Expressive Diagnostics

In addition to being fast and functional, we aim to make Clang extremely user friendly. As far as a command-line compiler goes, this basically boils down to making the diagnostics (error and warning messages) generated by the compiler be as useful as possible. There are several ways that we do this. This section talks about the experience provided by the command line compiler, contrasting Clang output to GCC 4.9's output in some cases.

Column Numbers and Caret Diagnostics

First, all diagnostics produced by clang include full column number information. The clang command-line compiler driver uses this information to print "point diagnostics". (IDEs can use the information to display in-line error markup.) This is nice because it makes it very easy to understand exactly what is wrong in a particular piece of code.

The point (the green "^" character) exactly shows where the problem is, even inside of a string. This makes it really easy to jump to the problem and helps when multiple instances of the same character occur on a line. (We'll revisit this more in following examples.)

```
  $ gcc-4.9 -fsyntax-only -Wformat format-strings.c
  format-strings.c: In function 'void f()':
  format-strings.c:91:16: warning: field precision specifier '.*' expects a
matching 'int' argument [-Wformat=]
     printf("%.*d");
                ^
  format-strings.c:91:16: warning: format '%d' expects a matching 'int' argument
[-Wformat=]
  $ clang -fsyntax-only format-strings.c
  format-strings.c:91:13: warning: '.*' specified field precision is missing a
matching 'int' argument
    printf("%.*d");
              ^
```

Note that modern versions of GCC have followed Clang's lead, and are now able to give a column for a diagnostic, and include a snippet of source text in the result. However, Clang's column number is much more accurate, pointing at the problematic format specifier, rather than the `)` character the parser had reached when the problem was detected. Also, Clang's diagnostic is colored by default, making it easier to distinguish from nearby text.

Range Highlighting for Related Text

Clang captures and accurately tracks range information for expressions, statements, and other constructs in your program and uses this to make diagnostics highlight related information. In the following somewhat nonsensical example you can see that you don't even need to see the original source code to understand what is wrong based on the Clang error. Because clang prints a point, you know exactly *which* plus it is complaining about. The range information highlights the left and right side of the plus which makes it immediately obvious what the compiler is talking about. Range information is very useful for cases involving precedence issues and many other cases.

```
  $ gcc-4.9 -fsyntax-only t.c
  t.c: In function 'int f(int, int)':
  t.c:7:39: error: invalid operands to binary + (have 'int' and 'struct A')
     return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                                         ^
  $ clang -fsyntax-only t.c
  t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
    return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                         ~~~~~~~~~~~~~~ ^ ~~~~~
```

Precision in Wording

A detail is that we have tried really hard to make the diagnostics that come out of clang contain exactly the pertinent information about what is wrong and why. In the example above, we tell you what the inferred types are for the left and right hand sides, and we don't repeat what is obvious from the point (e.g., that this is a "binary +").

Many other examples abound. In the following example, not only do we tell you that there is a problem with the `*` and point to it, we say exactly why and tell you what the type is (in case it is a complicated subexpression, such as a call to an overloaded function). This sort of attention to detail makes it much easier to understand and fix problems quickly.

```
  $ gcc-4.9 -fsyntax-only t.c
  t.c:5:11: error: invalid type argument of unary '*' (have 'int')
    return *SomeA.X;
           ^
  $ clang -fsyntax-only t.c
  t.c:5:11: error: indirection requires pointer operand ('int' invalid)
    int y = *SomeA.X;
            ^~~~~~~~
```

Typedef Preservation and Selective Unwrapping

Many programmers use high-level user defined types, typedefs, and other syntactic sugar to refer to types in their program. This is useful because they can abbreviate otherwise very long types and it is useful to preserve the typename in diagnostics. However, sometimes very simple typedefs can wrap trivial types and it is important to strip off the typedef to understand what is going on. Clang aims to handle both cases well.

The following example shows where it is important to preserve a typedef in C.

```
  $ clang -fsyntax-only t.c
  t.c:15:11: error: can't convert between vector values of different size
('__m128' and 'int const *')
    myvec[1]/P;
    ~~~~~~~~^~
```

The following example shows where it is useful for the compiler to expose underlying details of a typedef. If the user was somehow confused about how the system "pid_t" typedef is defined, Clang helpfully displays it with "aka".

```
  $ clang -fsyntax-only t.c
  t.c:13:9: error: member reference base type 'pid_t' (aka 'int') is not a
structure or union
    myvar = myvar.x;
            ~~~~~ ^
```

In C++, type preservation includes retaining any qualification written into type names. For example, if we take a small snippet of code such as:

```
namespace services {
  struct WebService {  };
}
namespace myapp {
  namespace servers {
    struct Server {  };
  }
}

using namespace myapp;
void addHTTPService(servers::Server const &server,
::services::WebService const *http) {
  server += http;
}
```

and then compile it, we see that Clang is both providing accurate information and is retaining the types as written by the user (e.g., "servers::Server", "::services::WebService"):

```
  $ clang -fsyntax-only t.cpp
  t.cpp:9:10: error: invalid operands to binary expression ('servers::Server
const' and '::services::WebService const *')
    server += http;
    ~~~~~~ ^  ~~~~
```

Naturally, type preservation extends to uses of templates, and Clang retains information about how a particular template specialization (like `std::vector<Real>`) was spelled within the source code. For example:

```
  $ clang -fsyntax-only t.cpp
  t.cpp:12:7: error: incompatible type assigning 'vector<Real>', expected
'std::string' (aka 'class std::basic_string<char>')
    str = vec;
        ^ ~~~
```

Fix-it Hints

"Fix-it" hints provide advice for fixing small, localized problems in source code. When Clang produces a diagnostic about a particular problem that it can work around (e.g., non-standard or redundant syntax, missing keywords, common mistakes, etc.), it may also provide specific guidance in the form of a code transformation to correct the problem. In the following example, Clang warns about the use of a GCC extension that has been considered obsolete since 1993. The underlined code should be removed, then replaced with the code below the point line (".x =" or ".y =", respectively).

```
  $ clang t.c
  t.c:5:28: warning: use of GNU old-style field designator extension
  struct point origin = { x: 0.0, y: 0.0 };
                          ~~ ^
                          .x =
  t.c:5:36: warning: use of GNU old-style field designator extension
  struct point origin = { x: 0.0, y: 0.0 };
                                  ~~ ^
                                  .y =
```

"Fix-it" hints are most useful for working around common user errors and misconceptions. For example, C++ users commonly forget the syntax for explicit specialization of class templates, as in the error in the following example. Again, after describing the problem, Clang provides the fix--add `template<>`--as part of the diagnostic.

```
  $ clang t.cpp
  t.cpp:9:3: error: template specialization requires 'template<>'
    struct iterator_traits<file_iterator> {
    ^
    template<>
```

Template Type Diffing

Templates types can be long and difficult to read. Moreso when part of an error message. Instead of just printing out the type name, Clang has enough information to remove the common elements and highlight the differences. To show the template structure more clearly, the templated type can also be printed as an indented text tree.

Default: template diff with type elision

```
t.cc:4:5: note: candidate function not viable: no known conversion from
'vector<map<[...], float>>' to 'vector<map<[...], double>>' for 1st argument;
```

-fno-elide-type: template diff without elision

```
t.cc:4:5: note: candidate function not viable: no known conversion from
'vector<map<int, float>>' to 'vector<map<int, double>>' for 1st argument;
```

-fdiagnostics-show-template-tree: template tree printing with elision

```
t.cc:4:5: note: candidate function not viable: no known conversion for 1st
argument;
  vector<
    map<
      [...],
      [float != double]>>
```

-fdiagnostics-show-template-tree -fno-elide-type: template tree printing with no elision

```
t.cc:4:5: note: candidate function not viable: no known conversion for 1st
argument;
  vector<
    map<
      int,
      [float != double]>>
```

Automatic Macro Expansion

Many errors happen in macros that are sometimes deeply nested. With traditional compilers, you need to dig deep into the definition of the macro to understand how you got into trouble. The following simple example shows how Clang helps you out by automatically printing instantiation information and nested range information for diagnostics as they are instantiated through macros and also shows how some of the other pieces work in a bigger example.

```
  $ clang -fsyntax-only t.c
  t.c:80:3: error: invalid operands to binary expression ('typeof(P)' (aka
'struct mystruct') and 'typeof(F)' (aka 'float'))
    X = MYMAX(P, F);
        ^~~~~~~~~~~
  t.c:76:94: note: expanded from:
  #define MYMAX(A,B)    __extension__ ({ __typeof__(A) __a = (A); __typeof__(B)
__b = (B); __a < __b ? __b : __a; })

~~~ ^ ~~~
```

Here's another real world warning that occurs in the "window" Unix package (which implements the "wwopen" class of APIs):

```
  $ clang -fsyntax-only t.c
  t.c:22:2: warning: type specifier missing, defaults to 'int'
          ILPAD();
          ^
  t.c:17:17: note: expanded from:
  #define ILPAD() PAD((NROW - tt.tt_row) * 10)    /* 1 ms per char */
                  ^
  t.c:14:2: note: expanded from:
          register i; \
          ^
```

In practice, we've found that Clang's treatment of macros is actually more useful in multiply nested macros that in simple ones.

Quality of Implementation and Attention to Detail

Finally, we have put a lot of work polishing the little things, because little things add up over time and contribute to a great user experience.

The following example shows that we recover from the simple case of forgetting a ; after a struct definition much better than GCC.

```
  $ cat t.cc
  template<class T>
  class a {};
  struct b {}
  a<int> c;
  $ gcc-4.9 t.cc
  t.cc:4:8: error: invalid declarator before 'c'
   a<int> c;
           ^
  $ clang t.cc
  t.cc:3:12: error: expected ';' after struct
  struct b {}
             ^
             ;
```

The following example shows that we diagnose and recover from a missing `typename` keyword well, even in complex circumstances where GCC cannot cope.

```
  $ cat t.cc
  template<class T> void f(T::type) { }
  struct A { };
  void g()
  {
      A a;
```

```
      f<A>(a);
  }
  $ gcc-4.9 t.cc
  t.cc:1:33: error: variable or field 'f' declared void
   template<class T> void f(T::type) { }
                                   ^
  t.cc: In function 'void g()':
  t.cc:6:5: error: 'f' was not declared in this scope
       f<A>(a);
       ^
  t.cc:6:8: error: expected primary-expression before '>' token
       f<A>(a);
          ^
  $ clang t.cc
  t.cc:1:26: error: missing 'typename' prior to dependent type name 'T::type'
  template<class T> void f(T::type) { }
                           ^~~~~~~
                           typename
  t.cc:6:5: error: no matching function for call to 'f'
      f<A>(a);
      ^~~~
  t.cc:1:24: note: candidate template ignored: substitution failure [with T =
A]: no type named 'type' in 'A'
  template<class T> void f(T::type) { }
                         ^    ~~~~
```

While each of these details is minor, we feel that they all add up to provide a much more polished experience.