Feature Design Specification

C/C++ Parser for CDT 2.0

Table of Contents

Document History

Revision	*Author*	*Date*	*Description*
1.0	John Camelon	2004-01-18	Initial Draft

Introduction

This document serves as the main repository for the architecture and design changes required to meet the requirements as they have been specified (refer to reference [1] for details).

Each section of this document describes a design proposal for either a set of requirements or a major outstanding defect. Due to the number of requirements, we have decided to limit this document to proposals we wish to recommend, rather than discussing ulterior options.

For parser related features, CDT 2.0's main focus is about improving the quality to the point where the core language related features work well out of the box.

Content Assist and Selection Search Support

This section addresses most of the Content Assist and Search requirements from [1]. The only requirement not mentioned in this section for Content Assist is #4, which concerns C++ templates. This is handled in a later section exclusively devoted to templates.

Interface Changes

ParserMode

Two new ParserMode's have been added to our public interface. Content Assist should use COMPLETION_PARSE and Search/Navigation should use SELECTION_PARSE.

IASTNode

IASTNode is a new interface that serves as the root of all AST nodes. It may be possible, given the right parse configuration, to use an IASTNode to query relevant symbols in the internal symbol table of the parser.

```
// To satisfy requirement Content Assist #3
// this interface represents the end-result of the query into the
// IParser via the AST which Content Assist can create completion
// proposals off of.
public static interface ILookupResult
{
	// IASTNodes that represent completion proposals
	public Iterator getNodes();

	// The prefix which all IASTNodes in the above iterator match
	public String getPrefix();

	// The size of the result set provided by the Iterator, to allow
	// for scaling from the client's side
	public int getResultsSize();
}

// this is the actual query off of IASTNode
public ILookupResult lookup(
	String prefix,            // the string prefix e.g."RT"
	LookupKind[] kind,  // the restrictions upon the query
	IASTNode context )  // a supplementary restriction on query
throws LookupError,       // error on inconsistent parameters
	throws ASTNotImplementedException; // COMPLETION_PARSE AST's only
```

The supplementary context provided in the last parameter of lookup() is necessary when the provided offset occurs after a '->', '.' or '::'. For example, if the offset provided is at the end of the code-snippet example **"int AClass::method(B * b) { b->"** , then the IASTNode where lookup() is called from is the method the node representing type 'B' with the supplementary context AClass::method(). This supplementary context is necessary in order to ensure that we provide accurate completion proposals according to

the rules of C/C++.

IASTCompletionNode

This construct represents all of the semantic information that is required to provide contextual content-completion at a given offset.

```
// what kind of completion is it?  CompletionKind is an enumeration
// representing all of the different cases ContentAssist defines
public CompletionKind     getCompletionKind();

// what scope is the cursor in?  A function body, a class
// specification, global, etc.
public IASTScope          getCompletionScope();

// within that scope, is there an additional context that serves as
// a more accurate context to work with?
public IASTNode           getCompletionContext();

// the final IToken prefix
public String             getCompletionPrefix();

// an iterator of strings of appropriate keywords that could
// be suggested at this particular point
public Iterator           getKeywords();
```

IParser

In COMPLETION_PARSE mode, the following interface is available on IParser:

```
// To satisfy requirement Content Assist #2
// given an offset in the file, provide the means in which one can
// expected client : Content Assist
IASTCompletionNode  parse( int offset ) // prefix ends @ offset
      throws ParseError; // significant error encountered
```

In SELECTION_PARSE mode, the following interface is available on IParser:

```
// To satisfy requirement Search Support #1
// given an offset duple, indicate to the user what AST construct it
// is referencing
// expected client : Search/Navigation
IASTNode  parse(    int startingOffset, // starts @ startingOffset
                    int endingOffset )  // ends @ endingOffset
      throws ParseError; // significant error encountered
```

IScanner

A new method has been added to IScanner to allow the parser to configure the scanner to stop scanning at a particular offset.

```
// throw OffsetLimitReachedException from IScanner::nextToken()
// once it encounters the provided offset
void setOffsetBoundary( int offset );

// the signature of nextToken : since OffsetLimitReachedException
```

```
// derives from EndOfFileException
public IToken nextToken() throws ScannerException, EndOfFileException;
```

Primary Interactions

Content Assist

1. Content Assist client constructs an IParser and IScanner instance through ParserFactory, and configures them appropriately to the buffer of the working copy.
2. ContentAssist calls IParser::parse(int) with the offset in the working copy that represents the point in where completion is to occur.
3. IParser calls setOffsetBoundary() on its IScanner so that the IScanner knows at what point to stop tokenizing the working copy buffer.
4. IParser then goes along parsing as it normally would in COMPLETE_PARSE mode.
5. IScanner reads up to the last offset and the throws an OffsetLimitReachedException, which references the last token created.
6. IParser handles this exception, and uses the last token contained in the exception to determine the completion prefix (or lack thereof). IParser then falls out of the grammar back down to IParser::parse(int).
7. Using its internal data structures, IParser creates an IASTCompletionNode that reflects the semantic value for that particular offset in the working copy.
8. Content Assist constructs an array of LookupKind's that best map to the CompletionKind of the IASTCompletionNode. Content Assist also determines the best IASTNode in the completion node to lookup symbols upon.
9. Content Assist calls IASTNode::lookup( ) with the appropriate values.
10. The IASTNode implementation delegates these calls to the parser symbol table and constructs the correct ILookupResult.
11. Content Assist uses the ILookupResult as contributions for completions.

Selection Search/Navigation

1. The navigation client constructs an IParser and IScanner instance through ParserFactory, and configures them appropriately to the buffer of the working copy.
2. The navigation client calls IParser::parse(int, int) with the appropriate offsets indicating what has been selected in the working copy.
3. IParser calls setOffsetBoundary() on its IScanner with the **second offset** provided so that the IScanner knows at what point to stop tokenizing the working copy buffer.

4. IParser then goes along parsing as it normally would in COMPLETE_PARSE mode.
5. IScanner reads up to the last offset and the throws an OffsetLimitReachedException, which references the last token created.
6. IParser handles this exception, and uses the last token contained in the exception to determine the last token consumed. IParser then falls out of the grammar back down to IParser::parse(int).
7. IParser then analyzes the last token consumed and uses this (along with other internal data structures) to lookup the name in the symbol table for that particular context context.
8. That value returned to the symbol table is then returned to the navigation client.
9. The navigation client then uses the standard search API to specifically search for the element, using the IASTNode as the determining factor.

Performance Enhancements

Both content assist and selection search have performance restrictions placed upon them, as for the most part, they are uncancellable gestures in the UI. While the onus is on the client to properly timeout the request, there are several performance improvements that we could do in the parser implementation and architecture to improve results. In the end, profiling will determine the what areas need attention, but here are a few ideas that have been mentioned:

- Add in support through ISourceElementRequestor for the client to signal to the parser that it has the information it requires and that we can stop parsing.
- Skipping function bodies in #include'd header files for COMPLETION_PARSE.
- Maintain an index or cache between parses that allows for optimizations in finding particular include files to avoid the header file linear-search for each item in the include path.
- Upping the buffer sizes for InputStreamBuffer.
- Other scanner optimizations that would minimize string manipulations and optimize file operations.

Template Support

This section describes the means that we plan to meet Content Assist requirement #4 from [1].

TBD – This section shall be completed in a later draft. It still requires significant refinement.

AST Interface Updates

This section describes the means that we plan to meet General Requirement #3 from [1].

Finish AST for ISO C++98/C99

Bugzilla Defects: 40768, 43241, 43242, 43579, 40422.

Aside from template support, one of the glaring holes in our AST representation regards pointers to functions, pointers to methods and pointers to members. While one can argue about the priority of supporting these constructs in features like content assist or search, if we do plan on publishing our interface for the upcoming release, it may prove difficult to restructure the AST post CDT 2.0.

Pointers to functions are difficult to represent in way other than what is in the C/C++ grammar, as they can net quite deep. An example from [3] reads as follows:

```
int (*fpif(int))(int);
/* fpif takes an integer argument and returns a pointer to a function
   that takes an integer argument and returns an integer */
```

The problem that we currently have is as follows: current IASTFunction contains the following to methods (amongst others):

```
public IASTAbstractDeclaration getReturnType();
public Iterator getParameters();
```

These constructs need to be restructured so they can scale in the same interface : i.e. A pointer to a function can be the entire signature for a declaration, just the return type of a function, just the type of a parameter, or any combination. Since there currently are clients using these interfaces, we shall attempt to maintain backwards compatibility through delegation of objects in the implementation.

Similarly, we currently do not have an AST representation for statements. This was just not completed last time due to lack of time and as none of the CDT 1.2 features required it.

Navigability amongst IASTNodes in COMPLETE_PARSE mode

Bugzilla defects: 50188 and 45372.

Currently, the method IASTScope::getDeclarations() throws ASTNotImplementedException when the parser runs in COMPLETE_PARSE mode. This is due to a deficiency in the parser symbol table and the design for solving this problem will most likely be detailed in this design document.

In other news, the current parser architecture made the assumption that clients interested in cross-reference information are not concerned with the details as to how a cross reference is used in the grammar. For example, the following code yields 3 different references to the same class, and it is unclear as to 'how' they came about:

```
/*
 * previously defined in a header file
 * class A {
 *   public:
 *      A( A * shallowCopy = 0 );
 * };
 *
 // reference #1: type of anA
 // reference #2: constructor call to A( A * )
 // reference #3: parameter to dynamic_cast
 A * anA = new A( dynamic_cast<A*>(this) );
```

Not only that, but the IASTVariable node that represents 'anA' in this example does not have a means of getting to the original IASTClassSpecifier 'A'. To solve this, we need to update our interface as follows:

```
/* add this method to IASTSimpleTypeSpecifier to enable variable
   declarations refer to their type */
IASTTypeSpecifier getTypeSpecifier()  // return cross-ref type spec if available
      throws ASTNotImplementedException; // not implemented for QUICK_PARSE
```

Add getSignature() to IASTExpression and type expressions

Bugzilla defect: 49990.

While the structure representation the AST interfaces allows for controlled access to how source code is represented, what is currently lost is the exact signature information as it is available in the code. For example, a constant integer variable could be declared as either 'const int' or 'int const'. Clients that wish to display the signature as it best appears in the code are not capable of doing so.

By adding a getSignature() method to these popular AST constructs (mainly types and expressions) we can satisfy these requirements. The parser can easily save 'begin' and 'end' tokens while it is working, which would allow for a preprocessed representation for what code produced a particular AST construct.

Significant CDTParser Defects/Features in Bugzilla

This section talks about the potentially significant design updates required to fix outstanding high-priority parser defects from CDT 1.2.

Inline method definition X-Reference support

Bugzilla defect: 44340

For methods defined inline, it is possible to refer to fields or methods that have not yet been reached in the file. Instead of reporting these unresolved references to the client through ISourceElementRequestor.acceptProblem(), we need to store enough meta-data about the lookup context and the element to be looked up to try and resolve the reference upon finishing the parse of the containing class specifier.

NOTE: We may have to accept slower content assist times on completion attempts within the body of an inline method.

C X-Reference support

Bugzilla defect: 49783

Since C does not require compile time name resolution, our current mechanism for determining cross references needs to be updated so that CDT index file maintains these link-time resolved cross-references.

The following method shall be added to ISourceElementRequestor and will only be called when parsing C files upon encountering an unresolved reference:

```
// does the client have any idea what this IdExpression is?
// return true ==> IParser will add a dummy symbol to the symbol table
// return false ==> IParser will generate and report an IProblem
boolean validateUnknownReference( String unresolvedIdExpression );
```

This should be enough infrastructure for the index to yield 'better' search results.

Scanner support for relative include paths

Bugzilla defect: 43051

Relative include paths specified upon the command line are common and popular within the C community. In GCC, relative include paths specified on the command line are relative to where GCC is being called from; however, relative paths within include directives are relative to the location of the current file. The Scanner therefore must:

- work with relative include paths and inclusions incrementally, as java.io.File does not handle paths contains “..” or “.” well
- have the information upon instantiation as to where the directory that the compiler is being called from

Scanner support for Readers of IResource

Bugzilla defect: 45140

Since the Scanner does not use the resource interface, it is possible for a particular resource's file to be open in the Scanner while the resource interface allows for clients to try and delete or modify the file. However, the parser source-code base needs to remain Eclipse and CDT agnostic in order to ensure that we can serve other clients beyond the CDT.

To do this, we shall add the following method to ISourceElementRequestor:

```
// client checks to see if the file specified is in the workspace
// if so create a reader off of the IResource, otherwise off the File
InputStreamReader provideReader( String filename );
```

We shall also provide a reference implementation outside of the parser source tree that clients can leverage.

Globalization/Unicode Support

Universal Character Name Support

This section describes the means that we plan to meet Globalization Support requirement #1 from [1].

Universal Character Names (UCNs) are specified in the ISO specifications [3,4] as escape sequences (\u or \U) followed by a series of hexadecimal digits ranging within a certain overall value. Ill-formed UCNs make the program ill formed, and thus will result in an additional IProblem being reported from the IScanner component.

Upon successfully scanning a UCN, an attempt will be made to translate this character into a Unicode character which serves almost like a macro-expansion. Offsets shall be preserved through the same manner that they are preserved for macro expansions.

Since UCNs are only allowed in particular places in the grammar (string literals, character literals and identifiers), this translated character will be added to the StringBuffer that is creating the upcoming IToken's image. (FYI: Regarding identifiers, Annex E of [3] places restrictions as to what UCNs are allowed within identifiers based upon locale.)

UTF-8 Encoding Support

This section describes the means that we plan to meet General Requirement #7 from [1].

Since the entire GNU tool chain supports input in the form of UTF-8 encoded files, it is necessary for the CDT's parsing framework to be able to handle files of this encoding. This support can be divided into two smaller requirements:

1. The parsing framework needs a way of determining the encoding of a source file. This responsibility shall be handled by the ParserFactory interface.
2. If the source file's encoding is supported (i.e. ASCII or UTF-8) then we need to be able to parse it appropriately so that content and offsets come out correctly. This responsibility shall be handled by our IScanner implementation (also known as Scanner).

ParserFactory

The signature of createScanner will unfortunately have to change, as it is not possible to know the encoding of the supplied Reader unless it is an InputStreamReader, which is a subclass of Reader. Since String's in Java are Unicode (UTF-16) then by converting the stream to a String we have already lost information. A migration plan shall be provided for clients that currently use StringReader as their input to ParserFactory.createScanner().

While we have be recommended against using InputStream as it does not support Unicode, it is clear that we need to work outside of the restrictions of Unicode when dealing with source files. (Note: gcc does not support unicode file input in its latest

version).

In order to create an InputStreamReader with the correct encoding, a client needs to figure out what the correct encoding should be according to the environment. For this, clients can call the following method:

```
// returns the appropriate character set identifier as specified by GCC & JRE
public static String getFileEncoding() throws ParserFactoryError;
```

To implement this method, ParserFactory uses the same logic that the GNU tool chain uses for determining the encoding of a file. It involves checking different environment variables (LANG, LC_CTYPE, LC_ALL) and parsing the values of these variables in order. The details of this logic is provided in [6]. The String that is returned is of the format specified in java.nio.charset.Charset [5]. ParserFactory.getFileEncoding() will only return the values "US-ASCII" or "UTF-8"; any other value determined through the GNU environment will yield a ParserFactoryError, as it is unsupported in this release.

The consequences of this strategy is that CDT users cannot specify only a subset of their resources to be UTF-8 and the rest US-ASCII. According to [6], that is an unsupported configuration.

With the result of ParserFactory.getFileEncoding(), an InputStreamReader can be constructed using the following constructor:

```
InputStreamReader( InputStream in, String charsetName );
```

Scanner

The IScanner implementation shall be refactored to use exclusively InputStreamReaders and to set the encoding appropriately. Special effort may be necessary in determining the offsets of files differing from the default encoding of Eclipse.

GCC Support

This section describes the means that we plan to meet General Requirement #1 from [1].

Even though we do not have support for templates entirely in the COMPLETE_PARSE mode for our parser, we have found that our lack of support for the extensions provided in our de-facto C++ compiler (g++). If we support as many gcc 3.3.2 extensions as possible, we will in the end, address the overall quality of our language-related features in CDT.

For the most part, adding support for the GNU compiler does not require any additional design, just further implementation and a whole lot of testing. One design consideration is that it would be nice to keep the Scanner, Parser and Symbol Table intact for pure ANSI C and C++ so that should we need to support another compiler which is unlike GNU (like VC++ for example), we would have a good starting point.

Once we decide upon the priority for each of these individual extensions, they shall be listed explicitly in [1].

GCC Extensions for C

The following extensions are detailed in reference [2]. I have left out the extensions that are already correctly handled by the current parser in CVS HEAD.

Statement Exprs: Putting statements and declarations inside expressions.

Description: gcc accepts statements (and declarations) like

```
int y( { int y = foo (); int z;
         if (y > 0) z = y;
         else z = - y;
         z; })
```

Proposed Solution : Restructure the parser to handle this type of syntax and refactor the AST for Expression to allow for these type of relationships.

Difficulty/Risk: High, since it means that we would have to restructure the C++ grammar somewhat in order to handle these new constructs.

Local Labels: Labels local to a statement-expression.

Description: gcc accepts the ability to create local labels inside function blocks using the following syntax:

```
while( true )
{
   __label__ topOfLoop;
   /* ... */
   goto topOfLoop; // goes to top of while
```

```
for( int i = 0; i < 1000; ++i )
{
   __label__ topOfLoop;
   /* ... */
   goto topOfLoop; // goes to top of for
   /* ...
```

Proposed Solution: We do not have to do anything more than accept this new syntax, perhaps more if a client requires it.

Difficulty/Risk: Low, since this is a simple change to the grammar and does not affect any of the ambiguities or lookahead we use to help us efficiently parse.

Labels as Values: Getting pointers to labels, and computed gotos.

Description: Labels can be passed around as void pointers, and expressions can be put into gotos. For example:

```
static const int array[] = { &&foo - &&foo, &&bar - &&foo, &&hack - &&foo };
goto *(&&foo + array[i]);
```

Proposed Solution: We would need to update the grammar and AST to handle the new type of postfix expression (&& postfixExpression). We would also need to update the grammar for 'goto'. Since labels are not stored in the symbol table, we will need to make sure that a lookup of a label in the symbol table does not fail the parse for these types of expressions.

Difficulty/Risk: Medium, since this type of change does affect many components.

Nested Functions: As in Algol and Pascal, lexical scoping of functions.

Description: Functions can be defined within functions.

```
void hack (int *array, int size)
{
   void store (int index, int value)
   { array[index] = value; }

   store( 3, 77 );
}
```

Proposed Solution: For the most part I believe we already handle this. I am not sure if cross-references (i.e. Symbol table) works completely correctly for these test cases though.

Difficulty/Risk: Unknown.

Builtin Functions for all GCC targets

Description: GCC provides some built in functions to allow for wormholes in the users code. (Refer to [2] for details).

Proposed Solution: Watch for these 22 identifiers in expressions and handle them appropriately in the grammar.

Difficulty/Risk: Low.

Typeof: typeof: referring to the type of an expression.

Description: GCC provides the way of determining the runtime type of an expression or declaration in order to allow for other variables to be declared similarily.

```
#define pointer(T)  typeof(T *) // pointer to type of T
#define array(T, N) typeof(T [N]) // array of size N of T's typ
```

Proposed Solution: Alter the grammar to accept this syntactically as another type of unary expression. Update the AST accordingly. This may cause some inaccurate search results, since we cannot know at compile time what the real type of a parameter to a function may be.

Difficulty/Risk: Medium.

Conditionals: Omitting the middle operand of a ?: expression.

Description: In GCC,

```
x ? : y == x ? x : y
```

Proposed Solution: Change the grammar to accept it and add another Expression category to IASTExpression.

Difficulty/Risk: Low.

Long Long: Double-word integers---long long int.

Description: ANSI C++ doesn't support 64 bit integers. GCC does.

```
long long int anInt = 46LL;
```

Proposed Solution: Update the scanner to accept LL as a suffix to an integer literal. Update the parser to accept “long long int” and the other combinations specified in the GCC manual as being valid type descriptors. Update the AST to reflect this capability so that it can be properly displayed in the Outline view.

Difficulty/Risk: Low.

Complex: Data types for complex numbers.

Description: Complex numbers are part of ANSI C but not ANSI C++. GNU allows them in ANSI C++.

Proposed Solution: Allow GCC variant to also accept _Complex and _Imaginary numbers, along with the specified aliases for those keywords.

Difficulty/Risk: Low

Hex Floats: Hexadecimal floating-point constants.

Description: GNU accepts constants initialized in the following format:

```
0x1.fp3
```

Proposed Solution: Update the scanner to accept this format.

Difficulty/Risk: Low

Variadic Macros: Macros with a variable number of arguments.

Description: C99 and GCC have different syntax when it comes to declaring macros with a variable number of arguments. We currently handle neither.

```
#define debug(format, ...) fprintf (stderr,format,__VA_ARGS__) //C99
#define debug(format, args...) fprintf (stderr, format, args) //GCC
```

Proposed Solution: Update the scanner to allow for these different macro specification and their usage.

Difficulty/Risk: Medium

Escaped Newlines: Slightly looser rules for escaped newlines.

Description: From the GCC manual:

> Recently, the preprocessor has relaxed its treatment of escaped newlines. Previously, the newline had to immediately follow a backslash. The current implementation allows whitespace in the form of spaces, horizontal and vertical tabs, and form feeds between the backslash and the subsequent newline. The preprocessor issues a warning, but treats it as a valid escaped newline and combines the two lines to form a single logical line. This works within comments and tokens, as well as between tokens. Comments are not treated as whitespace for the purposes of this relaxation, since they have not yet been replaced with spaces.

Proposed Solution: This is something that is handled entirely in the Scanner.

Difficulty/Risk: Medium : While the work is not that difficult, there are many test

cases that would need to be written for this.

Pointer Arith: Arithmetic on void-pointers and function pointers.

Description: The manual says:

> In GNU C, addition and subtraction operations are supported on pointers to void and on pointers to functions. This is done by treating the size of a void or of a function as 1. A consequence of this is that sizeof is also allowed on void and on function types, and returns 1. The option -Wpointer-arith requests a warning if these extensions are used.

Proposed Solution: Update the complete parse AST factory to allow for these additions.

Difficulty/Risk: Medium

Compound Literals

Description: GCC allows for the compound literal notation of C99 to be used in C++ mode, even though it is not part of the ANSI specification.

Proposed Solution: We may not have to do anything to support this; if we do, it will just be undoing a small amount of work that I did to ensure that we did not allow for that in C++ mode.

Difficulty/Risk: Low

Cast to Union: Casting to union type from any member of for the union.

Description: Examine the following code:

```
union foo u;
/* ... */
u = (union foo) x; // ==>  u.i = x
u = (union foo) y; // ==>  u.d = y
```

Proposed Solution: We would be required to update CompleteParseASTFactory in order to make sure that these type of casts as parameter arguments come out appropriately. (Note that the union u in the previous example is not a pointer, and thus does not yield an lvalue like normal casts).

Difficulty/Risk: Medium

Case Ranges: `case 1 ... 9' and such.

Description: One can specify a range in a case statement rather than a single value.

Proposed Solution: We would need to update the grammar and AST in order to accommodate this extension.

Difficulty/Risk: Low/Medium

Attributes:

Description: GCC provides formal syntax for adding "__attributes__" to these declarations:

- functions
- variables
- types

Proposed Solution: This would require significant work in the grammar as well as some additions to the AST and AST Factories for each mode. These features are also commonly used.

Difficulty/Risk: High

Function Prototypes: Prototype declarations and old-style definitions.

Description: GCC accepts K&R type function definitions and also allows for them to be used with standard C-style function prototypes.

```
/* Use prototypes unless the compiler is old-fashioned.  */
#ifdef __STDC__
#define P(x) x
#else
#define P(x) ()
#endif

/* Prototype function declaration.  */
int isroot P((uid_t));

/* Old-style function definition.  */
int
isroot (x)   /* ??? lossage here ??? */
     uid_t x;
{
  return x == 0;
}
```

Proposed Solution: Update the parser grammar to accept K&R C.

Difficulty/Risk: Medium. We've already tried this a couple of times last release, and it was somewhat cumbersome.

Dollar Signs: Dollar sign is allowed in identifiers.

Description: In GCC, one can name variables things like 'money$nothing'

Proposed Solution: Update the scanner to deal with it.

Difficulty/Risk: Low.

Character Escapes: \e stands for the character <ESC>.

Description: This new escape character can be used in string and character literals.

Proposed Solution: Update the scanner accordingly.

Difficulty/Risk: Low.

Alignment: Inquiring about the alignment of a type or variable.

Description: A new unary expression is provided along the lines of:

```
__alignof__( typeid ) || __alignof( expression )
```

Proposed Solution: Update the Scanner to allow for __alignof__ to be a token. Update the parser and AST to properly recognize the syntax and capture it in the AST. Update CompleteParseASTFactory to successfully navigate the expressions so that the appropriate cross references come out.

Difficulty/Risk: Medium

Extended Asm: Assembler instructions with C expressions as operands.

Description: C variables and expressions can be referenced from within asm() statements, like in the following example:

```
#define sin(x)       \
     ({ double __value, __arg = (x);   \
        asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
        __value; })
```

Proposed Solution: Update IASTASMDeclaration and the parser to handle this new syntax. Update CompleteParseASTFactory to properly handle the cross reference usage inside this new syntax.

Difficulty/Risk: High. The syntax is not simple. There are numerous constraints fo the operands that may need to be accounted for, say, if we wished to search only read accesses vs. write accesses.

Asm Labels: Specifying the assembler name to use for a C symbol.

Description: You can specify the way that you want the function name mangled in GCC through different syntax. For example,

```
// ordinarily the name in assembler code for 'foo' would be _foo
int foo asm ("myfoo") = 2; // foo's name in assembly code is myfoo
```

Proposed Solution: Update grammar accordingly.

Difficulty/Risk: Low

Explicit Reg Vars: Defining variables residing in specified registers.

Description: Similarly to the previous requirement, GCC allows for register varisables to be placed in particular registers.

```
register int x asm( “r1” ); // variable x goes in register r1
```

Proposed Solution: Update grammar accordingly.

Difficulty/Risk: Low

Alternate Keywords: __inline__, __asm__, etc., for header files.

Description: Quite a few aliases are available for particular keywords in GCC, to support interoperability between different dialects that the compiler supports.

Proposed Solution: Add the appropriate tokens for these new keywords.

Difficulty/Risk: Low

Function Names: Printable strings which are the name of the current function.

Description: A couple of additional macros are defined that allow for the user to obtain a string to the name (__FUNCTION__) or fully qualified name (__PRETTY_FUNCTION__).

Proposed Solution: Add these symbols to the macro table at startup.

Difficulty/Risk: Low

Target Builtins: Built-in functions specific to particular targets.

Description: There are approximately an additional 300 built-in functions that are specific to a particular target. These targets include x86, Alpha & PowerPC.

Proposed Solution: Check for these functions in expressions.

Difficulty/Risk: Medium – Need to investigate overlap and consequences of mixing these targets, if any.

Unnamed Fields: Unnamed struct/union fields within structs/unions.

Description: The following code example explains it all.

```
struct {
  int a;
  union {
     int b;
     float c;
  };
  int d;
} foo;
foo.b = 5; // accesses the anonymous union's member
```

Proposed Solution: The work that needs to be done to support this falls in the realm of the symbol table. The parser itself already supports these types of constructs.

Difficulty/Risk: Medium.

Thread-Local: Per-thread variables.

Description: Introduce a new keyword __thread to indicate that a variable should use thread local storage. There are restrictions on where it can be used and what other keywords can be mixed with it.

Proposed Solution: Add the new token, modify the grammar and update the AST to expose this information.

Difficulty/Risk: Medium

GCC Extensions for C++

Minimum and Maximum Operators in C++

Description: Operator >? and <? have been added to serve as minimum and maximum.

Proposed Solution: Update the scanner to allow for these two new operators. Add two new tokens to IToken, update the grammar to allow for these types of relational expressions and update the CompleteParseASTFactory to ensure that the operators return values are calculated properly.

Difficulty/Risk: Medium

Restricting Pointer Aliasing

Description: Keywords __restrict__ and __restrict are allowed in C++, similar to 'restrict' which is allowed in C++.

Proposed Solution: Update the token list appropriately.

Difficulty/Risk: Low

Extracting the function pointer from a bound pointer to member function

Description: Function pointers can be extracted from pointers to members. See the following example:

```
//ANSI C++
extern A a;
extern int (A::*fp)();
typedef int (*fptr)(A *);
fptr p = (fptr)(a.*fp);
```

/* For PMF constants (i.e. expressions of the form &Klasse::Member), no object is needed to obtain the address of the function. They can be converted to function pointers directly: */

```
//GNU C++
fptr p1 = (fptr)(&A::foo);
```

Proposed Solution: Since we do not currently have a satisfactory way of handling pointers to members in our AST (refer to the previous section named *AST Interface Updates*), there has been no work in the symbol table or CompleteParseASTFactory to support cross-reference information regarding this.

Difficulty/Risk: High

Undocumented GNU extensions

There are some defects in bugzilla (and assorted hearsay and rumours from verification department) which refer to different C and C++ extensions for GNU that I have not been able to verify in the manual as of yet. Many of these were uncovered by running our TortureTest Junit test upon a GCC test suite.

These undocumented extensions include:

- full-qualification of method names in declaration
- #include next
- forward declarations in parameter lists
- named return values

- __declspec support
- naming an expression's type
- extended syntax for template instantiation
- 'signature' keyword and constructs
- __extension__ keyword
- #ident directive

References

1. Software Requirements Specification – C/C++ Parser for CDT 2.0.
2. http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/C-Extensions.html#C Extensions
3. C++ Standard – ISOIEC14882-1998
4. C Standard – ISOIEC9899-1999
5. J2SE 1.4.2 API – java.nio.charset.Charset
6. gcc, locale and UTF-8 manual pages on SuSe 9.0 Linux.
7. The Java Tutorial - Character and Byte Streams
8. Globalization FDS in CDT 2.0