Drinking the Tears of D's Competitors

-or-

Implicit Conversion of Template Instantiations

by Walter Bright
Dlang.org
March 2024
https://twitter.com/WalterBright

I was going to do a presentation on pattern matching.

But then, I realized that pattern matching depends on sum types:

https://github.com/WalterBright/DIPs/blob/sumtypes/DIPs/1NNN-(wgb).md

And sum types proved to be controversial:

https://www.digitalmars.com/d/archives/digitalmars/D/sumtypes_for_D_366242.html

But one thing stood out in those discussions – a long requested feature was improving the ability to create template types that were as good as builtin types.

Implicit Conversions

```
void moon1(const(int)[ ]);
void moon2(const int[ ]);

void sun()
{
    const(int)[ ] a;
    moon1(a);   // works
    moon2(a);   // works

    const int[ ] b;
    moon1(b);   // works
    moon2(b);   // works
}
```

Trying It With a Template

```
struct X(T) { T[ ] t; }

void moon1( X!(const int) );
void moon2( const X!int );

void sun()
{
    X!(const int) a;
    moon1(a);   // works
    moon2(a);   // fails

    const X!int b;
    moon1(b);   // fails
    moon2(b);   // works
}
```

How Do We Solve This?

First Suggestion

Make X!(const T) and const(X!T) the same type

Danger, Will Robinson!

```
struct X(T)
{
    T t;
    void bar(T);
}
```

If X is instantiated with X!(const int* p), bar becomes void bar(const int*).
But const X!(int*) will instantiate bar as void bar(int*).
The function parameter types are different!

Structural Non-Conformance

```
struct X(T)
{
    static if (is(T == const(T))
    {
        int a;
    }
    T t;
}
```

So That Isn't Going To Work

But const(int)[ ] and const(int[ ])
are not the same type, either,
so it isn't necessary for
X!(const T) and const(X!T)
to be the same type.

Only Need Implicit Conversion With Qualifier Conversion

https://dlang.org/spec/function.html#function-overloading

Other implicit conversions will not be considered in this proposal.

Key Insight

- Implicit conversions work on builtin types
- because the top level can be converted
- because it can be trivially copied.
-
- Apply that same principle to structs/classes.

Method

- Fields
- Non-static member functions
- Ignore other members

Fields

- Match names
- Match ordering
- Match placement
- Be implicitly convertible

Non-Static Member Functions

- Match names
- Match ordering
- Address of function must be implicitly convertible (i.e. covariance like overriding functions)

If all tests pass, it is implicitly convertible!

The Beauty

- It's principled
- Follows all existing rules
- Doesn't break the type system
- Doesn't create holes in type system

Existing Code

- Will break if it relies on such conversions not compiling
- Hard to see legitimate code relying on that
- Can consider this a bug fix rather than a new feature?

Blast Wave

- Can do implicit conversions of structs/classes under most circumstances
- Nothing clever about it
- We'll see how this influences things

Tears

```
template<class T> struct X { T t; };

void moon1(const X<int>);
void moon2(X<const int>);

void sun() {
    const X<int> *a;
    moon1(*a);
    moon2(*a); // could not convert ‘* a’ from ‘const X<int>’ to ‘X<const int>’

    X<const int> *b;
    moon1(*b); // could not convert ‘* b’ from ‘X<const int>’ to ‘X<int>’
    moon2(*b);
}
```