ast month we began describing the class which will act as the basis for all of the chess pieces we have at our disposal. If you remember, I also mentioned something called an abstract class. This month I'll show you why an abstract class can be helpful in defining a specialized class hierarchy, and just how one goes about defining an abstract class.

he Next Step

So far we've defined the methods we need to describe each and every chess piece on the chess board. Without writing any code, let's quickly add some of the other general functions that will be necessary to describe any chess piece. For instance, each piece has a distinct way it can move on the chess board: a knight moves one space in one direction and two spaces in the perpendicular direction. Since each of our CChessPiece objects has a position on the board (the mPiecePosition data member), we can ask a chess piece if it can move to another position. We will define another function called ValidMove which specifies whether or not a chess piece can move to a spot on the board based on its current position. When we ask ourselves how such a function should reply (i.e. what is its return value?), we must realize that ValidMove gives us a yes or no answer. Yes, I can move there. No, I can't move there. We can easily match a standard C++ data type to that kind of answer: bool (or Boolean in some compilers).

```
Boolean
CChessPiece::ValidMove(SBoardPosition inToWhere)
```

So now we've added another function to our class. Now, on to the abstract!

bstraction

About this abstract thing. What we've been defining so far in the CChessPiece class is an abstract chess piece. The functions we add are generalized to apply to any kind of piece. However, there is no "general piece" that exists in real life (yeah, a king is commander in chief, but he's still no general!). In other words, if we can't actually have a generalized chess piece in real life, why should we really be able to create one on our virtual chess board? An abstract class allows us to define the functionality of a set of child classes, but it is set up in such a way that a program cannot use the abstract base class.

Creating abstract classes depends on the use of virtual class functions. There are two options for the functions in a class hierarchy. Functions can be inherited by child classes as-is, or they can be inherited and re-implemented by the child class. Any function which will be inherited as-is is in its final form in the parent class; calling that method for that class and every child class uses that function. Consider our ValidMove method. We know that every child class will have a ValidMove function; however, we also know that every kind of chess piece has a different pattern to its movement. So each child class will have to define its own ValidMove method. In other words, we know that each child class must have a ValidMove method, and that the ValidMove function in CChessPiece is not the same function which the child classes will use. Thus, ValidMove will be defined as virtual.

```
class CChessPiece

  public:
                                CChessPiece();
    virtual                     ~CChessPiece();

    virtual SChessPieceType GetChessPieceType();

    virtual Boolean           ValidMove(SBoardPosition inToWhere);

  protected:
    SBoardPosition             mPiecePosition;
;
```

That's the first step in developing a class hierarchy: decide what functions should be inherited as-is and which will be re-implemented in subsequent classes. The CChessPiece definition above shows this.

Finally, in creating an abstract class, we need to make sure a program cannot use it; only subclasses of CChessPiece are usable. How do we accomplish this? The answer will probably make you say "duh!" In our current definition of CChessPiece the constructor function is marked public.

What if we change it to private? Remember, private members of a class can only be accessed by that class. So if the constructor is private, no piece of programming code can actually create an instance of that class!

he New CChessPiece

Now that we know about abstract classes, our generalized chess piece looks like this:

```
class CChessPiece

  public:
    virtual                    ~CChessPiece();

    virtual SChessPieceType GetChessPieceType();

    virtual Boolean            ValidMove(SBoardPosition inToWhere);

  protected:
    SBoardPosition             mPiecePosition;

  private:
                               CChessPiece();
;
```

Now that we have a "template" to the creation of our class hierarchy, we can begin to code for individual chess pieces. Next month...

Jeff Frey
jeff@applewizards.net