deduced to be "`volatile X`" for the appropriate type `X`. We only need these distinct overloads to allow the use of `EXPR_TYPE` with expressions that return a non-`const` temporary, since with only a single function taking a `T&`, `T` is deduced to be the non-`const` type, and temporaries cannot bind to non-`const` references. With both overloads, the temporary can be bound to the overload that takes `const T&`. The result is that temporaries are deduced to be `const`[4].

In the final implementation, all the classes and functions are in namespace `ExprType`, to avoid polluting the global namespace, and the macro definitions have been adjusted accordingly.

Restrictions

The most obvious restriction is that this only works for expressions that have a type for which we have specialized `SizeToType` and `TypeToHelperInfo`. This has the consequence that we cannot define a specialization for `std::vector` in general; we have to define one specialization for `std::vector<int>`, and another for `std::vector<double>`. Also, in order to avoid violating the One Definition Rule, the user must ensure that the same value is used for the same type in all translation units that are linked to produce a single program. This includes any libraries used, so when writing library code that uses `EXPR_TYPE`, it is probably best to put the templates in a private namespace, to isolate them from the rest of the program, and avoid the problem.

Also, `EXPR_TYPE` cannot tell the difference between an *lvalue* and an *rvalue*, or between a reference and a value, except that *rvalues* are always const, whereas *lvalues* may not be – given:

```
int f1();
int& f2();
const int& f3();
int i;
```

`EXPR_TYPE(f2())` and `EXPR_TYPE(i)` are `int`, whereas `EXPR_TYPE(f1())`, `EXPR_TYPE(f3())` and `EXPR_TYPE(25)` are `const int`. The reason for this is that the mechanism used for type deduction – function template argument type deduction – can only distinguish by cv-qualification, and *rvalues* are mapped to `const` references. This means you cannot use `EXPR_TYPE` to pass references – you must explicitly add the & where needed, though this can be done automatically for non-`const` references. It also means that you may have to strip the `const` if the expression results in an

3 for compilers that support partial ordering of function templates.

4 Some compilers that don't support partial ordering of function templates, also allow the binding of temporaries to non-`const` references, so we only need supply a single function, taking `T&`, in which case temporaries are deduced to be non-`const`.

rvalue, and you wish to declare a non-`const` variable of that type, using something like `boost::remove_const` [3].

Finally, `EXPR_TYPE` cannot be used for expressions with `void` type, such as functions returning `void`. This is because, though references to incomplete types are permitted in general, references to `void` are explicitly not permitted.

Revisiting the Example

To come back to the example from the introduction, we can now implement our `addValues` template as:

```
template<typename T,typename U>
typename EXPR_TYPE(T()+U())
addValues(const T& t,const U& u) {
  return t+u;
}
```

However, this still relies on the types `T` and `U` having default constructors. We can avoid this restriction by declaring a `makeT` function template:

```
template<typename T>
T makeT();
```

and then using this template in the parameter for `EXPR_TYPE` – `EXPR_TYPE(makeT<const T&>()+makeT<const U&>())`. This is a useful technique, whenever one is writing such an expression, where all we want is its type, or size – since the expressions using `makeT` are never evaluated, there is no need to provide a definition of `makeT`; consequently using `makeT` imposes no restrictions on the types.

Further Examples

This technique is useful in any scenario where you wish to declare an object of a type related to the type of something else, e.g. declaring a pointer to another variable

```
int i;
EXPR_TYPE(i) * j=&i; // j is "int *"
```

or supporting containers that may or may not declare `iterator` and `const_iterator typedef`s.

```
template<typename Container>
void func(const Container& container) {
  for(typename EXPR_TYPE(container.begin())
                  it = container.begin();
      it != container.end(); ++it) {
    // do something
  }
}
```

It can also be used for such things as implementing `min` and `max` function templates:

```
template<typename T,typename U>
struct MinHelper {
  typedef typename EXPR_TYPE(
    (makeT<const T&>()<makeT<const U&>())
     ? makeT<const T&>()
     : makeT<const U&>()) Type;
};

template<typename T,typename U>
typename MinHelper<T,U>::Type min(const
T& t,const U& u) {
  return (t<u)?t:u;
}
```

In general, it is of most use in template code, with expressions where the type depends on the template parameters in a non-trivial fashion. This allows the writing of more generic templates, without complex support machinery.

It must be noted, however, that the user must ensure that all types to be deduced have been registered in advance. The library can facilitate this by declaring all the builtin types, and some compound types (e.g. `char*`, `const double*`, etc.), but class types and more obscure compound types (such as `char volatile**const*`) must be declared explicitly.

Summary

The macros and templates that make up the `EXPR_TYPE` library enable a `typeof`-like mechanism using Standard C++. The only cost is the maintenance burden placed on the user, to ensure there is a one-to-one correspondence between types and size-values across each project, and the only restriction is that it cannot tell the difference between lvalues and rvalues, and cannot detect `void`, though all cv-qualifications are identified.

This powerful mechanism is another tool in the Generic Programming toolbox, enabling people to write generic code more easily.

The code presented here will be available with the online version of the article at :
`http://cplusplus.anthonyw.cjb.net/articles.html`.

Anthony Williams

References

[1] Andrei Alexandrescu, “Generic<Programming>: Min and Max Redivivus.” *C/C++ Users Journal*, 19(4), April 2001. Available online at
`http://www.cuj.com/experts/1904/alexandr.htm`.

[2] Bill Gibbons, “A Portable “typeof” operator.” *C/C++ Users Journal*, 18(11), November 2000.

[3] The Boost Team. Boost C++ libraries. See
`http://www.boost.org`

Exported Templates

by Jean-Marc Bourguet

Exported templates are one of the two standard template compilation models. Exported template implementations have only recently become available. During the time they where defined but unavailable, they were the subject of much expectation, some of it unreasonable and some of it the consequence of confounding compilation models and instantiation mechanisms.

This article reviews the compilation models and instantiation mechanisms defined by the C++ standard, and then looks at some related issues with C++ templates and examines what we can expect from the `export` keyword.

Template Compilation Models

According to [1],

> *The* compilation model *determines the meaning of a template at various stages of the translation of a program. In particular, it determines what the various constructs in a template mean when it is instantiated. Name lookup is an essential ingredient of the compilation model.*

Name lookup is an essential ingredient of the compilation model, but the standard models share the same name lookup rules, which is called two phase lookup, so it is sufficient for this article to state that names independent of the template parameters are searched in the context of the definition of the template (that means that only names visible from the definition are found) while names dependent on the template parameters are searched in both the definition and instantiation contexts (that means that names visible from the place where the instantiation is used may also be found). [1] provides a more complete description, including a more precise definition of what the definition and instantiation contexts are.

Note that while these name lookup rules were introduced in the draft standard in 1993, until 1997 all compilers looked up both dependent and independent names only in the instantiation context; the first compilers to correctly implement the name lookup rules found so many errors in programs that they had to output warnings instead of errors.

At the heart of this article is another ingredient of the template compilation model: how the definitions of non class templates are found. At first, this part of the compilation model was not clearly specified. For example, Bjarne Stroustrup [2] wrote:

> *When a function definition is needed for a class template member function for a particular type, it is the implementation's job to find the template for the member function and generate the appropriate version. An implementation may require the programmer to help find template source by following some convention.*

CFront, which was the first implementation of C++ templates, used some conventions which are described in the appendix. However, the standard provides two ways, both of which differ from the CFront approach.

The Inclusion Compilation Model

This is the only commonly provided compilation model: the definition of a template has to be provided in the compilation unit where the template is instantiated[1].

In an effort to be able to compile source code written for CFront, some compilers provide a variant where the source file which would be used by CFront is automatically included when needed.

1 While the standard requires that the definition is either available in every compilation unit where the template is instantiated or exported, no compilers I know of check this rule, they all simply don't instantiate a template in a compilation unit where the definition is not present, and some take advantage of this behaviour.

The Separation Compilation Model

When using this model, template declarations have to be signalled as exported[2] (using the export keyword). A definition of the template has to be compiled in one (and only one, by the one definition rule) compilation unit and the implementation has to manage with that.

It should be noted that while in the inclusion model the two contexts where names are looked up are usually not very different (but remember the problems found by the first implementation of two phase name lookup rules), in this model the differences may be far more important and give birth to some surprising consequences, especially in combination with overloading and implicit conversions.

Template Instantiation Mechanisms

According to [1][3],

> *The* instantiation mechanisms *are the external mechanisms that allow C++ implementations to create instantiations correctly. These mechanisms may be constrained by the requirements of the linker and other software building tools.*

One may consider there to be two kinds of instantiation mechanisms:

- **local**, where all the instantiations are done when considering each compilation unit, and
- **global**, where the instantiations are done when considering all the compilation units in the program or library.

CFront used a global mechanism: it tried a link and used the error messages describing missing symbols to deduce the necessary template instantiations. It then generated them and retried the link until all required instantiations where found.

Borland's compiler introduced the local mechanism: it added all the instantiations to every object file and relied on the linker to remove any duplicates.

Sun's compiler also uses a local mechanism. It also generates all the needed instantiations when compiling a compilation unit, but instead of putting them in the object file, it puts them in a repository. The linker does not need to be able to remove duplicates and an obvious optimisation is generating an instantiation only if it is not already present in the repository.

Comeau's and HP's compilers have a global mechanism: they use a pre-linker to detect the needed instantiations[4]. These are then assigned to compilation units which can generate them and these compilation units are recompiled. The assignment is cached so that when recompiling a compilation unit to which instantiations have been assigned, they are also regenerated; the pre-linking phase is then usually simply a check that all the needed instantiations are provided, except when starting a compilation from a clean state.

Comeau's compiler has an additional mode where several objects are generated from a compilation unit to which instantiations have been assigned. This removes the need to link a compilation unit (and other compilation units upon which it depends) only because an instantiation has been assigned to it.

2 The standard seems to imply that only the definition has to be marked as exported, but the only implementation demands that the declaration is marked and a defect report (the mechanism to report and correct bugs in the standard) on this issue has been introduced to demand it.

3 The classification of instantiation mechanisms used in this book is different to the one presented here. They use *greedy instantiation* and *queried instantiation* for what we call *local instantiation* and use *iterated instantiation* class for *global instantiation.*

4 Unlike CFront, they do not detect them by examining the missing symbols from a failed link, but use a more efficient mechanism.

Issues related to template instantiations

It should be noted that the compilation model and the instantiation mechanisms are mostly independent: it is possible (though not always convenient or especially useful) to implement the standard compilation models with each of the instantiation mechanisms described. A consequence is that one should not expect the separation compilation model to solve problems related to instantiation mechanisms.

Publishing the source code

The need in the inclusion model to provide the source code of the template definition is seen by some as a problem. Is the separation model a solution?

First, it should be noted that the standard formally ignores such issues and so a compiler could always demand that the source code be present until link time. Not going to such extremes, some compilers delay code generation until link time and so generate *high level* object files[5].

It should also be noted that a compiler could provide a way to accept encrypted source or high level intermediate format (something very similar to what is done with precompiled headers) and so if there is enough demand, the compiler makers can provide a solution (not perfect but probably good enough for most purposes: it is used in other languages; the main problem would probably be to generate useful error messages when encrypted code is all what is available) even with the inclusion model.

These remarks made, we'll consider the related but quite different question: can the separation model be implemented in such a way that only low level information is needed to instantiate templates?

The two phase lookup rule and other modifications made during the standardisation process allowed compilers to check the template definition for syntactic errors, but most semantic ones can only be detected at instantiation time. Indeed, most operations done in a template depend on the template parameters, and so the parameters must be known to get the precise meaning.

So, obviously the answer is no: the separation model may not prevent the need to furnish the template definition as a high level description.

Compilation time

Templates are often blamed for long compilation times. Is this attribution correct?

Concerning the compilation model, in the inclusion model, every compilation unit using a template has to include the definition of the template and so everything needed for the definition. So more code has to be read and parsed than for the export model, but some techniques such as precompiled headers can reduce the overhead.

The separation model does not have obvious overhead in forcing redundant work to be done, even if the current implementations force a reparsing of the definition for each instantiation.

The main overhead of the global instantiation mechanisms is in the way the required instantiations are detected. CFront's way of trying links until closure was costly. More modern methods such as those of HP and Comeau are less costly. But, they still have the disadvantage of increasing the link time of a clean build. The global mechanisms have also an overhead in the recompilation of the compilation units to which instantiations have been assigned. Although, this overhead exists only when doing a clean build.

There is a serious overhead in the local mechanism without using a repository: the instantiations are compiled several times, and the optimisation and code generation phases of a compiler usually do take a significant part of the process. Doing so only to throw the result away is a waste, and bigger files are created and it complicates and slows down the linker.

Recompilation

In this section, we'll examine what recompilations are needed when a file is modified, and if the recompilation can be done automatically.

When modifying a type used as a template argument, all the files using this type should be recompiled and the compilation model has no influence on that.

The normal use of makefiles[6] triggers the recompilation in all combinations of compilation models and instantiation mechanisms.

When modifying a template definition, things are sensibly different.

With the inclusion model, the normal use of makefiles triggers a recompilation of all compilation units including the definition of the template and so the needed instantiation will be recompiled whatever the compilation model is used.

With the separation model, the normal use of makefiles will trigger a recompilation of the compilation unit providing the exported definition and trigger a relinking. Is this enough?

When using a local mechanism, all compilation units using the template should be recompiled, so additional dependencies should be added to the makefile. In practice, a tool aware of exported templates should be used to generate the makefile dependencies.

When using a global mechanism, the pre-link phase should be able to trigger the needed recompilation: it only needs to be able to detect that the instantiations are out of date; being able to launch recompilations is inherent to this mechanism. Exported templates provide a natural way to trigger the pre-link phase and to allow it to check the consistency of the objects.

What happens when a definition becomes available late?

When a definition becomes available when it was previously not, the used instantiations need to be provided. That can be considered as a modification of a stub definition and the needed recompilations would be the same.

What to expect from export?

Compared to the inclusion model, what are the expected effects of using the separation model?

- It removes the need to provide the definition of the function template along with the declaration. This mimics what is true for normal functions and a behaviour expected by most people starting to use templates.
- It also removes the need to include all the declarations needed by the definition of the function templates, preventing a "pollution" of the user code.

The other effects are dependent on the instantiation mechanism used.

5 I'll consider an intermediate format to be *high level* if the source code without the comments can be reconstructed; an intermediate format which is not high level will be qualified as *low level*. Obviously in practice the separation between low level and high level is not clear.

6 That is where dependencies are generated by the preprocessor (with an option like `-MM` for gcc) or an external tool (like makedepend)

- in conjunction with a local mechanism, without duplicate instantiation avoidance (like Borland's), it could need more parsing than the inclusion model as the headers needed for both the definition and the declarations have to be parsed twice if one requires the definition to be available at instantiation time (as does the only implementation).
- in conjunction with a local mechanism with duplicate instantiation avoidance (like Sun's), it could reduce the needed file reading and parsing, but the disadvantage for the inclusion model may be reduced by using techniques such as precompiled headers
- in conjunction with a global mechanism
  - it reduces file reading and parsing, but the disadvantage for the inclusion model may be reduced by using techniques such as precompiled headers.
  - it reduces the need for recompilations after a change in the template as only the compilation unit providing the instantiation has to be provided.

Experiment report

I have performed some experiments with exported templates using Comeau's compiler, to check if they are usable. I wanted to see if it was possible to set up a makefile so that all needed recompilations were triggered automatically without adding dependencies manually, to see if it was possible to use them with libraries, and to see if it was possible to organize the code so that it could be compiled in both the inclusion model and separation one.

I also wanted to check if the expected effects on the instantiation mechanisms described above where measurable. As Comeau's compiler provides a global mechanism, I expected a reduction in compile time, and a reduction in file reading and parsing, and I wanted to see how it compared with what could be obtained with using pre-compiled headers.

Obviously such effects depend on the code. The simple setup I used was designed to be favourable to export: a project made of a simple template function making use of the standard IOStream implementation, but not its interface, was instantiated for the same argument in ten files containing very little else. In such a setup if export did not provide a speed up in compilation time, there is little hope that it will in real life projects.

I measured
- the time to build from scratch
- the time to rebuild after touching the template definition file
- the time to rebuild after touching the header defining the template argument type

For each kind of compilation[7]:
- normal compilation
- using precompiled headers
- using export

The results are seen in this table:

	Normal build	Precompiled headers	Exported template
From scratch	10.2	5.2	3.7
Touching the type definition	9.4	4.7	2.5
Touching the template definition	9.3	4.7	2

7 I also tried to measure it for the combination of export and precompiled headers, but it triggered a bug in Comeau's compiler.

One can see that, at least for this kind of use, exported templates have some benefit in build time. This is especially true when modifying the template definition (which for exported templates resulted in one file compilation and a link, while there where several file compilations for both the normal build and when using precompiled headers), but the effect of parsing reduction can be seen when the same instantiation is used in several files and when the use of export reduces the need for include (in the experiment: the `<iostream>` and `<ostream>` headers were only needed in the template definition).

Obviously, in more realistic scenarios, the proportion of the timing reduction would be different and using export could result in degradation of building time when template instantiation was used in only one compilation unit or when the usage of export does not reduce the need to include files.

CFront compilation model and instantiation mechanism[8]

When instantiating templates, CFront compiled (in a special mode to ensure that only template instantiations were provided) a new compilation unit made up of
- the file containing the template declaration,
- a file expected to contain the template definition whose name was made up by changing the extension of the file containing the declaration,
- a selection of files included in the file which requested the template instantiation,
- special code triggering the wanted instantiations.

A name (dependent or independent) used in a template, was searched in the context of instantiation in this compilation unit, this context was different but usually very similar to the context at the true instantiation point.

CFront compiled template instantiations at link time. A pre-linker launched a link, deduced the required instantiations from the missing symbols and generated them if they where not already present in a repository. Then it restarted the process until all needed instantiations where available. The behaviour of CFront was reputed to be slow (linking takes a lot of time and doing several of them takes even more so) and fragile (needed recompilation of instantiations sometimes did not occur and so the first step in handling a strange error was to clean the repository and recompile everything).

Jean-Marc Bourguet

Bibliography

[1] David Vandevoorde and Nicolai M. Josuttis, *C++ Templates, The Complete Guide*, Addison-Wesley, 2003

[2] Bjarne Stroustrup, *The C++ programming language*, Addison-Wesley, second edition, 1991

[3] Herb Sutter, "export" restrictions, part 1, *C/C++ Users Journal*, September 2002, also available at `http://www.gotw.ca/publications/mill23.htm`.

[4] Herb Sutter, "export" restrictions, part 2, *C/C++ Users Journal*, November 2002, also available at `http://www.gotw.ca/publications/mill24.htm`

8 I've never used CFront, so this description is not from a first hand experience but is the summary of information found at different places.

The Nature and Aesthetics of Design

An extended review by Jon Jagger

Author: David Pye

Publisher: Herbert Press Limited

ISBN: 0-7136-5286-1

David Pye was an architect, industrial designer, and wood craftsman. He was also the Professor of Furniture Design at the Royal College of Art, London. He wrote two books that I know of (the other one is called The Nature and Art of Workmanship).

Chapter 1: Art and Science. Energy. Results.

In this chapter the author distinguishes between art and science with the observation that designers have limits set upon their freedom whereas artists typically do not. The idea of design limitations and their liberating effect is an important one which you may recall from Safer C [1]. The author then considers "function" and the fuzzy and unhelpful notion of form-follows-function. He asserts that the major factors in designing devices are considerations of economy and style - both matters purely of choice. Further, that every device when used produces a concrete, measurable, objective result and this is the only sure basis for a theory of design. When reading this I was reminded of eXtreme Programming and testing. Later in the chapter the author talks about systems - that things never exist in isolation - that things always act together and need to be in balance [2][3]. The author also says that design is not just about getting the intended results, it is as much about averting possible consequences of unwanted inputs [4].

Chapter 2: Invention and design distinguished

In this very short chapter the author states that invention is the process of discovering a principle whereas design is the process of applying that principle. It is interesting that in his other book there is also a very short chapter in which design *and workmanship* are distinguished: design can be conveyed in words or drawings, workmanship cannot. Based on this distinction one might say that design and workmanship is the process of applying a principle. (In fact later in the book the author asserts that workmanship *is* design). He also talks about description [5] - that the description of an invention describes the essential principle of the device, which is purely a matter of its arrangement.

Chapter 3: The six requirements for design

All but one of the six facets presented have clear parallels in software. The only one that doesn't is that the appearance must be acceptable. Since software does not have a physical manifestation the closest parallel to appearance must be just the names embodied in a software design. Names matter. The chapter discusses the six requirements in the context of how far, if at all, each of the requirements limits the designer's freedom of choice (relating back to chapter 1). The third requirement concerns the strength of the components; components must be strong enough to transmit (delegate) or resist forces. The relationship between strength and size is examined. For example, large bridges need to support not only their traffic but also their own weight. Economy and minimalism are also briefly discussed (he returns to the theme of economy in a later chapter). The fourth requirement of use is that of access. A device may be conceptually self-contained but in truth every device is part of a larger system that man is always part of. "The engines...must allow access for the engineer's hands when he is maintaining them". The idea of complexity is briefly touched on: "the most characteristic quality of modern devices is their complexity".

Chapter 4: The geometry of a device

This is a short chapter and of limited applicability to software. However, the idea of achieving results by way of intermediary results is covered and has clear parallels with software refactoring [6].

Chapter 5: Techniques. Skill.

This is a fascinating chapter. The author discusses construction as the idea of making a whole by connecting parts together. He talks about the need for techniques enabling us to vary properties independently and locally. The issue of size crops again and the author states that many techniques, when taken together, are so versatile that they impose hardly any limitations on the shape of the design (that form does *not* follow function) but that they do impose limitations on its size and that "standard pieces tend to be small". Again I am reminded of Richard Gabriel and Christopher Alexander, both of whom emphasize the importance of components right down to 1/8 of an inch. The second part of the chapter is devoted to skill. He defines any system in which constraint is variable at will as a skilled system. He asserts that skilled systems are likely to be discontinuous; that the intended result is arrived by way of a series of intermediate results. Again the parallels with refactoring are strong. Then there is an illuminating section on speed in which the author says that a system with skilled constraints usually gets the intended results slowly. He gives two reasons for this: because the change is likely to be discontinuous, and because it is easier to maintain constraints if energy is put in slowly. This chapter, particularly the part on skill, has a strong connection to chapter 2 of David Pye's other book (The Nature of Art and Workmanship). In that chapter the author makes a distinction between manufacturing and craftsmanship by defining manufacturing as the workmanship-of-certainty and craftsmanship as the workmanship-of-risk. In other words, something can be manufactured, even if made by hand (possibly with the aid of jigs, etc) if the risks involved in its creation are minimal. On the other hand, something is "crafted" if there are ever-present risks involved in its creation; if "the quality of the result is not pre-determined, but depends on the judgment, dexterity and care the maker exercises as he works". Which of these two sounds like software? The second book revolves around workmanship-of-risk and the idea that it has long been widely valued.

Chapter 6: Invention: analogous results

In this chapter the author considers the influence and dangers of similarity (designing something as an adaptation of an existing design). He urges us to consider first and foremost the intended results. But on the other hand he points out that if the problem is old, the old solution is likely to be the best (the alternatives are that new techniques have been invented or that all the designers in previous generations have all been fools). The final paragraph considers the influence of similarity again: invention and design are in tension. If you improve your powers of design is it necessarily at the expense of your powers of invention?

Chapter 7: We can wish for impossibilities. Utility, Improvement, Economy

This is a somewhat philosophical chapter. The section on improvement in particular talks about the secret of happiness and notes that there is no secret of unhappiness, that avoiding unhappiness does not imply happiness. The section on impossibilities is surprisingly relevant to software. The author mentions abstraction when he writes "We get experience by attending and we do that by abstracting one thing or event from all those in reach of our perception and then ignoring the rest". Or to use Andrew Koenig's sound-byte, selective ignorance. What has this to do with impossibilities? The author asserts that we can *never* wish for impossibilities; we can only frame our wishes in terms of past experience, experiences of the possible.

Chapter 8: The requirements conflict. Compromise.

This is a chapter resonating with parallels in software. The author's basic premise is that design requirements are *always* in conflict and so it follows that *all* designs are arbitrary. Where and how limitations apply is, ultimately, a matter of choice. However, some of the major limitations of physical design are simply not present in software. For example, a physical design often requires several parts which need to be in the same place at the same time and compromises must be made. This does not apply (or applies much less) to software. Given that all design is arbitrary, it follows that there are no ideal, perfect designs. That is not what design is about. Design is about creating a practical balance between competing and conflicting requirements. And no matter how you strike the balance your boss always wants it sooner and cheaper.

Chapter 9: Useless work. Workmanship

This too is a fascinating chapter. In it the author contends that man performs an immense number of every-day actions which are not necessary. For example, many Paleolithic tools are made with better-than-needed workmanship. There is something very deep about workmanship. Workmanship is design. The author states that "the most noticeable mark of good workmanship is a good surface". What is the surface of software? Isn't it simply its physical appearance? Layout matters.

Chapter 10: Architecture. Inventing the objects

This is an interesting chapter, a flavour of which is best summed up by quoting the opening paragraph. "Architecture is differentiated from engineering and from nearly all other branches of design by the fact that the architect has to act as if no object in the result, except the earth itself, is given. His first preoccupation is neither with how to get the intended result, nor with what kind of result to aim at, but with deciding what the principal objects are."

Chapter 11: 'Function' and fiction

In this chapter the author again urges the reader to abandon the idea of form and to concentrate instead on results. In the last paragraph of this short chapter he asserts that "economy, not physics, is always the predominant influence because it directly and indirectly sets the most limits." Perhaps the non-physical nature of software is not so relevant after all.

Chapter 12: The designer's responsibility

This chapter is also somewhat philosophical in nature. The author argues that since a designer always has some freedom of choice, they have a duty to exercise that freedom with care. The design of physical things in the environment, no matter how small, really matters; designing one motor car for one person to drive means designing millions of cars for thousands of people as scenery. In design, as in art, small things matter. Indeed the author goes further saying "art is not a matter of giving people a little pleasure in their time off. It is in the long run a matter of holding together a civilization." He invites you to consider what would happen if each new generation rebuilt its entire environment so that everything was always new.

Chapter 13: The aesthetics of design

This is another thought provoking chapter, this time on aesthetics, beauty, and value. He argues that design appreciation is beauty appreciation. He also questions whether aesthetics matter and touches again on happiness and unhappiness. He argues persuasively that while beds reduce cold, ploughs reduce hunger, and toothbrushes reduce toothache, such devices do not, of themselves, endow happiness. Being cold, hungry and in pain can certainly make you miserable, but for many people being warm, fed and pain-free is not enough to live for. Or, as he puts its, "not having toothache is no goal for a lifetime".

Chapter 14: Perception and looking

In this chapter the author considers how we look at things and how we see things. These clearly relate to the aesthetics of design (the title of this book) and to taste in design (a topic covered in the next chapter). After a section on the biological process of seeing he draws the distinction between sight and perception; we are born able to see but we have to learn to perceive, that we learn slowly, but then become largely unaware of the skills involved. He again returns to the idea of abstraction, "if we could not ignore we should die" and how perception requires abstraction.

Chapter 15: Taste and style

This chapter continues the theme of perception and notes that recognition occurs by means of a *few* characteristics rather than many. In contrast, he asserts that to experience the beauty of something arises from the totality of its characteristics and their relationships. That "to recognize the style of a design and to appreciate the beauty of it are two quite different things". He argues that design without style is an impossibility and that "any style ... has positive value ... for it puts limits on designers' freedom of choice". He also discusses change noting how civilization seems to be losing the concept of continuous change by small variations (ie seems to be losing tradition).

Chapter 16: Originality

This chapter starts by considering the role and nature of artists but quickly comes to the conclusion that originality is largely irrelevant to art. The last paragraph ends with a sentence that could have come straight out of [2] or [3] "The best designs have always resulted from an evolutionary process, by making

[concluded at foot of page 24]

Software development and the learning organisation

by Allan Kelly

"When you ask people about what it is like being part of a great team what is most striking is the meaningfulness of the experience.... Some spend the rest of their lives looking for ways to recapture that spirit." Peter Senge, 1990.

Think back over the last ten years, what have you learnt? What have we, the programmer community, learnt? En masse we've learnt C++, added the standard library, re-learnt ISO-C++, incorporated meta-programming and picked up Java, C99 and C#. And don't forget Python, JavaScript, Perl and other scripting languages that didn't exist in 1993.

Then add the technologies we've invented and learned: HTML, CGI, HTTP, ASP, JSP, XML, ... Or maybe you've avoided the Internet and just moved through Windows 3.1, 95, NT, 2000 and XP, or one of a myriad of Unix/Linux systems and versions.

The programmer community is constantly learning. If change is the only constant then learning is the only real skill you need. Indeed, how can we change if we can't learn?

Being a programmer you learn the technologies, but you also have to learn your "problem domain." That is, the field you are developing applications for. Some are lucky enough to work on IT technologies and develop e-mail systems, or databases, but most of us work outside the technology domain, so, I've managed to become a minor expert in train time-tabling, electricity markets, and financial instruments.

All the time we are learning, our teams are learning and our organisations are learning. So too are our customers who use our products. This happens whether we intend it to or not. Authors like Peter Senge and John Seely Brown argue that we can harness this learning through "Organisational Learning", so creating "Learning Organisations" which deliver better businesses - and in our case better software.

How does learning relate to software development?

If we look at the software development process there are at least four key learning activities:

- Learn new technology
- Learn the problem domain
- Apply our technology to the problem, the process of problem solving is learning itself
- Users learn to use our application and learn about their own problem - which changes the problem domain

Each one of these points reinforces the others: in our effort to solve a problem we need to learn more about the problem, our solution may use a technology that is new to us. When the user sees the end product their mental model of the problem will change too. They too will learn, through the software, and acquire new insights into the task that may lead to changes to the software.

Learning is thus inherent at every stage of software development. We can either choose to ignore it and muddle through somehow, or to accept it and help improve the learning process.

Maybe your manager is having difficulty understanding this - after all he hired you because you already know Java so why do you need to learn some more? - so let's put it in business terms.

Although we can buy the fastest machines on the market, only hire people with an IQ above 160 and expand our teams endlessly and work to ISO-9001, we aren't actually doing anything our competitors can't do. All we are doing is proving we can spend money. Worse still, none of this guarantees we will actually develop good software.

If instead of viewing software development as a problem task we view it as a learning activity we get some new insights. First we can recognise that most developers actually want to make customers happy, what is more they like learning new things. It is just conceivable that a firm which encourages its staff to learn will find it easier to retain staff, hire new staff and at the same time see the ability of existing people increase.

Now to be really radical, John Seely Brown and Paul Duguid believe that learning increases our ability to innovate. If we can learn better we will find we can produce new ideas and better solutions.

Since software development is intrinsically a learning process it doesn't seem that great a jump to claim recognising it as such, removing barriers to learning, and promoting learning within our group will improve our software. Once we do this we have something that competitor firms can't copy because we will create our own environment.

What can we do to promote learning?

The one thing I'm not suggesting is that you go to your boss and ask to be sent on a course. Brown and Duguid (1991) suggest there are two types of learning:

[continued from page 23]

successive slight modifications over a long period of time". Originality and innovation often hinder improvement.

Chapter 17: The common ground between visual art and music. What we really see

The final chapter has few parallels with software that I can perceive. However it does mention the importance of context and it also touches on size again. Many painters maintain that a picture has a "right-size" and that making it smaller or larger will be to its detriment. The theme of size recurs. I'm increasingly sure that making software elements the "right-size" is important in a deep way. I'll leave you with some quotes from Richard Gabriel "build small abstractions only", "buildings with the quality are not made of large modular units", "its modules and abstractions are not too big", "every module, function, class, and abstraction is small".

Jon Jagger

References

[1] Les Hatton, *Safer C*. See section 3.1 (p87) on Discipline.
[2] Richard Gabriel, *Patterns of Software*. Part I in particular.
[3] Christopher Alexander, *The Timeless Way of Building*.
[4] Henri Petroski, *To Engineer is Human* (subtitled The Role of Failure in Successful Design).
[5] Michael Jackson, *Software Requirements and Specifications*. See p58 - Descriptions.
[6] Martin Fowler, *Refactoring*.

- Canonical learning: going on courses, sitting in class rooms, reading manuals
- Non-canonical learning: learning by watching, doing, listening and a whole bunch of other stuff.

What is more they go on to suggest that the second form, is the better. By learning non-canonically we are more flexible and innovative. While canonical, 5-day, £1500 courses have a use, there are a lot more things we can do to promote learning in our organisations.

Before we go further, it is worth pointing out that there are two types of knowledge. The sort we're all familiar with: explicit knowledge, which can be written down, codified. Go pick up your copy of Stroustrup, you can learn C++ from that, its all explicit knowledge.

The second form is subtler: tacit knowledge. This is more difficult to write down and we normally learn it by some process of osmosis. Much of this knowledge is embedded in our work environment or our programmer culture. So, when you write `delete this` you aren't breaking any of the rules in Stroustrup, it's legal, but all of a sudden you're in trouble because your team doesn't do that. Of course, coming from a COM shop you think `delete this` is perfectly OK.

This is a simple example of tacit knowledge and the fact that I can write it down maybe invalidates it but I'm sure you get the idea. But how do we learn this stuff?

We get much of it through our society. That is, by watching other programmers, reading their code, exchanging war stories. Being programmers of course we like things to be black and white, quantifiable, written down, codified, so I'm sorry to tell you it ain't going to happen. In fact writing it down may make things worse!

In part we have so much knowledge we can't write it all down. Some of it is so obvious nobody thinks it worth writing down, and some we can't even put into words - although we may be able to mumble some grammatically incorrect phrases over a beer which stick in someone's mind.

Tacit knowledge is the reason so many specifications are inadequate. The stuff is a lot like jelly, you can't nail it down, it isn't in the specification because it is hard to codify. Only when you come to use the specification do you find gaps that are hard to fill. Computer code is inherently explicit knowledge.

Acquiring and learning to use this knowledge can take time. We need to be immersed in the society and let this stuff lap around us. Eventually we may try and do something, and from this we learn some more.

This brings us to another important point about this kind of learning. We're going to make mistakes. There will be failures. We need room to try ideas, see how they work, or don't work and add that information to our mental models. If we don't make mistakes part of our internal model will be missing.

For example, have you ever needed to write a piece of code and thought "I bet I could use templates for that? But I don't really know enough about meta-programming, O, I'll give it a try." And although it takes longer at the end of the week you have something that works and - very importantly - you understand templates? Along the way you probably made a million syntax errors, many compilation errors and got umpteen unexpected results but in doing so they completed your mental model.

Now the difficult bit for managers is to accept this trial-and-error approach. We need it. And although it doesn't look good when you're filling in the weekly timesheet the numbers are hiding the learning that occurred during development.

So, we need to accept mistakes will happen, we need to accept that risk. And maybe we need to do away with processes that penalise taking risks and making mistakes. (Although you may not want to take risks the night before release.)

By implication here there needs to be trust. If we don't trust someone to do the job they won't feel able to make those mistakes. They'll stick to the tried and tested solutions. We need to trust them to know when it is OK to play and when they should focus.

Nor does learning finish with our own team. The QA team will be learning how to use our releases, and the best way to test them. And when we finish and throw the software over the wall, users will start their learning.

What should we not do?

Even those who have worked in adversarial, deadline driven environments have learned. The first thing we need to stop doing is denying that learning happens. Brown and Duguid point out that when managers deny that learning is occurring two things happen:

- Individuals feel undervalued, managers don't recognise the role they play
- Managers think their systems are working, "We sent them on a Java course and now they all write quality Java" when in fact everyone is helping everyone else.

These effects describe "Plug compatible programmer" syndrome. Management feel, or suggest by their actions, that they can "just hire another C++ contractor" to plug a programmer shaped gap. After all, all C++ programmers are compatible. Meanwhile, developers know you need more than just C++ knowledge to work on the system, so they feel their skills aren't recognised by management and leave.

Brown and Duguid also suggest that seeking closure can be damaging too. By seeking closure, say by writing up all that is known about a given application, complete with UML diagrams, we actually inhibit future change.

I once worked on a team who worked to ISO-9001 standards. You weren't supposed to change the code without changing the documentation. Not only did this increase the work load but it made you reluctant to change anything. Increasingly code and documentation said different things and you had to talk to people, or step through the code with the debugger to see what was happening, that is, exactly the situation the standard was meant to prevent!

The need for closure made things worse. This happens all the time with documentation, whether it is program documentation or specifications. The emphasis on closure is one of the fundamental reasons waterfall methodologies are so troublesome.

Closure prevents change and it prevents further learning, but change and learning will happen. This doesn't only happen with us, the developers, it happens with our customers. A customer signs off on a spec, we develop the UI, show it to the customer who now wants to change it. In seeing the UI the customer has learnt something. Customer and developer are both engaged in a joint learning process.

This search for closure manifests itself in many forms: product contract, specification, program documentation, working procedures, code standards and perhaps the worst of all: code itself!

Some degree of closure is always necessary, otherwise we would never make a release. However premature closure limits learning opportunities and development. We need to strike a balance.

Likewise, we need to strike a balance on how much risk we accept. One organisation I know introduced procedures asking for every change to be estimated in terms of lines of code. Developers would naturally over estimate but it also made them more risk averse, there was clear message: management wanted change and risk limited. Thus they limited learning opportunities - specifically code refactoring.

Blunt measurements such as lines of code, and timesheets asking what you do with every half-hour in the week also send another message: you aren't trusted. These are tools of managers who believe that software development is an industrial process. "Scientific management" is at odds with the concept of learning because it doesn't allow for learning and change. It assumes that one person knows best, the manager, the designer, or the architect has looked at the problem and found the one true solution.

Practical things to do

Organisational learning isn't a silver bullet, you can't go out and buy it from Rational or Accenture. It is a concept. One of the ways it manifests itself as a *Learning Organisation.* You have to build this for yourself. The good news is that it need not entail a lot of up front expenditure, no expensive courses from QA or Learning Tree.

Much of the change is about mindset: accept mistakes will happen, accept some knowledge is tacit, accepting change, trusting people and leaning to live with open issues.

Managers face a difficult position. They can't force anyone to learn, nor should they. However, there are some practical things they can do to encourage the process. These have both an obvious role to play and a less obvious: by arranging these events and making time for them there is a message that "it is good to learn."

Whatever your position you need to start with yourself. For an organisation to learn the teams making up the firm must learn, for teams to learn individuals must learn. And we must learn to learn.

If you want to start encouraging others here are a few things you could try:

- Set up an intranet and encourage everyone to produce their own web pages, even if these aren't directly work related.
- Keep documentation on the intranet in an easy to access format, e.g. if your developers are using Solaris don't put the documents in Word format.
- If your documentation must live in a source control system then arrange for it to be published to the intranet with the batch build.
- Allow an hour or two a month for "tech talks" - get developers to present work they have done on the project or outside the project.
- Encourage debate - friction can be a good thing.
- Organise a book study group.
- Make your procedures fit your working environment and be prepared to change them.
- Hold end of project reviews, Alistair Cockburn (2002) even suggest such reviews should be held during the project - why wait to the next project to improve things?

Finally

I've only scratched surface of this subject, I'm still learning a lot about it myself but I think it has important ramifications for the software development community.

Unfortunately these ideas really require support from management before they can really deliver benefits, and I know most Overload readers are practising programmers who regard managers as part of the problem not the solution. That's why I spent some time advocating organisational learning in language they may understand.

Still, there is a lot we as a community can learn here and most of it has direct applicability to software development on a day-to-day basis.

Allan Kelly
`allan@allankelly.net`

Bibliography and further reading

John Seely Brown and Paul Duguid, 1991: "Organizational learning and communities-of-practice: Toward a unified view of working, learning, and innovation", *Organisational Science*, Vol. 2, No. 1, February 1991, also `http://www2.parc.com/ops/members/brown/papers/orglearning.html`
I've drawn extensively on this article, although it is quite long (18 pages) it is well worth a read.

Brown, Collins & Duguid: *Situated Cognition and the Culture of Learning* - `http://www.ilt.columbia.edu/ilt/papers/JohnBrown.html`
More psychological than the first but still interesting. Brown is a PARC researcher and has several interesting papers at `http://www2.parc.com/ops/members/brown/index.html` - including some on software design.

Cockburn, A., 2002: *Agile Software Development*, Addison-Wesley, 2002
I haven't heard anyone from the Agile methodologies movement specifically link them with Learning Organisations but I instinctively feel they are. (Hopefully I'll explore this some more in future.) In the meantime you'll find terms such as "courage" and "coaching" used in both sets of literature, and similar discussions on the importance of people, teams and listening.

Nonaka, Ikujiro, et al., 1995: *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, 1995
Nonaka distinguishes "Knowledge Creation" from organisational learning but the two are complementary. As the title suggests, this book concentrates on the way Japanese companies exploit knowledge creation.
For a book summary see:
`http://www.stuart.iit.edu/courses/mgt581/filespdf/nonaka.pdf`

Senge, P.M. 1990: *The Fifth Discipline*, Random House, 1990
Easy to read, authority introduction to "The Art and Practice of the Learning Organisation." To a hardened software engineer this may look like a touchy-feely meander. Don't let this put you off, you can't get the most from people if you stick with a binary view.

Addendum to “Tiny Template Tidbit”

by Oliver Schoenborn

I would like to add a few comments to the Overload 47 article that I wrote, entitled “Tiny Template Tidbit”. They don’t change conclusions of the article but may prevent some confusion.

First, as kindly pointed out to me by Alan Griffiths and Chris Main, I misuse the term “partial specialization”.

On page 16, 2nd column, I say “what we need is a template overload for pointers”. No problem so far. I then explain briefly that it is somewhat like partial specialization available for classes but not the same. Unfortunately, the code examples contain C++ comments that describe some syntax as “partial specialization” applied to functions (not supported in C++). They should instead read “overload for ...”. I must have had a few beers too many that day.

Secondly, a minor mistake in the formatting of the two lines “Advantages:” and “Disadvantages:”. The typeface makes it look like there are three paragraphs of advantages and almost two pages of disadvantages! The disadvantages use up only three paragraphs as well.

Finally, an interesting mistake was pointed out to me by Chris Main. It appears that a bug in the SGI MipsPro C++ compiler allows the compiler to pick the intended `getCoord()` function template even though, according to the Standard, it should not be able to. This is worth expanding upon. Let me recall the summary here:

I showed how a processor (object or function) that uses data from a set of objects of a certain type can be generalized with a very small number of lines of code, using templates and specializations, such that the processor doesn’t need to be changed when the data structure changes. More specifically, your processor object or function is able to extract the needed data from the container elements, regardless of whether the container element type

1. *Is* the data, or is a pointer to the data (as opposed to containing it)
2. Is an object or a pointer to an object, *containing* the data
3. Makes data available as *attribute OR method* (which we don’t address in this addendum)

Finally, the compiler does this for you automatically, without requiring your intervention.

Two of the functions and templates used were:

```
/// General function template
template <class P> inline
const Coord& getCoord(const P& p) {
  return p.coords;
}

/// Overload for pointers to things
template <class P>
const Coord& getCoord(const P* p) {
  return p->coords;
}
```

where `P` is one of the six types of data described in 1-3 above. As Chris pointed out to me, given the code fragment

```
Struct Point {Coord coord;};
Point coord;
getCoord(&coord);
```

the compiler has two choices for `getCoord()`:
1. `getCoord<Point*>(const Point*&)`
2. `getCoord<Point>(const Point*)`

Which one will the compiler choose? The one we want (#2) or #1? With SGI’s mipspro compiler, it chooses #2. With gcc 2.95, it chooses #1. I thought it was a bug with gcc but Chris gives good evidence that it is the other way around: overload resolution states that the compiler must choose the most “specific” template, which is #1, since the parameter is a pointer to the type rather than just the type.

This could seem like a major problem for the techniques discussed in that article but really it isn’t because the concepts used were sound, it’s just the technique used to implement them that got side-tracked onto a wrong path. The fundamental problem is how to tell the compiler to deal with two separate cases of data, one a type, the other a pointer to a type (a “pointer type”). This is easy with partial specialization of a class that does nothing for types, and does a dereference for a pointer type:

```
template <typename P>
struct Obj {
  static const P& getObj(const P& p) {
    return p;
  }
};
// partial specialization for pointer
// types
template <typename P>
struct Obj<P*> {
  static const P& getObj(const P* p) {
    return *p;
  }
};
```

They replace two of the four `getCoord()` overloads mentioned in the article, namely the ones with pointer types as parameter, and are used by calling

```
getCoord( Obj<T>::getObj(p) )
```

instead of

```
getCoord(p)
```

In any case, I hope you found the article interesting and maybe even useful. Thanks to Alan and Chris for sending feedback. As usual, I find that writing articles is fun and challenging, but I inevitably seem to learn a lot more than I could have expected from the feedback of readers. I encourage you to give it a try, write an article for Overload!

Oliver Schoenborn
Oliver.Schoenborn@utoronto.ca

References

[1] Bjarne Stroustrup, *C++ Programming Language*, 3rd ed.
[2] Andrei Alexandrescu, *Modern C++ Design*
[3] Oliver Schoenborn, “Tiny Template Tidbit,” *Overload 47*

Observer Pattern Implementation

Correspondence between Stefan Heinzmann (`stefan_heinzmann@t-online.de`) and Phil Bass (`Phil@stoneymanor.demon.co.uk`).

SH: I finally found some time to read your article series in Overload 52&53, and I've got some comments:

PB: Thanks for spending the time to provide some feedback. It's very much appreciated.

SH: I used to work on a framework that supported connections between logic-gate-like objects in much the same way as you describe. It was to be used in control systems in industrial user interfaces (real physical knobs and buttons). Thus I think I've got some experience to draw from.

PB: Yes, that's exactly what we have. In our case it's mainly buttons and lamps - very few knobs.

SH: You wrestle with the problem of copying events (or the objects that contain them). Funny enough that you say yourself that you often try to solve the wrong problem if you find that the solution seems elusive. I couldn't agree more. I think events shouldn't have value semantics. Here's why:

You rightly associate an event with an output in a logic gate. The logic gate would be represented by an object that contains the event as a member object. What does it mean then to copy such a gate with its event member? It would be much like copying a gate on a schematic drawing, right? If you ever worked in hardware design, you know that copying a gate gives you a gate of the same type but with its pins detached. Copying a gate doesn't copy its connections!

PB: Absolutely. I haven't done any electronics at all, but I have worked alongside electrical and electronics engineers. And, I agree that copying a logic gate wouldn't make sense if its connections were copied as well.

However, I'm interested in the general problem of de-coupling objects in one software layer from the higher-level objects that observe them. Perhaps there are some objects for which a copy operation really should copy the connections. I couldn't think of any specific examples, but I suspected that different copy semantics would be appropriate for different types of object. In particular, some objects shouldn't be copyable at all.

SH: What it really boils down to is that a gate has a unique identity (which is usually shown in schematics, i.e. "U205"). You can't have two objects with the same identity. Copying a gate creates a new gate with a new identity. Hence gates can not have value semantics. The result is that storing them in STL containers directly is a bad idea. You store pointers to them instead.

PB: Now you're losing me. I'm familiar with the idea of two broad categories of objects: 1) those with value semantics and 2) those with reference semantics. I appreciate, too, that identity is often of little or no importance for objects in category 1, whereas it is usually crucial for category 2 objects. However, I don't see why an object with identity shouldn't have value semantics. And I would be quite upset if objects with identity could/should not be stored in STL containers.

For example, what's wrong with a std::vector<Customer>? Real customers certainly have identity and I'd expect objects of the Customer class to have identity, too. You might argue that it doesn't make sense to copy a Customer (which is true), but it makes perfect sense to store Customers in a vector, and that requires Customer objects to be copyable. My Events are just like Customers: it makes no sense to copy them, but perfect sense to store them in STL containers.

SH2: I do see a conflict between object identity and value semantics. Let me quote from Josuttis' Standard Library Book (page 135): "All containers create internal copies of their elements and return copies of those elements. This means that container elements are equal but not identical to the objects you put into the container."

I think that this makes it quite clear that a std::vector<Customer> might not be such a good idea. As a further hint, consider what would happen if you'd like to hold a customer in two containers at the same time (say, one that holds all customers, and another that holds customers who are in debt).

What I'm saying is that if the only reason to make a class copyable is to be able to put them into STL containers, you're probably trying to do the wrong thing.

SH: The pointers may well be smart pointers. The details depend on your ownership model. It is not always necessary to use reference counted pointers. If you have a different method to ensure that connections are removed before the gates are deleted, then a bare pointer may be adequate.

PB: Agreed.

SH: Note that you don't necessarily need to prevent event copying altogether. It may well be useful to be able to copy events, but the key is that copying an event does not copy the connections. I feel, however, that this style of copying is better implemented through a separate clone function instead of the copy constructor.

PB: This really depends on whether we choose value or reference semantics, I think. No, on second thoughts, it depends on whether the objects in question are polymorphic (the virtual function kind of polymorphism). The value/reference semantics design decision is separate.

SH2: In the polymorphic case you have no choice but storing pointers in the container anyway. Copy construction is not needed here. If you nevertheless want to make copies of your objects, they need a virtual clone member function. The value/reference decision is not entirely separate, since you can not have value semantics with polymorphism in the same object. If you wanted to model that, you'd end up with a handle-body pair of objects, which is just another variant of the smart pointer theme.

SH: Regarding event ownership I think that a hierarchical ownership model may well be better than a distributed "shared ownership" model. If we carry on a little longer with the hardware analogy, I would propose to have a "Schematic" object that owns all the gates in it. Deleting the schematic object deletes all connections and gates. The problem then is reduced to "external" connections that go between the schematic and the outside world. Internal connections don't need to be implemented with smart pointers.

PB: Again, I agree that a hierarchical ownership model has benefits. In fact, I suspect our software could be improved by modeling the hardware more closely - tools contain modules, which contain other modules, which contain I/O boards - and the logic gates would be owned by the module that contains them. In practice, though, this is not particularly easy to do and I felt that discussing the short-comings of the existing software would distract from the main point of the article. The difficulty boils down to the difference you highlight, here. External connections and internal connections are not distinguished, so we would have to use a mechanism that supports the more general ownership model for both.

SH2: It would certainly go too far for the purpose of the article to introduce several levels of hierarchy. The point I was hinting at however was that looking at the problem from one level up in the hierarchy might render a different - and maybe more adequate - design. The problem here is a general problem with the observer

Labouring: An Analogy

by Seb Rose

The project plan says you're going to design your unit tests next week. Your code is being dropped into system test tomorrow. The Functional Spec was conditionally signed off on Monday (the condition was the successful resolution of one or two queries about inconsistencies in the Business Requirements Document)

Does any of this sound familiar?

Background

I've been a freelance consultant (or contractor scum or disguised employee) for ten years, though it seems longer. I've worked with plenty of your major, run-of-the-mill financial institutions that we all know, love, and trust with our money.

I'd started working with a popular Internet Bank at the height of the dot-com bubble when rates went silly (that's the rates they paid consultants, not the ones they paid their customers). Since then the bubble had burst, rates had collapsed and accepted wisdom was that those rates has gone for good and a recovery would mean being able to find work again.

And so I found myself sitting at my desk looking at the terms of another 6 month contract renewal from the human resources (HR) department of the bank.

So, why was I thinking so hard about refusing this renewal? Was I looking for more money? Did I think that the wording might have got me in trouble with Hector (remember him?) and the IR35 posse? No, neither of these. I was seriously considering refusing the extension on the basis that I just couldn't face wasting another 6 months.

I began calculating 6 months as a percentage of my age; as a percentage of my adult life; as a percentage of my future life. The percentage kept getting higher. In fact the more I though about it, the more it seemed that if I renewed this contract 6 months might constitute 100% of my remaining time on earth. Or maybe even more.

I didn't sign. I politely declined. After almost two years of trying to get a large business to adopt sensible software development strategies (like having a development process - not a specific process, but any identifiable set of activities that might conceivably constitute a development process) I just walked away. I left my PL/SQL test scripts to posterity and headed for the hinterland.

I decided I'd go back to my bad old ways.

Allegedly I'm a designer and/or architect (that's not what it said on my contract, which was of the generic "Hot-Deployable Freely-Interchangeable Plug-And-Play Software-Resource"-style) so I thought I'd do some designing and architecting. In the real world. With a building.

Another World

Now, I have no formal training in construction techniques, but some pals and me did build a lovely straw house a few years ago, so I have had some experience. (Any of you going to OT2003 will have the opportunity to join me as I use straw bale building projects to discover software requirements elicitation techniques, which should be fun!). Also, my partner and I had a house built for us a while back and have plenty of experience with the Planning and Building Control departments of our local council. (Luckily agricultural buildings don't actually require Planning Permission or Building Warrants, but more of that later).

I ran the project in an informal sort of way, with the customers on site all the time. My customers were vegetable growers that needed a shed to pack their vegetables after their old one had ended up 30 foot up an ash tree during a particularly violent storm in February.

So, I asked them what their requirements were (well, we actually had a more protracted discussion that started off with "What's it for?" but you get the idea) and I ended up with this list:

- The shed should be large enough to pack boxes of vegetables.
- It would be nice to have a separate space to store potatoes (and such like) in.
- An office space would be quite useful, too.
- It shouldn't be too warm in summer or too cold in winter.
- No vermin allowed in.
- Easy access for loading/unloading the van.
- Oh, and please could I make sure that it didn't blow away this time?

Could it be done for £3000 (not including labour) and be finished before the rains started in the autumn? Well, maybe not. It was in use within the timescale, but like so many projects it may never be 'finished'. And like so many projects it went over budget (by maybe 25%).

Awkward Analogies?

Why am I writing about this in a journal for software developers? I guess my basic question is that if small software projects are easy and fun, why are large software projects so difficult and depressing? Or in general, do you always have a bad time when things get big? My gut feeling says "Yes", but let's see where it goes.

The good things about this project were:

- We knew what was wanted
- Within reason we could use any appropriate technology
- I knew everyone working on the project and what they could be relied on to do
- No one was making silly rules that just slowed us down
- The budget & timescale weren't restrictive

Before we get started, let's dispel one or two myths.

1 We did have fun putting up this building, but it wasn't because all the work was enjoyable (putting rock wool insulation into a roof space is not fun) or that we spent all our time sloping off to the pub (the nearest pub was 5 miles away). We had fun because we were working as a team toward a recognisable goal.

2 We weren't 'pushing the envelope'. All the technologies involved were completely mainstream, so that wasn't what kept us happy. None of us were full time builders, so you could reasonably argue that the fun would lessen with repetition (I'll let you know).

Now let's have a look at where the 'good things' go when a project goes from small to large.

[concluded from page 28]

pattern: Who owns/controls the connections? It is by no means clear that it should be the subject that owns them. This would be analogous to a chip on a PCB that owns the wires connected to its outputs. Wouldn't it be more natural to think that all wires belong to the PCB itself? Each output could still have a container of pointers to connections for managing the updates, but it wouldn't necessarily own them from a lifetime management perspective.

Now, as you rightly point out, the hardware analogy isn't necessarily always the right one. So it will likely depend on the situation what kind of ownership model you would choose. This makes me wonder whether it would be sensible and feasible for a general purpose observer implementation to provide some latitude in this respect, maybe through policies (in Alexandrescu's sense).

We know what we want

How well do we ever understand the requirements of a project? Human systems are complex and the relationship between size and complexity is not linear.

As complexity grows the ability of any one human to grasp what's going on disappears and the likelihood that something has been missed grows. We can cope with this, but the personal cost is high.

The prize of component-based development is simply the management of complexity by dividing large projects into smaller ones. The problems is that specifying the interfaces to, and responsibilities of, a component is no easy matter.

We can use the most appropriate tools

Large organisations often have preferred suppliers. It may not be fit for your purpose, but the license has already been purchased, so you'll just have to use it. Even if the ideal product is free it hasn't been certified by the Platforms team, isn't supported by a team of highly motivated technicians, and hasn't got a brand name behind it. Anyway, if it's so good why are they giving it away?

You'll notice that I talked about large organisations rather than projects here. This sort of constraint is due to the context of the project rather than the project itself. You might be working on a small project for a small company, but if the major client is a large company with a 500 page procurement process you'll be doing things the way they want whether it's a good idea or not.

The project is manageable

Things can slip in any project.

There's an anti-pattern that I read about somewhere that describes how an item on the project plan can be 90% finished for weeks on end, because the 10% left isn't well understood.

Then there are the bugs, misunderstandings, design flaws and 3rd party product problems. Not to mention compiler bugs ("this should work, so it must be a bug in the compiler").

Without proper feedback processes you won't spot the slippage until it's too late. This is true for projects large and small. The thing is that it's easier to put effective feedback processes in to small projects: you're familiar with the overall vision, you understand the design, and you know the team. On large projects all these things work against you and there will likely be layers of management with an inflexible attitude to time boxes.

Equally, you need to understand enough about the problems and the people to match the person to the task. If you fail in this then your staff end up feeling less part of a team and more like a cog in a machine. I know which I prefer.

Rather than try to overcome this, most large organisations talk about people as resources and seem to think that people with the same keywords on their CV are freely interchangeable.

As an example, I overheard a discussion between one of the HR team and an IT program manager. They were trying to decide what to 'do' with a new-start that was arriving on Monday. They considered placing him as a mid-tier development team leader, a front-tier developer and a tester, eventually settling on the latter! They had hired a 'resource' without any idea of the role they were trying to fill, and with very little understanding of what skills he possessed.

Bureaucracy didn't stop us

How easy is it to fix a problem?

The answer depends on many factors. How fundamental is the problem? Who can decide on the appropriate solution? Who can do the fix? What processes need to be followed before any action can be taken?

I've found that it's the last question that seems to be the most limiting in large organisations.

When you have to coordinate large numbers of people you have to put processes in place to allow monitoring and control. The designing of appropriate processes - structured enough to deliver the management requirements, lightweight enough to not be a burden and flexible enough to cope with the whole spectrum of project problems - is a high art.

Many is the methodology that has been found wanting!

Project constraints were realistic

How long does the project plan say the bit you are working on is going to take? Where did that number come from? Did you provide the estimate? Did you provide the estimate, only to be asked to shave some time off it? Did you provide the estimate, only to have functionality added? Does the plan allow for testing or rework?

It is widely accepted that there are 3 variables in any project: scope, resources and time. They are related. You can't change one without a compensating change in one, or both, of the others.

As the project size grows so does the management layer, and for some reason the sort of manager that fills that layer just doesn't seem to buy this self-evident theorem.

It's not even that the theorem is too simple, because what they propose is often so simple as to be idiotic:

- "I can motivate the team to deliver more functionality in less time".
- "I can compensate by hiring more staff"
- "The team will pull together and work longer hours"
- "I've promised that we will deliver, and my word is my bond"

Again this really shouldn't have anything to do with project size, but the larger a project the less likely the grim reality will make it though all those middle management layers to someone who is prepared to make a tough decision.

Conclusion

The shed was a small project:

- The complexity was not overwhelming.
- We used aterials that we had to hand and sourced from local suppliers, without needing to get it approved by the client or building control.
- The people helping me were friends who I've known for years. One was time-served electrician, the others were just generally handy and reliable.
- We didn't need planning permission or a building warrant, because it was an agricultural building. Bureaucratic involvement was minimal.
- I had no unrealistic timescales and the budget was only slightly unrealistic.

We had a lot of fun. More fun than we would have had working on a building site.

Large projects could be fun if you could decompose them enough to understand what's going on and your organisation is helping rather than hindering. I'm just not sure that either of these things is achievable.

Seb Rose
`seb@acmeorganics.co.uk`