Polymorphism Group 16

Holly Buteau - David Gluckman - Meredith Kiessling - Desmond Lau - Brett Naylor - Jerrett Quella - Benjamin Tate - Rachel Williams - Emma Yaffe

Polymorphism is a powerful feature of C++ in regards to class inheritance. A commonly used feature of the language is the ability to create classes and subclasses. One can easily see how a subclass might need to modify the information of the parent class. If there is a geometric shape class, how do you decide the number of sides the shape has, or the equation for the area? Different shapes have different features. Polymorphism allows the code to behave differently based on the type of object that calls a function.

Polymorphism uses the keyword “virtual” which prefaces any functions to be redefined in derived classes. This signals to the compiler to bind at run-time (e.g. dynamic binding), so it will call the function specific to the class of the object type calling it, even if base class pointers are used to call the function. Otherwise the compiler binds at compile time (e.g. static binding), thus causing the program to ignore any additional data specific to the derived class. This allows virtual member functions to be redefined in a derived class. Alternatively, virtual member functions can have no definition in the parent class, and *only* be defined in derived classes. These pure virtual functions would not be possible without polymorphism.

Polymorphism is important because it offers a solution to something called the slicing problem. The slicing problem can be thought of as a “partial assignment” of objects. For example, when a derived object is assigned by value to a base class object, any information specific to the derived class is lost, or “sliced off”. Since the derived class inherits from the base class, this is not a problem the other way around: the derived class has access to all the base class members. The compiler does not know to look in the derived class when you pass a derived type into a base type variable.

It is important to avoid the slicing problem because the information in derived classes is ignored or lost when it has been sliced. Any information or behavior specific to the derived class is not utilized after slicing. Members of derived class objects will not be accessed, which can lead to errors. Although this can be avoided, it also has the potential to cause unexpected behavior, and it can make the code difficult for future programmers to work with because they might unknowingly encounter slicing. It is important to note that while polymorphism provides a solution to the slicing problem, it does not eliminate it entirely. If an object of the parent class is assigned the values of an object of a derived class, the parent object still will not see the functions of the derived class, even if they have been assigned as virtual. This can be avoided through the use of pointers, by making a pointer of the parent class point to an instance of the derived class you should be able to maintain functionality. So, while polymorphism reduces the possibility of slicing, it is still an issue that the programmer should be aware of and testing for.

In addition to avoiding the slicing problem, polymorphism is useful in other ways. Polymorphism also allows for more generic code. Because derived classes can all contain functions with the same name, the interface can be the same for each even if the object type differs. Going back to the shape example, if you change a square into a parallelogram, you do not have to change every function call; you just have to change the type of object. This helps streamline the process of writing code and also reduce errors. By creating more generalized code, it becomes easier to apply to both changes in existing code and in new code. If you create a new subclass, all you have to do is define the relevant virtual functions, and call them the same way you call all the other functions: with the polymorphic base function. It also makes your code easier to maintain by requiring fewer changes each time a class is updated or added. There are also benefits of polymorphism that come from using pointers to a base class to house pointers to a derived class. We saw this in Lab 5 with an array of pointers to Critter was able to hold pointers to either Ant or Doodlebug objects. This flexibility allows run-time assignment of whichever type of derived (or base, in the case of non-abstract base classes) object is necessary on the fly.

For all of these reasons—avoiding the slicing problem, creating generic code that is easy to update and add to, increasing the flexibility of pointers—polymorphism is a very important and useful principle of C++. We expect to find new uses for it as we learn more and can already see where it would have been helpful in past work in this course as well as in CS 161.