Subject: Re: REDUCE
Posted by John-David T. Smith on Mon, 02 Apr 2001 18:17:40 GMT
View Forum Message <> Reply to Message

Richard Younger wrote:
>
>
> [SNIP]
>
>> generated code. Granted, this is a very simple example, but what I am
>> looking for is a solution which makes use of the redundancy in this code
>> to avoid generating most of it. I may be asking more out of compilers
>> than they can offer.
>
>
>
> I was thinking about this a little more, and then it hit me: You
> probably don't want templates.
>
> Templates are a means for providing ~compile time~ polymorphism. With
> compile time polymorphism, you'll always need some sort of switch()
> statement or a decision table to decide which compiled path to take at
> runtime. What you're really looking for is ~run time~ polymorphism.
>
> Run time polymorphism is provided for with function pointers in C and
> virtual functions (and parent class / derived class hierarchies) in
> C++. Now I suppose I could try to cook up a C++ example using
> virtuality, but outside of a brief exercise or two, I've never actually
> used it. So here's an example of a function pointer table. Tell me if
> you like it. No C++ required.
>
> WARNING -- large code block follows -- see bottom for more actual text.

Hi Rich:

Thanks again for the ideas. Unfortunately, this implements, albeit in a
more manageable and less ugly way, pretty much what I'd already
accomplished with macros. That is, each and every of the functions
BYTE_foo, INT_foo, ULONG_foo, etc., get compiled and included in the
executeable separately, and you essentially choose among them with the
run-time type information. This works, but leads to extreme code bloat
if you're replicating a large function 10's of times.

So far, all these solutions have succeeded in allowing the program to
branch to some other portion of its code, depending on the type. I was
looking for a solution in which the executeable could use the exact same
piece of compiled code to accomplish the calculation with a variety of
types. That is, run-time, *in-place* type polymorphism. One sneaky way

of reducing this duplication is to perform in place casting only where you need it. Here's an example which you can try, in which the variable "tmp" is used in many ways:

```
#include <stdlib.h>
#include <stdio.h>
#include "/usr/local/rsi/idl/external/export.h"

main() {
  int a=5;
  float b=6.;
  IDL_ULONG64 c=1000000;
  void *tmp=malloc(sizeof(IDL_ULONG64));

  *(int *)tmp=a;

  printf("GOT %d\n",*(int *)tmp);

  *(float *)tmp=b;
  printf("GOT %f\n",*(float *)tmp);

  *(IDL_ULONG64 *)tmp=c;
  printf("GOT %Lu\n",*(IDL_ULONG64 *)tmp);
}
```

Now, all the code emitted by the compiler which, for example, performs the loops, can be the same. All that's required is branched code for casting your input and output arrays, and any statement which uses that cast data. I.e, you could have something like:

```
void ALL_process (void *in, void *out, int skip, int atom, int n_cdim,
    int new_nel )
{
  IDL_LONG i, j, ind, base;
  void *tmp=(void *)IDL_GetScratch(0,1,sizeof(IDL_ULONG64));
  for(i=0,base=0;i<new_nel;base+=skip) {
    for(j=0;j<atom;j++) {
      switch(type) {
      case IDL_TYP_BYTE:
 *(UCHAR *)tmp=((UCHAR *)in)[j+base];
 break;
      case IDL_TYPE_INT:
 *(short *)tmp=((short *)in)[j+base];
 break;
       .... (etc.)
      }
      for(ind=j+base;ind<j+base+atom*n_cdim;ind+=atom) {
 switch(type) {
```

```
 case IDL_TYP_BYTE:
   if( ((UCHAR *)in)[ind]>*(UCHAR *)tmp )
     *(UCHAR *)tmp=((UCHAR *)in)[ind];
   break;
 case IDL_TYPE_INT:
   if( ((short *)in)[ind]>*(short *)tmp )
     *(short *)tmp=((short *)in)[ind];
   break;
        .... (etc.)
 }
      }
      switch(type) {
      case IDL_TYP_BYTE:
 ((UCHAR *)out)[i++]=*(UCHAR *)tmp;
 break;
      case IDL_TYPE_INT:
 ((short *)out)[i++]=*(short *)tmp;
 break;
      .... (etc.)
      }
    }
  }
}
```

I guess it's hard to say how much of a bloat savings this is, and whether it impacts performance. Obviously, you'd need a clever way to do all those run-time branched type castings, rather than by hand. Maybe a perl script would be easier than trying to futz with the pre-processor (but less portable).

Anyway, let me know if you think of anything along these lines.

Thanks again,

JD