Type erasure and reification - Eli Bendersky's website  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
 Toggle navigation  
  
  
  
  
  
 Eli Bendersky's website  
  
  
  
  
  
  
  
About  
  
  
  
  
  
 Projects  
  
  
  
  
  
 Archives  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Type erasure and reification  
  
  
  
  
  
  
  
  
  
December 05, 2018 at 05:10  
  
 Tags  
  
Programming , C & C++ , Python  
  
  
  
 In this post I'd like to discuss the concepts of type erasure and reification in programming languages. I don't intend to dive very deeply into the specific rules of any particular language; rather, the post is going to present several simple examples in multiple languages, hoping to provide enough intuition and background for a more serious study, if necessary. As you'll see, the actual concepts are very simple and familiar. Deeper details of specific languages pertain more to the idiosyncrasies of those languages' semantics and implementations.  
  
Important note: in C++ there is a programming pattern called type erasure , which is quite distinct from what I'm trying to describe here [1] . I'll be using C++ examples here, but that's to demonstrate how the original concepts apply in C++. The programming pattern will be covered in a separate post.  
  
 Types at compile time, no types at run-time  
  
The title of this section is a "one short sentence" explanation of what type erasure means. With few exceptions, it only applies to languages with some degree of compile time (a.k.a. static ) type checking. The basic principle should be immediately familiar to folks who have some idea of what machine code generated from low-level languages like C looks like. While C has static typing, this only matters in the compiler - the generated code is completely oblivious to types.  
  
For example, consider the following C snippet:  
  
 When compiling the function extract , the compiler will perform type checking. It won't let us access fields that were not declared in the struct, for example. Neither will it let us pass a pointer to a different struct (or to a float ) into extract . But once it's done helping us, the compiler generates code which is completely type-free:  
  
 The compiler is familiar with the stack frame layout and other specifics of the ABI, and generates code that assumes a correct type of structure was passed in. If the actual type is not what this function expects, there will be trouble (either accessing unmapped memory, or accessing wrong data).  
  
A slightly adjusted example will clarify this:  
  
 The compiler will generate exactly identical code from this function, which in itself a good indication of when the types matter and when they don't. What's more interesting is that extract\_cast makes it extremely easy for programmers to shoot themselves in the foot:  
  
 In general, type erasure is a concept that descibes these semantics of a language. Types matter to the compiler, which uses them to generate code and help the programmer avoid errors. Once everything is type-checked, however, the types are simply erased and the code the compiler generates is oblivious to them. The next section will put this in context by comparing to the opposite approach.  
  
  
  
Reification - retaining types at run-time  
  
While erasure means the compiler discards all type information for the actual generated code, reification is the other way to go - types are retained at run-time and used for perform various checks. A classical example from Java will help demonstrate this:  
  
 This code creates an array of String , and converts it to a generic array of Object . This is valid because arrays in Java are covariant , so the compiler doesn't complain. However, in the next line we try to assign an integer into the array. This happens to fail with an exception at run-time :  
  
 A type check was inserted into the generated code, and it fired when an incorrect assignment was attempted. In other words, the type of objects is reified . Reification is defined roughly as "taking something abstract and making it real/concrete", which when applied to types means "compile-time types are converted to actual run-time entities".  
  
C++ has some type reification support as well, e.g. with dynamic\_cast :  
  
 We can call call\_derived thus:  
  
 The first call will successfully invoke derivedfunc ; the second will not, because the dynamic\_cast will return nullptr at run-time. This is because we're using C++'s run-time type information (RTTI) capabilities here, where an actual representation of the type is stored in the generated code (most likely attached to the vtable which every polymorphic object points to). C++ also has the typeid feature, but I'm showing dynamic\_cast since it's the one most commonly used.  
  
Note particularly the differences between this sample and the C sample in the beginning of the post. Conceptually, it's similar - we use a pointer to a general type (in C that's void\* , in the C++ example we use a base type) to interact with concrete types. Whereas in C there is no built-in run-time type feature, in C++ we can use RTTI in some cases. With RTTI enabled, dynamic\_cast can be used to interact with the run-time (reified) representation of types in a limited but useful way.  
  
  
  
Type erasure and Java generics  
  
One place where folks not necessarily familiar with programming language type theory encounter erasure is Java generics, which were bolted onto the language after a large amount of code has already been written. The designers of Java faced the binary compatibility challenge, wherein they wanted code compiled with newer Java compilers to run on older VMs.  
  
The solution was to use type erasure to implement generics entirely in the compiler. Here's a quote from the official Java generics tutorial :  
  
 Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:  
  
 Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.  
  
Insert type casts if necessary to preserve type safety.  
  
Generate bridge methods to preserve polymorphism in extended generic types.  
  
  
  
Here's a very simple example to demonstrate what's going on, taken from a Stack Overflow answer . This code:  
  
 Uses a generic List . However, what the compiler creates prior to emitting bytecode is equivalent to:  
  
 Here List is a container of Object , so we can assign any element to it (similarly to the reification example shown in the previous section). The compiler then inserts a cast when accessing that element as a string. In this case the compiler will adamantly preserve type safety and won't let us do list.add(5) in the original snippet, because it sees that list is a List<String> . Therefore, the cast to (String) should be safe.  
  
Using type erasure to implement generics with backwards compatibility is a neat idea, but it has its issues. Some folks complain that not having the types available at runtime is a limitation (e.g. not being able to use instanceof and other reflection capabilities). Other languages, like C# and Dart 2, have reified generics which do preserve the type information at run-time.  
  
  
  
Reification in dynamically typed languages  
  
I hope it's obvious that the theory and techniques described above only apply to statically-typed languages. In dynamically-typed languages, like Python, there is almost no concept of types at compile-time, and types are a fully reified concept. Even trivial errors like:  
  
 Fire at run-time, because there's no static type checking [2] . Types obviously exist at run-time, with functions like type() and isinstance() providing complete reflection capabilities. The type() function can even create new types entirely at run-time .  
  
  
  
  
  
[1] But it's most likely what you'll get to if you google for "c++ type erasure".  
  
  
  
  
  
 [2] To be clear - this is not a bug; it's a feature of Python. A new method can be added to classes dynamically at runtime (here, some code could have defined a joe method for Foo before the f.joe() invocation), and the compiler has absolutely no way of knowing this could or couldn't happen. So it has to assume such invocations are valid and rely on run-time checking to avoid serious errors like memory corruption.  
  
  
  
  
  
  
  
 For comments, please send me an email .  
  
  
  
  
  
  
  
  
  
 © 2003-2024 Eli Bendersky  
  
 Back to top