The Truth about Type Safety in Programming Languages

Gene Michael Stover

created 12 March 2004
updated 14 April 2004

Contents

1 Introduction **1**

2 Two Kinds of Type Safety **1**

3 Summary of Languages **3**
3.1 Notes about that table 3

4 Conclusion **4**

A Pencils & Battleshipts **4**

B Manifest Typing **4**

C Other File Formats **5**

1 Introduction

"C++ is a type-safe language, & Lisp isn't." I used to believe that until Brian T. Rice, a Lisp programmer in Seattle, explained the truth to me. Here's what he said, in my own words.

2 Two Kinds of Type Safety

When programmers talk about type safety, they usually mean *compile-time* type safety. C++ (& C, Java, Ada, Pascal, & lots of other languages) have that.

Languages that don't have compile-time type safety include Lisp, Smalltalk, Perl, old-fashioned BASIC[1], Self, & others.

There is also run-time type safety. It is no less important than compile-time type safety, but programmers discussing type safety don't often think of run-time type safety.

C & C++ don't have run-time type safety. In either of those languages, you can (attempt to) convert any type to any other. Here are some examples:

You can convert an integer to a pointer:

```
/*
 * example of unsafe run-time type-cast in C
 */
void
foo () {
  int i;
  void *p;

  p = (void *) i;
}
```

You can convert between Pencils, Battleships, & Vectors:

```
/*
 * example of unsafe type-conversion
 */
class Pencil {
public:
  virtual void erase ();

  /* other members (functions & data)
   * might go here */
};

class Battleship {
public:
  virtual void peel ();

  /* other members might go here */
};

void
UnsafePencil2Battleship (Pencil const *pencil,
                         Battleship *battleship)
{
```

[1] If you remember BASIC in which every line was numbered, you remember old-fashioned BASIC.

```
  /* BAD CODE: This is an unsafe & horrible thing
   * to do, & it would probably fail.  It's just
   * wrong! */
  memcpy (battleship, pencil, sizeof *battleship);
}

void
UnsafeBattleship2VectorOfInt (Battleship const *battleship,
                              std::vector<int> *v)
{
  /* BAD CODE: This is a type-conversion error
   * & a nightmare.  Don't do this. */
  memcpy (v, battleship, sizeof *v);
}
```

I can hear the C++ programmers grimmacing now, maybe even shouting "That's crap code! It won't work! Only a horrible programmer would write it!" Nevertheless, the language allows it. The language is not type-safe at run-time.

A language that has strong run-time type safety would not allow such a conversion. In Java, I guess you could try to cast a Pencil reference to a Battleship reference, but the system would toss a type-error exception at you. In Lisp, there isn't any notation to do such a terrible thing at all.

3 Summary of Languages

Here's a table that relates some languages to their compile-time type safety & run-time type safety.

language	compile-time?	run-time?
Ada	yes	yes*
assembly	yes*	no
C	yes	no
C++	yes	no
Java	yes	yes
Lisp	no	yes
Pascal	yes	yes*
Perl	no	sort of*
Self	no	yes
Smalltalk	no	yes

3.1 Notes about that table

Pascal I don't think traditional, true Pascal keeps much type information at run-time, but it doesn't offer any way to do unsafe conversions. Combined with its strong compile-time type safety, I think gives Pascal type safety at run-time by default.

Ada I believe – but I'm not sure – that Ada is like Pascal with respect to type safety. Ada might keep more type information at run-time.

assembly It depends on your assembler & your processor. Last I used Microsloth 80x86 Assembler (MASM), it actually had some compile-time type safety features. Like it would warn you if you tried to CALL a label in the data segment. It's clearly very weak type safety, but it is a form of type safety.

Perl Perl's type system is interesting. It tries to figure out what type you need & what type you have & do a conversion, & I think the conversions often go through a string form. (The string form might even be explicit in the language specification.) So Perl *almost* has a form of manifest typing (Section B). It isn't manifest typing, & it's a stretch to call it manifest typing, but it's sort of kind of almost somewhat reminiscent of manifest typing.

4 Conclusion

Type safety is not a single dimension. It has at least two dimensions: compile-time & run-time. Ironically, two languages which are considered type-safe (C & C++) have no type safety at run-time.

The lack of type safety in C & C++ is probably useful for low-level programming, bit-twiddling. That's what C is for. It's a dangerous feature of a language that's used to write large applications.

A Pencils & Battleshipts

A friend told me about an article he had read about type-safety & collections. Maybe it was about parameterized types in general in C++. The article's author had said there would never be a need for a collection of pencils & battleships.

I pointed out that the world is a collection of pencils, battleships, & some other things.

B Manifest Typing

If a programming language gives types to values, not to variables, we say that programming language uses *manifest typing*. Other programming languages do not use manifest typing. (I don't know what their type systems are called, other than "not manifest typing".)

Lisp uses manifest typing. In Lisp, variables do not have types. Any variable can hold any type of value. Values do have types.

C & Ada do not use manifest typing. In C & Ada, variables have types & are allowed to hold only values of their same types.

Notice that in a purely object oriented language in which there is one class hierarchy with one root, that OO type system effectively becomes manifest typing. So if a programming language requires all values to be objects, & if all classes are direct or indirect classes of a single root class, that language offers manifest typing as a side-effect. Smalltalk is one such language.

In *ANSI Common Lisp* ([Gra96,)], Paul Graham says that Lisp is manifestly typed because values have types but variables do not.

There is a brief corroborative definition of manifest typing at http://www.lisp.org/table/types.htm.

There is yet another brief definition of manifest typing at http://www.cs.auc.dk/~normark/prog3-02/html/notes/fu-intr-1_themes-type-section.html.

According to the definition of manifest typing on Wiki[2], manifest typing has the opposite definition I've given here. Maybe the Lisp community uses the term one way & the non-Lisp programming community uses it the other way. Maybe this suggests that we should use a different term for these concepts.

Webster's Ninth Collegiate Dictionary didn't decide the issue for me. Here are the three definitions in that volume for "manifest":

1. *adj.* 1 : readily perceived by the senses & esp. by the sight. 2 : easily understood or recognized by the mind : OBVIOUS

2. *vt* : to make evident or certain by showing or displaying

3. *n* : MANIFESTATION, INDICATION 2 : MANIFESTO 3 : a list of passengers or an invoice of cargo for a ship or plane

These definitions don't help much. The first two might suggest that types known at compile-time are manifest because they are easily recognized & evident when a programmer reads the source code. The first two definitions could equally be applied to type systems such as Lisp's because the types of values are easily recognized & evident to the run-time system whereas types declared at compile-time are implicit at run-time.

The third definition, that of a manifest as a list of goods, could be applied to compile-time type systems because the declarations of variables & their types could be interpreted as a manifest, but the information about values & their types at run-time is a manifest for the run-time system.

I'd say we need new terms for the two kinds of type systems. Maybe "compile-time types" & "run-time types".

C Other File Formats

This document is available in multi-file HTML format at http://lisp-p.org/tat/.

This document is available in DVI format at http://lisp-p.org/tat/tat.dvi.

This document is available in PostScript format at http://lisp-p.org/tat/tat.ps.

[2] http://c2.com/cgi/wiki

References

[Gra96] Paul Graham. *ANSI Common Lisp*. Prentice Hall, Englewood Cliffs, New Jersey 074623, USA, 1996. ISBN 0-13-370875-6.