Creating a C++ Scripting System

Most of today's games require some kind of system to allow the game designers to program the story of the game, as well as any functionality that is not general enough to be directly supported by the AI or some other system. Often, the scripting system of a game is also made available to players for customization. Today it is not uncommon for third-party companies to customize an existing game to incorporate completely different gameplay, in part by changing the game scripts.

Depending on the game type, the best results are achieved using different approaches to scripting. In environments that have very well defined rules, such as RTS games, the most important task of the designer is to achieve good balance between the different units and resources, while also producing interesting maps with features that clever players can use to their advantage. Clearly, the scripting support for such games is focused on fast and easy tweaking of the different parameters exposed by the game engine, and is usually directly supported by the map editor.

Other games, for example many FPS titles, require very limited levels of customization. This is usually done by tagging objects in the game editor.

And finally, many games are driven by complex enough logic to require a complete programming language for scripting. In some cases, developers have designed and implemented their own programming languages to serve this need. To cut development time, today most game companies opt to use existing programming languages customized for their own scripting needs.

Scripting Languages

Most so-called scripting languages have one thing in common: they have been developed by a small team or even a single person for the purpose of writing simple programs to solve a particular, limited set of problems. By their very nature they are not universal, yet many end up being employed beyond their intended purpose.

Sometimes, "scripting language" simply means a programming language used to write scripts. Indeed, people have successfully used languages such as Java for scripting.

When selecting a scripting language for a game, chances are that a developer will not find a language that matches the desired functionality completely. At the very least, a developer will have to design and implement an interface between the game engine and the scripting language. The programming language the developer chooses is just one of the components — often not the most important one — of a scripting engine.

Using C++ for Scripting

One of the most important characteristics of C++ is its diversity. Unlike many other languages that are efficient for a particular programming style, C++ directly supports several different programming techniques. It's like a bag of tricks that allows many, often very different solutions to a problem. Some of the C++ features — such as function and class templates — are so powerful that even the people who developed and standardized them could not have foreseen the full spectrum of problems they can solve.

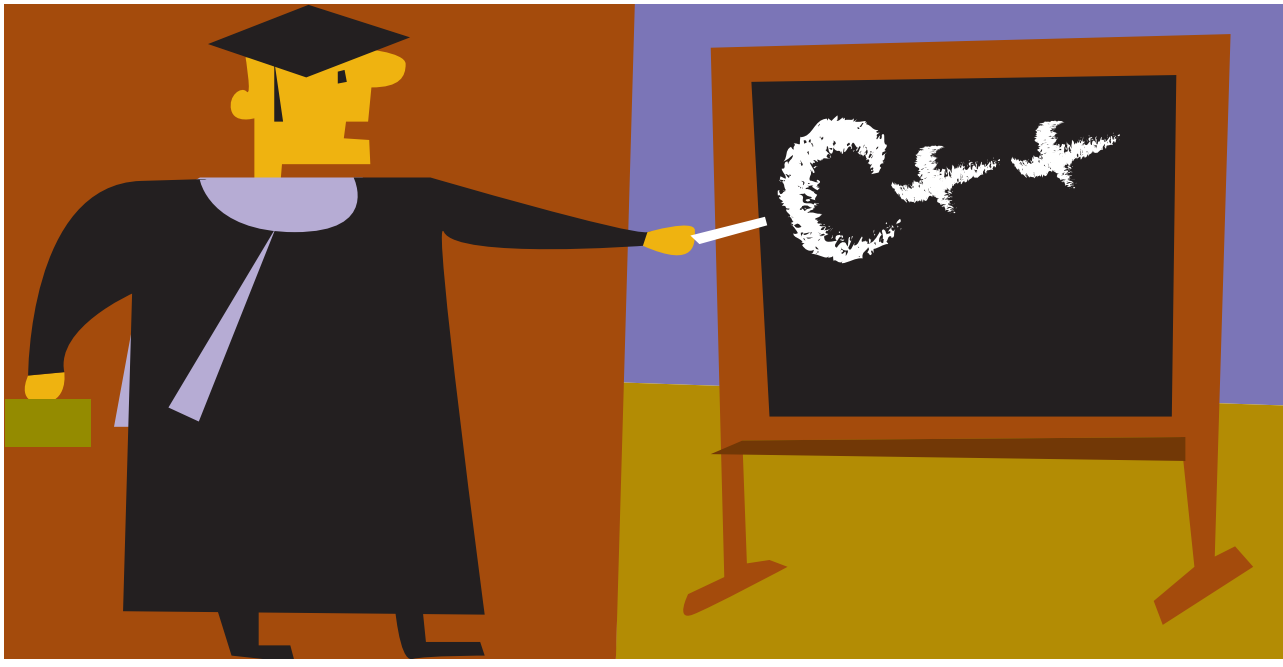
The idea of using C++ for scripting may seem strange at first. Indeed, most people associate C++ with pointers and dynamic memory management, which are powerful features but are more complex to work with than what most game designers would consider friendly.

On the other hand, C++ is the natural choice to program the rest of the game in. If the scripts are also written in C++, then integration with the rest of the code is seamless. In addition, C++ is translated to highly optimized machine code. While speed is rarely a problem for most game scripts, faster is always better.

Naturally, using C++ for scripting has some drawbacks. Because it is a compiled (as opposed to interpreted) language, a C++ program can't be changed on the fly, which is important in some applications. Also, C++ does not provide a standard for plugging in program modules at run time, which makes it nearly impossible to expose the script for customization by users (usually this is not an issue for console titles).

It's important to create a safe and easy-to-use environment for our scripts. The language used is a secondary concern, because by their very nature scripts are simple and mostly linearly executed, with limited use of if-then-else or switch state-

EMIL DOTCHEVSKI | *Emil is currently co-lead programmer at Tremor Entertainment on a soon-to-be-announced original Xbox title. His previous work includes RAILROAD TYCOON II for Playstation and an enhanced 3D version for Dreamcast. Emil has an M.S. in computer science from the Sofia University, Bulgaria. He can be reached at emil@tremor.net.*



ments. In this article we will demonstrate how to use some advanced C++ features to create a safe "sandbox" environment for the scripts within our game code.

Creating a Safe Scripting Environment

More often than not, programmers design a class hierarchy to organize the objects that exist in the game's digital reality. For the purposes of this article, let us assume that our game uses the classes whose partial declaration is given in Listing 1.

LISTING 1. An example class hierarchy.

```
class CRoot {
   virtual ~CRoot();
};

class CActor: public CRoot {
public:
   bool HasWeapon() const;
   bool HasArmor() const;
   int GetHealth() const;
   void SetHealth( int health );
   void Attack();
   ...
};

class CGrunt: public CActor {
public:
   ...
};

class CAgent: public CActor {
public:
   ...
};
```

Using C++ for scripting allows us to use plain pointers for interfacing with the script code. The main drawback of using pointers is that they can point to invalid memory. The benefits are that pointers are directly supported by the language and are very efficient. Also, pointers make it easy for programmers to implement and later extend the interface between the scripts and the game.

To eliminate the possibility for the script code to access invalid memory, the use of pointers must be hidden from the scripts. In C++, we can do this by organizing the pointers in a set container. Then we can define functions and operations for working with entire sets of (pointers to) objects. This neatly unifies the processing of one or more objects and usually allows the scripts to not have to handle empty sets as a special case.

For example, to represent a set of `CGrunt` objects (see Listing 1), we can use something like this:

```
std::set<CGrunt*> grunts;
```

Let's also define a functor to expose the `CActor::Attack` function to the script:

```
struct Attack {
   void operator()( CActor* pObj ) const {
      pObj->Attack();
   }
};
```

Now we can use `std::for_each` with the function object `Attack` to have all the `grunts` in the set use their attack functionality:

```
std::for_each( grunts.begin(), grunts.end(), Attack() );
```

The `for_each` template function is defined so that it can work with any sequence of objects, which is a level of flexibility we do not need. Instead, we can define our own version of `for_each`, which for convenience we can simply call `X` (from Execute):

```
template <class Set,class Functor>
void X( const Set& set, Functor f ) {
   for( typename Set::const_iterator i=set.begin(); i!=set.end(); ++i )
      f(*i);
}
```

Now we can simply say:

```
X( grunts, Attack() );
```

It's also possible to write functors that take arguments and pass them to the function they call:

```
struct SetHealth {
   int health;
   SetHealth( int h ): health(h) {
   }
   void operator()( CActor* pObj ) const {
      pObj->SetHealth(health);
   }
};
```

Now we can use the SetHealth functor like so:

```
X( grunts, SetHealth(5) );
```

Predicates

Using function objects to perform actions on an entire set of objects is a powerful feature by itself, but it becomes even more powerful if we define another version of the X function that allows us to call the function object only for selected objects in a set:

```
template <class Set,class Pred,class Functor>
void X( const Set& set, Pred p, Functor f ) {
   for( typename Set::const_iterator i=set.begin(); i!=set.end(); ++i )
      if( p(*i) )
         f(*i);
}
```

A predicate is a special type of function object that checks a given condition. For example, we can define the following predicate:

```
struct HasArmor {
   bool operator()( const CActor* pObj ) const {
      return pObj->HasArmor();
   }
};
```

Now we can write something like:

```
X( grunts, HasArmor(), Attack() );
```

This will execute the Attack functor on the members of the grunts set that are armored. Again, exposing the armor property of objects of class CActor to the script is as easy as writing a simple predicate.

Type Predicates

In the preceding examples, because Attack::operator() takes CActor* as an argument, the Attack functor can only be used with sets of objects of class CActor. Because any object of class CGrunt is also of class CActor, we can use the Attack functor with a set of grunts too. But what if we have a set of objects of class CRoot? It would be nice to be able to select only the objects of class CActor and execute Attack on them.

To do this, we need our predicates to define an output_type. Then we can design our system so that if an object passes a predicate, we can assume it is of class output_type that the predicate defines. For example, we could use the predicate IsActor that checks if a given object is of class CActor:

```
struct IsActor {
   typedef CActor output_type;
   bool operator()( const CRoot* pObj ) const {
      return 0!=dynamic_cast<const CActor*>(pObj);
   }
};
```

Note in this case that some compilers do not implement dynamic_cast efficiently because it has to work in nontrivial cases such as multiple inheritance and the like. Instead of dynamic_cast, we could use a virtual member function to do our type checks.

We also need to modify the predicate version of our X template function:

```
template <class Set,class Pred,class Functor>
void X( const Set& set, Pred p, Functor f ) {
   for( typename Set::const_iterator i=set.begin(); i!=set.end(); ++i )
      if( p(*i) )
         f( static_cast<typename Pred::output_type*>(*i) );
}
```

The output_type defined by our predicates makes it safe for the X template function to use a static_cast when calling the functor.

Now, if we have a set of objects of class CRoot, we can write:

```
X( objects, IsActor(), Attack() );
```

Complex Predicates

So now we have seen how to use predicates to execute a functor on selected objects from a given set of objects. But what if we want to combine multiple predicates to select the objects we need?

Generally speaking, it's easy to combine multiple simple predicates in a single complex predicate that our X function can check. As an example, let's define a predicate called pred_or:

```
template <class Pred1,class Pred2>
struct pred_or {
   Pred1 pred1;
   Pred2 pred2;
   pred_or( Pred1 p1, Pred2 p2 ): pred1(p1),pred2(p2) {
   }
   bool operator()( const CActor* pObj ) const {
      return pred1(pObj) || pred2(pObj);
   }
};
```

To make it possible to use pred_or without having to explicitly provide template arguments, we can define the following helper function template:

```
template <class Pred1,class Pred2>
pred_or<Pred1,Pred2> Or( Pred1 p1, Pred2 p2 ) {
   return pred_or<Pred1,Pred2>(p1,p2);
}
```

LISTING 2. Defining pred_or.

```
template <class Pred1,class Pred2>
struct pred_or {
   typedef typename select_child<
      typename Pred1::input_type,
      typename Pred2::input_type>::type
         input_type;
   typedef typename select_root<
      typename Pred1::output_type,
      typename Pred2::output_type>::type
         output_type;
   Pred1 pred1;
   Pred2 pred2;
   pred_or( Pred1 p1, Pred2 p2 ): pred1(p1),pred2(p2) {
   }
   bool operator()( const input_type* pObj ) const {
      return pred1(pObj) || pred2(pObj);
   }
};
```

LISTING 3. Defining pred_and.

```
template <class Pred1,class Pred2>
struct pred_and {
   typedef typename Pred1::input_type
         input_type;
   typedef typename select_child<
      typename Pred1::output_type,
      typename Pred2::output_type>::type
         output_type;
   Pred1 pred1;
   Pred2 pred2;
   pred_and( Pred1 p1, Pred2 p2 ): pred1(p1),pred2(p2) {
   }
   bool operator()( const input_type* pObj ) const {
      return pred1(pObj) &&
         pred2(static_cast<const typename
Pred1::output_type*>(pObj));
   }
};
```

With the `Or` function template in place, we can use `pred_or` to combine the `HasArmor` and the `HasWeapon` predicates:

```
X( grunts, Or(HasArmor(),HasWeapon()), Attack() );
```

But wait, why did we define `pred_or::operator()` to take `CActor*`? This is not ideal, because we want `pred_or` to be able to combine predicates that take objects of different classes. In addition, our `X` function requires us to define an `output_type`. What is the `output_type` for `pred_or`?

Let's extend our system to require that the predicates define not only `output_type` but also `input_type`, which is the class of objects the predicate can be checked for. With this in mind, let's define `pred_or` as shown in Listing 2.

For this to work, we need two helper template classes, `select_child` and `select_root`. We see how they work later, but for now let's just assume the following: `select_root<T,U>::type` is defined as the first class in the class hierarchy that is common parent of both `T` and `U`, or as void if `T` and `U` are unrelated. For example, `select_root<CGrunt,CAgent>::type` will be defined as `CActor`.

`select_child<T,U>::type` is defined as `T` if `T` is a (indirect) child class of `U`, as `U` if `U` is a (indirect) child class of `T`, or as void otherwise. For example, `select_child<CRoot,CGrunt>::type is` defined as `CGrunt`, while `select_child<CGrunt,CAgent>::type` is defined as void.

Indeed, for `pred_or::operator()` to return true, either the first or the second predicate should have returned true. Since we do not know which predicate returned true, our `output_type` is the root class of the `output_types` defined by the `Pred1` and `Pred2` predicates.

Similarly, because `Pred1::operator()` takes objects of class `Pred1::input_type`, and `Pred2::operator()` takes objects of class `Pred2::input_type`, `pred_or::operator()` must take objects that are of both class `Pred1::input_type` and class `Pred2::input_type`. This is why we need `select_child`.

Now let's define `pred_and` as shown in Listing 3. Here, the `static_cast` is justified because C++ always evaluates the left side of `operator&&` first, and then evaluates the right side only if the left side was true — and we know that if an object passes `Pred1`, it is of class `Pred1::output_type`.

Let's extend the definition of `HasArmor` to define `input_type` and `output_type` as required:

```
struct HasArmor {
   typedef CActor input_type;
   typedef input_type output_type;
   bool operator()( const input_type* pObj ) const {
      return pObj->HasArmor();
   }
};
```

Now, if we have a set of objects of class `CRoot`, we can do something like this (assuming we have defined a function template `And` similar to the function template `Or`):

```
X( objects, And(IsActor(),HasArmor()), Attack() );
```

We can even combine `pred_and` and `pred_or` in an even more complex predicate expression:

```
X( objects, And(IsActor(),Or(HasArmor(),HasWeapon())),
   Attack() );
```

To complete our set of complex predicates, let's define `pred_not`. Because not satisfying a predicate does not give us any additional information about an object, `pred_not::output_type` is the same as its `input_type`:

```
template <class Pred>
struct pred_not {
   typedef typename Pred::input_type input_type;
   typedef input_type output_type;
   Pred pred;
   pred_not( Pred p ): pred(p) {
   }
   bool operator()( const input_type* pObj ) const {
      return !pred(pObj);
   }
};
```

Besides being powerful, the complex predicates we defined are also type-safe. Consider the following example:

```
X( objects, Or(IsActor(),HasArmor()), Attack() );
```

If used with our class hierarchy, the above example will not compile. This is because if an object passes the predicate, it may or may not be of class `CActor`, and the compiler will generate a type mismatch error when trying to call `HasArmor::operator()`. However, if we used `And` instead of `Or`, there would be no compile error due to the `static_cast` in `pred_and::operator()`.

Predicate Expressions

So far, our predicate system is pretty powerful, but nested predicate expressions are not fun. We need to be able to use a more natural syntax. For example, instead of

```
X( objects, And(IsActor(),HasArmor()), Attack() );
```

we want to be able to write:

```
X( objects, IsActor() && HasArmor(), Attack() );
```

The obvious solution to this problem is to overload the operators we need for predicates. For the system to work, we need to overload the operators in a way that can be used for any predicates, including custom predicates defined further in our project.

However, if we define `operator&&` the way we earlier defined the `Or` function template, it would be too ambiguous — we need a definition that the compiler will consider for predicates only. To achieve this, we need some mechanism to distinguish between a predicate and any other type. One way of doing this is to have all our predicates inherit from this common class template:

```
template <class T>
struct expr_base {
   const T& get() const {
      return static_cast<const T&>(*this);
   }
};
```

For example, let's define `pred_and` like this:

```
template <class Pred1,class Pred2>
struct pred_and: public expr_base<pred_and<Pred1,Pred2> >{
...
};
```

With this trick, we can overload `operator&&` like so:

```
template <class Pred1,class Pred2>
pred_and<Pred1,Pred2>
operator&&( const expr_base<Pred1>& p1, const expr_base<Pred2>& p2 ) {
   return pred_and<Pred1,Pred2>(p1.get(),p2.get());
}
```

Following this pattern, we can overload operator `||` and operator `!`. Now we can build Boolean predicate expressions that follow the natural C++ syntax, while also taking advantage of the operator precedence defined by the language.

This technique of building an expression tree through operator overloads is commonly known as Expression Templates (see Veldhuizen in For More Information).

Numerical Predicates

Predicates are usually defined as Boolean functions, but we can extend our definition of predicate to include numerical predicates. This is useful for exposing non-Boolean properties of objects. For example:

```
struct Health {
   typedef CActor input_type;
   typedef input_type output_type;
   int operator()( const input_type* pObj ) const {
      return pObj->GetHealth();
   }
};
```

Of course, the predicate version of the `X` template function treats all predicates as Boolean. We can use the `Health` predicate directly, but then we would only be able to check if the health of an actor is not 0 (or we can check for 0 if we use `pred_not`). Obviously, we need to be able to check for other values as well.

So, we can define the following predicate:

```
template <class Pred,class Value>
struct pred_gt: public expr_base<pred_gt<Pred,Value> > {
   typedef typename Pred::input_type input_type;
   typedef typename Pred::output_type output_type;
   Pred pred;
   Value value;
   pred_gt( Pred p, Value v ): pred(p),value(v) {
   }
   bool operator()( const Pred::input_type* pObj ) const {
      return pred(pObj)>value;
   }
};
```

Similarly to the case of `pred_or`, `pred_and`, and `pred_not`, we can overload the `>` operator to provide access to `pred_gt`:

```
template <class Pred,class Value>
pred_gt<Pred,Value>
operator>( const expr_base<Pred>& p, Value v ) {
   return pred_gt<Pred,Value>(p.get(),v);
}
```

Now, we can do something like:

```
X( grunts, Health()>5, Attack() );
```

This will execute the `Attack` functor only on the objects with health greater than 5. As any other predicate that defines `input_type` and `output_type`, we can combine `pred_gt` in complex predicate expressions. For example:

```
X( objects, IsGrunt() && (Health()>5 || HasArmor()), Attack() );
```

Following this pattern, we can overload all other comparison operators: `<`, `>=`, `<=`, `==`, and `!=`.

Additional Set Operations

The predicate expression system we just described is the core of our scripting support, but we still need to write some additional functions to make it easy to work with sets. Table 1 shows some additional function templates that we may find useful to define.

In addition, it is convenient to define operator functions for set intersection (*,*=), set union (+,+=), and set difference (-,-=). All of these functions can be defined as templates for maximum flexibility.

TABLE 1. Some additional functions that enhance usability of sets.

`All(set,predicate)`	Returns a set that contains all the elements of the input set that satisfy the predicate
`Num(set)`	Returns the number of elements in the set
`Num(set,predicate)`	Returns the number of the elements in the set that satisfy the predicate
`Any(set)`	Returns true if the set contains at least one element, false otherwise
`Any(set,predicate)`	Returns true if the set contains at least one element that satisfies the predicate
`FirstFew(set,count)`	Returns a set that contains the first count elements of the input set
`FirstFew(set,predicate,count)`	Returns a set that contains the first count elements of the input set that satisfy the predicate
`Some(set,count)`	Returns a set that consists of count random elements from the input set
`Some(set,predicate,count)`	Returns a set that consists of count random elements from the input set that satisfy the predicate

Integration with the Game

We have a powerful system to manipulate sets of objects, but to be able to do anything with it, we need to define the interface between the script and the game. For example, it could be appropriate to define a class that is the root of all scripts:

```
class CScriptBase {
public:
   void RegisterObject( CRoot* pObject );
   void RemoveObject( CRoot* pObject );
   virtual void Tick( float deltaTime )=0;
   ...
protected:
   set<CRoot*> m_Objects;
   set<CRoot*> CheckArea( const char* areaName );
   ...
};
```

The public section of our class contains functions that the game can execute. `RegisterObject` is called from the constructor of `CRoot` whenever a game object is created. The task of `RegisterObject` is to filter out any objects we do not want the script to have access to, and include all other objects in the `m_Objects` set which is accessible by the script. Similarly, `RemoveObject` is called from the destructor of `CRoot` to make sure `m_Objects` does not contain pointers to invalid objects. The game also calls `Tick` on each frame to let the script do its job. We can continue along these lines, but obviously there is not much the game has to know about the script.

The protected section of our class has functions that the child script classes can use to query the game for information. All such functions are implemented by `CScriptBase`. In our example, `CheckArea` will check a named area in the level for any objects and return a set that contains them. Once the script has the set, it can use the predicate system to get the information it needs. For example, to check if the player has advanced to a given area, we can do something like this:

```
if( Any(CheckArea("Area1"),IsPlayer()) )
   SignalPlayerIsInArea1();
```

Besides query functions, it is also useful to define functions that make it easier for the script to perform common tasks. This could include, for example, the ability to register a member function of the script for automatic periodic execution.

Defining Additional Templates

To operate properly, our predicate system depends on the `select_root` and `select_child` templates. The C++ language does not provide direct support for something like that, but we can trick the compiler into doing what we need with some meta programming.

We will need to associate a numerical identifier with each class of our class hierarchy. To do this, we can define the following templates:

```
template <class T>
struct get_id {
   enum {value=0};
};
template <int ClassID>
struct get_type {
   typedef void type;
};
```

To associate identifiers with classes, we simply define explicit specializations of `get_id` and `get_type`. For example:

```
template<>
struct get_id<CGrunt> {
   enum {value=30};
};
template<>
struct get_type<30> {
   typedef CGrunt type;
};
```

Now, `get_id<CGrunt>::value` evaluates to 30, and `get_type<30>::type` evaluates to `CGrunt`.

In addition, let's define the following template:

```
template <class T>
struct tag {
   typedef char (&type)[get_id<T>::value];
};
```

For a given class `T`, this template defines a reference to a char array the size of the numerical identifier associated with `T`.

Finally, to continue our `CGrunt` example, let's declare the following function:

```
tag<CGrunt>::type caster( const CGrunt*,const CGrunt*);
```

Note that we only declare this function. We do not provide a definition.

For `select_root` to work, all of the classes in our hierarchy must be properly registered by providing explicit specialization of `get_id` and `get_type`, plus a declaration of the caster function. This is best done using a macro:

```
#define REGISTER_CLASS(CLASS,CLASSID)\
   template<> struct get_id<CLASS> { enum {value=CLASSID}; };\
   template<> struct get_type<CLASSID> { typedef CLASS type; };\
   tag<CLASS>::type caster( const CLASS*,const CLASS*);
```

Now let's register `CRoot`, `CActor`, `CGrunt`, and `CAgent` by invoking the `REGISTER_CLASS` macro, using a different numerical identifier for each class:

```
REGISTER_CLASS(CRoot,10)
REGISTER_CLASS(CActor,20)
REGISTER_CLASS(CGrunt,30)
REGISTER_CLASS(CAgent,40)
```

LISTING 4. Defining `select_child`.

```
template <class T,class U>
struct select_child
{
   typedef
      typename meta_if<
         get_id<typename
select_root<T,U>::type>::value==get_id<T>::value,     //if
         U,                                           //then
         typename meta_if<                            //else
            get_id<typename
select_root<T,U>::type>::value==get_id<U>::value,     //if
            T,                                        //then
            void                                      //else
      >::type>::type type;
};
```

With the classes properly registered, the `select_root` template can be defined like this:

```
template <class T,class U>
struct select_root {
   enum { ClassID=sizeof(caster((T*)0,(U*)0)) };
   typedef typename get_type<ClassID>::type type;
};
```

Now let's follow what happens when we use `select_root` with `CGrunt` and `CAgent` (this is all done at compile time):

```
typename select_root<CGrunt,CAgent>::type* pObj;
```

We have invoked the `REGISTER_CLASS` macro for `CRoot`, `CActor`, `CGrunt`, and `CAgent`. As a result, now we have the following function declarations:

```
tag<CRoot>::type caster( const CRoot*,const CRoot*);
tag<CActor>::type caster( const CActor*,const CActor*);
tag<CGrunt>::type caster( const CGrunt*,const CGrunt*);
tag<CAgent>::type caster( const CAgent*,const CAgent*);
```

In our `select_root` template, we define `ClassID` as the size of the type returned by `caster((T*)0,(U*)0)`. In our example, `T` is `CGrunt` and `U` is `CAgent`. Since we have not declared a version of the caster function that takes `CGrunt*` and `CAgent*`, the compiler automatically picks the best match from the ones we did declare, which is:

```
tag<CActor>::type caster( const CActor*,const CActor*);
```

This defines `ClassID` as `sizeof(tag<CActor>::type)`. If you recall how the tag template was defined, `tag<CActor>::type` is a reference to a `char` array of size `get_id<CActor>::value`. Since `get_id<CActor>::value` is 20, `select_root<CGrunt,CAgent>::ClassID` will also be 20. Now we simply use `get_type<ClassID>::type` to retrieve the class the number 20 identifies, which is `CActor`.

So, if we return back to our example,

```
typename select_root<CGrunt,CAgent>::type* pObj;
```

`pObj` will be defined as pointer to object of class `CActor`.

And finally, Listing 4 shows how we can define `select_child`. Here, `meta_if` is a template that's commonly used for meta programming. I'll skip its definition, but assume that `meta_if<CONDITION,T,U>::type` is defined as `T` if the condition is nonzero, and as `U` otherwise.

The implementation of `select_root` and `select_child` could be simplified if C++ had compile-time `typeof()` fuctionality. The emulation of `typeof()` through type registration used here was first discovered by Bill Gibbons (see For More Information).

Safety and Performance Considerations

One of the most important features of our scripting system is that it is type-safe. Thanks to the `input_type` and `output_type` each predicate defines, the compiler knows the class of the objects that pass a given predicate expression and will issue error messages if we try to use incompatible functors with

them. Predicate expressions that make no sense — for example, `IsGrunt() && IsAgent()` — will not compile. In addition, predicate expressions will benefit from the compiler's expression short-circuit logic.

To further improve safety, we can avoid using pointers in our sets. In this case we can convert back to pointers just before we call functors and predicates from our `X` function (or any other helper function that works with sets). Note that only one conversion to pointer per object occurs, regardless of how complex the predicate expression we use is.

We can further separate the script and the game code by putting them in their own namespaces. Thus we can control what to hide from the script, and what to expose by providing functors and predicates.

Most of today's compilers will optimize any of our predicate expressions to inline code as if a programmer wrote a custom if statement to check for the condition of the predicate. This means that the performance of our scripting system depends mostly on the implementation of `std::set` and the iterator classes it defines.

The C++ standard mandates that the iterators of `std::set` are not invalidated when adding or removing elements from the set, which usually means that each element of the set is allocated as individual heap block. Because the elements of our sets are simply pointers, this translates to a waste of memory, increased heap fragmentation, and overall slow processing of `std::set`s due to cache misses. In addition, copying a `std::set` of pointers is a relatively slow and heap-intensive operation.

Even if we do not do anything to speed the system up, our scripts will most likely execute faster than if we used an interpreted scripting language. Still, we can speed them up significantly by using a custom allocator with the std::set class template, or by designing our own, faster set container.

Example Source Code

The source code available for download at www.gdmag.com uses the class hierarchy from Listing 1 to demonstrate the ideas discussed in this article, but it is a bit more complex because it has added support for const and volatile type modifiers. The code has been tested and is compatible with Visual C++ version 6 and 7, but should be compatible with most of today's C++ compilers.

FOR MORE INFORMATION

T. Veldhuizen. "Expression Templates." *C++ Report* Vol. 7, No. 5, (June 1995) www.osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html

B. Gibbons. "A Portable typeof Operator" www.accu-usa.org/2000-05-Main.html

ACKNOWLEDGEMENTS

I would like to thank Peter Dimov for his help in implementing and further enhancing the predicate expressions system described in this article.