LLVM Programmer's Manual

> Warning
>
> This is always a work in progress.

Introduction

This document is meant to highlight some of the important classes and interfaces available in the LLVM source-base. This manual is not intended to explain what LLVM is, how it works, and what LLVM code looks like. It assumes that you know the basics of LLVM and are interested in writing transformations or otherwise analyzing or manipulating the code.

This document should get you oriented so that you can find your way in the continuously growing source code that makes up the LLVM infrastructure. Note that this manual is not intended to serve as a replacement for reading the source code, so if you think there should be a method in one of these classes to do something, but it's not listed, check the source. Links to the doxygen sources are provided to make this as easy as possible.

The first section of this document describes general information that is useful to know when working in the LLVM infrastructure, and the second describes the Core LLVM classes. In the future this manual will be extended with information describing how to use extension libraries, such as dominator information, CFG traversal routines, and useful utilities like the `InstVisitor` (doxygen) template.

General Information

This section contains general information that is useful if you are working in the LLVM source-base, but that isn't specific to any particular API.

The C++ Standard Template Library

LLVM makes heavy use of the C++ Standard Template Library (STL), perhaps much more than you are used to, or have seen before. Because of this, you might want to do a little background reading in the techniques used and capabilities of the library. There are many good pages that discuss the STL, and several books on the subject that you can get, so it will not be discussed in this document.

Here are some useful links:

1. cppreference.com - an excellent reference for the STL and other parts of the standard C++ library.
2. C++ In a Nutshell - This is an O'Reilly book in the making. It has a decent Standard Library Reference that rivals Dinkumware's, and is unfortunately no longer free since the book has been published.
3. C++ Frequently Asked Questions.
4. SGI's STL Programmer's Guide - Contains a useful Introduction to the STL.
5. Bjarne Stroustrup's C++ Page.
6. Bruce Eckel's Thinking in C++, 2nd ed. Volume 2 Revision 4.0 (even better, get the book).

You are also encouraged to take a look at the LLVM Coding Standards guide which focuses on how to write maintainable code more than where to put your curly braces.

Other useful references

1. Using static and shared libraries across platforms

Important and useful LLVM APIs

Here we highlight some LLVM APIs that are generally useful and good to know about when writing transformations.

The `isa<>`, `cast<>` and `dyn_cast<>` templates

The LLVM source-base makes extensive use of a custom form of RTTI. These templates have many similarities to the C++ `dynamic_cast<>` operator, but they don't have some drawbacks (primarily stemming from the fact that `dynamic_cast<>` only works on classes that have a v-table). Because they are used so often, you must know what they do and how they work. All of these templates are defined in the `llvm/Support/Casting.h` (doxygen) file (note that you very rarely have to include this file directly).

`isa<>`:

> The `isa<>` operator works exactly like the Java "`instanceof`" operator. It returns true or false depending on whether a reference or pointer points to an instance of the specified class. This can be very useful for constraint checking of various sorts (example below).

`cast<>`:

> The `cast<>` operator is a "checked cast" operation. It converts a pointer or reference from a base class to a derived class, causing an assertion failure if it is not really an instance of the right type. This should be used in cases where you have some information that makes you believe that something is of the right type. An example of the `isa<>` and `cast<>` template is:

```
static bool isLoopInvariant(const Value *V, const Loop *L) {
  if (isa<Constant>(V) || isa<Argument>(V) || isa<GlobalValue>(V))
    return true;
```

```
  // Otherwise, it must be an instruction...
  return !L->contains(cast<Instruction>(V)->getParent());
}
```

Note that you should not use an `isa<>` test followed by a `cast<>`, for that use the `dyn_cast<>` operator.

`dyn_cast<>`:

The `dyn_cast<>` operator is a "checking cast" operation. It checks to see if the operand is of the specified type, and if so, returns a pointer to it (this operator does not work with references). If the operand is not of the correct type, a null pointer is returned. Thus, this works very much like the `dynamic_cast<>` operator in C++, and should be used in the same circumstances. Typically, the `dyn_cast<>` operator is used in an `if` statement or some other flow control statement like this:

```
if (auto *AI = dyn_cast<AllocationInst>(Val)) {
  // ...
}
```

This form of the `if` statement effectively combines together a call to `isa<>` and a call to `cast<>` into one statement, which is very convenient.

Note that the `dyn_cast<>` operator, like C++'s `dynamic_cast<>` or Java's `instanceof` operator, can be abused. In particular, you should not use big chained `if/then/else` blocks to check for lots of different variants of classes. If you find yourself wanting to do this, it is much cleaner and more efficient to use the `InstVisitor` class to dispatch over the instruction type directly.

`isa_and_nonnull<>`:

The `isa_and_nonnull<>` operator works just like the `isa<>` operator, except that it allows for a null pointer as an argument (which it then returns false). This can sometimes be useful, allowing you to combine several null checks into one.

`cast_or_null<>`:

The `cast_or_null<>` operator works just like the `cast<>` operator, except that it allows for a null pointer as an argument (which it then propagates). This can sometimes be useful, allowing you to combine several null checks into one.

`dyn_cast_or_null<>`:

The `dyn_cast_or_null<>` operator works just like the `dyn_cast<>` operator, except that it allows for a null pointer as an argument (which it then propagates). This can sometimes be useful, allowing you to combine several null checks into one.

These five templates can be used with any classes, whether they have a v-table or not. If you want to add support for these templates, see the document How to set up LLVM-style RTTI for your class hierarchy

Passing strings (the `StringRef` and `Twine` classes)

Although LLVM generally does not do much string manipulation, we do have several important APIs which take strings. Two important examples are the Value class - which has names for instructions, functions, etc. - and the `StringMap` class which is used extensively in LLVM and Clang.

These are generic classes, and they need to be able to accept strings which may have embedded null characters. Therefore, they cannot simply take a `const char *`, and taking a `const std::string&` requires clients to perform a heap allocation which is usually unnecessary. Instead, many LLVM APIs use a `StringRef` or a `const Twine&` for passing strings efficiently.

The `StringRef` class

The `StringRef` data type represents a reference to a constant string (a character array and a length) and supports the common operations available on `std::string`, but does not require heap allocation.

It can be implicitly constructed using a C style null-terminated string, an `std::string`, or explicitly with a character pointer and length. For example, the `StringRef` find function is declared as:

```
iterator find(StringRef Key);
```

and clients can call it using any one of:

```
Map.find("foo");                 // Lookup "foo"
Map.find(std::string("bar"));    // Lookup "bar"
Map.find(StringRef("\0baz", 4)); // Lookup "\0baz"
```

Similarly, APIs which need to return a string may return a `StringRef` instance, which can be used directly or converted to an `std::string` using the `str` member function. See `llvm/ADT/StringRef.h` (doxygen) for more information.

You should rarely use the `StringRef` class directly, because it contains pointers to external memory it is not generally safe to store an instance of the class (unless you know that the external storage will not be freed). `StringRef` is small and pervasive enough in LLVM that it should always be passed by value.

The `Twine` class

The `Twine` (doxygen) class is an efficient way for APIs to accept concatenated strings. For example, a common LLVM paradigm is to name one instruction based on the name of another instruction with a suffix, for example:

```
New = CmpInst::Create(..., SO->getName() + ".cmp");
```

The `Twine` class is effectively a lightweight rope which points to temporary (stack allocated) objects. Twines can be implicitly constructed as the result of the plus operator applied to strings (i.e., a C strings, an `std::string`, or a `StringRef`). The twine delays the actual concatenation of strings until it is actually required, at which point it can be efficiently rendered directly into a character array. This avoids unnecessary heap allocation involved in constructing the temporary results of string concatenation. See `llvm/ADT/Twine.h` (doxygen) and here for more information.

As with a `StringRef`, `Twine` objects point to external memory and should almost never be stored or mentioned directly. They are intended solely for use when defining a function which should be able to efficiently accept concatenated strings.

Formatting strings (the `formatv` function)

While LLVM doesn't necessarily do a lot of string manipulation and parsing, it does do a lot of string formatting. From diagnostic messages, to llvm tool outputs such as `llvm-readobj` to printing verbose disassembly listings and LLDB runtime logging, the need for string formatting is pervasive.

The `formatv` is similar in spirit to `printf`, but uses a different syntax which borrows heavily from Python and C#. Unlike `printf` it deduces the type to be formatted at compile time, so it does not need a format specifier such as `%d`. This reduces the mental overhead of trying to construct portable format strings, especially for platform-specific types like `size_t` or pointer types. Unlike both `printf` and Python, it additionally fails to compile if LLVM does not know how to format the type. These two properties ensure that the function is both safer and simpler to use than traditional formatting methods such as the `printf` family of functions.

Simple formatting

A call to `formatv` involves a single format string consisting of 0 or more replacement sequences, followed by a variable length list of replacement values. A replacement sequence is a string of the form `{N[[,align]:style]}`.

`N` refers to the 0-based index of the argument from the list of replacement values. Note that this means it is possible to reference the same parameter multiple times, possibly with different style and/or alignment options, in any order.

`align` is an optional string specifying the width of the field to format the value into, and the alignment of the value within the field. It is specified as an optional alignment style followed by a positive integral field width. The alignment style can be one of the characters `-` (left align), `=` (center align), or `+` (right align). The default is right aligned.

`style` is an optional string consisting of a type specific that controls the formatting of the value. For example, to format a floating point value as a percentage, you can use the style option `P`.

Custom formatting

There are two ways to customize the formatting behavior for a type.

1. Provide a template specialization of `llvm::format_provider<T>` for your type `T` with the appropriate static format method.

```
namespace llvm {
  template<>
  struct format_provider<MyFooBar> {
    static void format(const MyFooBar &V, raw_ostream &Stream, StringRef Style) {
      // Do whatever is necessary to format `V` into `Stream`
    }
  };
  void foo() {
    MyFooBar X;
    std::string S = formatv("{0}", X);
  }
}
```

   This is a useful extensibility mechanism for adding support for formatting your own custom types with your own custom Style options. But it does not help when you want to extend the mechanism for formatting a type that the library already knows how to format. For that, we need something else.

2. Provide a format adapter inheriting from `llvm::FormatAdapter<T>`.

```
namespace anything {
  struct format_int_custom : public llvm::FormatAdapter<int> {
    explicit format_int_custom(int N) : llvm::FormatAdapter<int>(N) {}
    void format(llvm::raw_ostream &Stream, StringRef Style) override {
      // Do whatever is necessary to format ``this->Item`` into ``Stream``
    }
  };
}
namespace llvm {
  void foo() {
    std::string S = formatv("{0}", anything::format_int_custom(42));
  }
}
```

   If the type is detected to be derived from `FormatAdapter<T>`, `formatv` will call the `format` method on the argument passing in the specified style. This allows one to provide custom formatting of any type, including one which already has a builtin format provider.

formatv Examples

Below is intended to provide an incomplete set of examples demonstrating the usage of `formatv`. More information can be found by reading the doxygen documentation or by looking at the unit test suite.

```
std::string S;
// Simple formatting of basic types and implicit string conversion.
S = formatv("{0} ({1:P})", 7, 0.35);  // S == "7 (35.00%)"
```

```
// Out-of-order referencing and multi-referencing
outs() << formatv("{0} {2} {1} {0}", 1, "test", 3); // prints "1 3 test 1"

// Left, right, and center alignment
S = formatv("{0,7}",  'a');  // S == "      a";
S = formatv("{0,-7}", 'a');  // S == "a      ";
S = formatv("{0,=7}", 'a');  // S == "   a   ";
S = formatv("{0,+7}", 'a');  // S == "      a";

// Custom styles
S = formatv("{0:N} - {0:x} - {1:E}", 12345, 123908342); // S == "12,345 - 0x3039 - 1.24E8"

// Adapters
S = formatv("{0}", fmt_align(42, AlignStyle::Center, 7));  // S == "  42   "
S = formatv("{0}", fmt_repeat("hi", 3)); // S == "hihihi"
S = formatv("{0}", fmt_pad("hi", 2, 6)); // S == "  hi      "

// Ranges
std::vector<int> V = {8, 9, 10};
S = formatv("{0}", make_range(V.begin(), V.end())); // S == "8, 9, 10"
S = formatv("{0:$[+]}", make_range(V.begin(), V.end())); // S == "8+9+10"
S = formatv("{0:$[ + ]@[x]}", make_range(V.begin(), V.end())); // S == "0x8 + 0x9 + 0xA"
```

Error handling

Proper error handling helps us identify bugs in our code, and helps end-users understand errors in their tool usage. Errors fall into two broad categories: programmatic and recoverable, with different strategies for handling and reporting.

Programmatic Errors

Programmatic errors are violations of program invariants or API contracts, and represent bugs within the program itself. Our aim is to document invariants, and to abort quickly at the point of failure (providing some basic diagnostic) when invariants are broken at runtime.

The fundamental tools for handling programmatic errors are assertions and the llvm_unreachable function. Assertions are used to express invariant conditions, and should include a message describing the invariant:

```
assert(isPhysReg(R) && "All virt regs should have been allocated already.");
```

The llvm_unreachable function can be used to document areas of control flow that should never be entered if the program invariants hold:

```
enum { Foo, Bar, Baz } X = foo();

switch (X) {
  case Foo: /* Handle Foo */; break;
  case Bar: /* Handle Bar */; break;
  default:
    llvm_unreachable("X should be Foo or Bar here");
}
```

Recoverable Errors

Recoverable errors represent an error in the program's environment, for example a resource failure (a missing file, a dropped network connection, etc.), or malformed input. These errors should be detected and communicated to a level of the program where they can be handled appropriately. Handling the error may be as simple as reporting the issue to the user, or it may involve attempts at recovery.

Note

> While it would be ideal to use this error handling scheme throughout LLVM, there are places where this hasn't been practical to apply. In situations where you absolutely must emit a non-programmatic error and the `Error` model isn't workable you can call `report_fatal_error`, which will call installed error handlers, print a message, and abort the program. The use of *report_fatal_error* in this case is discouraged.

Recoverable errors are modeled using LLVM's `Error` scheme. This scheme represents errors using function return values, similar to classic C integer error codes, or C++'s `std::error_code`. However, the `Error` class is actually a lightweight wrapper for user-defined error types, allowing arbitrary information to be attached to describe the error. This is similar to the way C++ exceptions allow throwing of user-defined types.

Success values are created by calling `Error::success()`, E.g.:

```
Error foo() {
  // Do something.
  // Return success.
  return Error::success();
}
```

Success values are very cheap to construct and return - they have minimal impact on program performance.

Failure values are constructed using `make_error<T>`, where `T` is any class that inherits from the ErrorInfo utility, E.g.:

```
class BadFileFormat : public ErrorInfo<BadFileFormat> {
public:
  static char ID;
  std::string Path;

  BadFileFormat(StringRef Path) : Path(Path.str()) {}

  void log(raw_ostream &OS) const override {
    OS << Path << " is malformed";
  }

  std::error_code convertToErrorCode() const override {
    return make_error_code(object_error::parse_failed);
  }
};

char BadFileFormat::ID; // This should be declared in the C++ file.

Error printFormattedFile(StringRef Path) {
  if (<check for valid format>)
    return make_error<BadFileFormat>(Path);
  // print file contents.
  return Error::success();
}
```

Error values can be implicitly converted to bool: true for error, false for success, enabling the following idiom:

```
Error mayFail();

Error foo() {
  if (auto Err = mayFail())
    return Err;
  // Success! We can proceed.
  ...
```

For functions that can fail but need to return a value the `Expected<T>` utility can be used. Values of this type can be constructed with either a `T`, or an `Error`. Expected<T> values are also implicitly convertible to boolean, but with the opposite convention to `Error`: true for success, false for error. If success, the `T` value can be accessed via the dereference operator. If failure, the `Error` value can be extracted using the `takeError()` method. Idiomatic usage looks like:

```
Expected<FormattedFile> openFormattedFile(StringRef Path) {
  // If badly formatted, return an error.
  if (auto Err = checkFormat(Path))
    return std::move(Err);
  // Otherwise return a FormattedFile instance.
  return FormattedFile(Path);
}

Error processFormattedFile(StringRef Path) {
  // Try to open a formatted file
  if (auto FileOrErr = openFormattedFile(Path)) {
    // On success, grab a reference to the file and continue.
    auto &File = *FileOrErr;
    ...
  } else
    // On error, extract the Error value and return it.
    return FileOrErr.takeError();
}
```

If an `Expected<T>` value is in success mode then the `takeError()` method will return a success value. Using this fact, the above function can be rewritten as:

```
Error processFormattedFile(StringRef Path) {
  // Try to open a formatted file
  auto FileOrErr = openFormattedFile(Path);
  if (auto Err = FileOrErr.takeError())
    // On error, extract the Error value and return it.
    return Err;
  // On success, grab a reference to the file and continue.
  auto &File = *FileOrErr;
  ...
}
```

This second form is often more readable for functions that involve multiple `Expected<T>` values as it limits the indentation required.

If an `Expected<T>` value will be moved into an existing variable then the `moveInto()` method avoids the need to name an extra variable. This is useful to enable `operator->()` the `Expected<T>` value has pointer-like semantics. For example:

```
Expected<std::unique_ptr<MemoryBuffer>> openBuffer(StringRef Path);
Error processBuffer(StringRef Buffer);

Error processBufferAtPath(StringRef Path) {
  // Try to open a buffer.
  std::unique_ptr<MemoryBuffer> MB;
  if (auto Err = openBuffer(Path).moveInto(MB))
    // On error, return the Error value.
    return Err;
  // On success, use MB.
  return processContent(MB->getBuffer());
}
```

This third form works with any type that can be assigned to from `T&&`. This can be useful if the `Expected<T>` value needs to be stored an already-declared `Optional<T>`. For example:

```
Expected<StringRef> extractClassName(StringRef Definition);
struct ClassData {
  StringRef Definition;
  Optional<StringRef> LazyName;
  ...
  Error initialize() {
    if (auto Err = extractClassName(Path).moveInto(LazyName))
      // On error, return the Error value.
      return Err;
    // On success, LazyName has been initialized.
    ...
  }
};
```

All `Error` instances, whether success or failure, must be either checked or moved from (via `std::move` or a return) before they are destructed. Accidentally discarding an unchecked error will cause a program abort at the point where the unchecked value's destructor is run, making it easy to identify and fix violations of this rule.

Success values are considered checked once they have been tested (by invoking the boolean conversion operator):

```
if (auto Err = mayFail(...))
  return Err; // Failure value - move error to caller.

// Safe to continue: Err was checked.
```

In contrast, the following code will always cause an abort, even if `mayFail` returns a success value:

```
mayFail();
// Program will always abort here, even if mayFail() returns Success, since
// the value is not checked.
```

Failure values are considered checked once a handler for the error type has been activated:

```
handleErrors(
  processFormattedFile(...),
  [](const BadFileFormat &BFF) {
    report("Unable to process " + BFF.Path + ": bad format");
  },
  [](const FileNotFound &FNF) {
    report("File not found " + FNF.Path);
  });
```

The `handleErrors` function takes an error as its first argument, followed by a variadic list of "handlers", each of which must be a callable type (a function, lambda, or class with a call operator) with one argument. The `handleErrors` function will visit each handler in the sequence and check its argument type against the dynamic type of the error, running the first handler that matches. This is the same decision process that is used decide which catch clause to run for a C++ exception.

Since the list of handlers passed to `handleErrors` may not cover every error type that can occur, the `handleErrors` function also returns an Error value that must be checked or propagated. If the error value that is passed to `handleErrors` does not match any of the handlers it will be returned from handleErrors. Idiomatic use of `handleErrors` thus looks like:

```
if (auto Err =
      handleErrors(
        processFormattedFile(...),
        [](const BadFileFormat &BFF) {
          report("Unable to process " + BFF.Path + ": bad format");
        },
        [](const FileNotFound &FNF) {
          report("File not found " + FNF.Path);
        }))
  return Err;
```

In cases where you truly know that the handler list is exhaustive the `handleAllErrors` function can be used instead. This is identical to `handleErrors` except that it will terminate the program if an unhandled error is passed in, and can therefore return void. The `handleAllErrors` function should generally be avoided: the introduction of a new error type elsewhere in the program can easily turn a formerly exhaustive list of errors into a non-exhaustive list, risking unexpected program termination. Where possible, use handleErrors and propagate unknown errors up the stack instead.

For tool code, where errors can be handled by printing an error message then exiting with an error code, the ExitOnError utility may be a better choice than handleErrors, as it simplifies control flow when calling fallible functions.

In situations where it is known that a particular call to a fallible function will always succeed (for example, a call to a function that can only fail on a subset of inputs with an input that is known to be safe) the cantFail functions can be used to remove the error type, simplifying control flow.

StringError

Many kinds of errors have no recovery strategy, the only action that can be taken is to report them to the user so that the user can attempt to fix the environment. In this case representing the error as a string makes perfect sense. LLVM provides the `StringError` class for this purpose. It takes two arguments: A string error message, and an equivalent `std::error_code` for interoperability. It also provides a `createStringError` function to simplify common usage of this class:

```
// These two lines of code are equivalent:
make_error<StringError>("Bad executable", errc::executable_format_error);
createStringError(errc::executable_format_error, "Bad executable");
```

If you're certain that the error you're building will never need to be converted to a `std::error_code` you can use the `inconvertibleErrorCode()` function:

```
createStringError(inconvertibleErrorCode(), "Bad executable");
```

This should be done only after careful consideration. If any attempt is made to convert this error to a `std::error_code` it will trigger immediate program termination. Unless you are certain that your errors will not need interoperability you should look for an existing `std::error_code` that you can convert to, and even (as painful as it is) consider introducing a new one as a stopgap measure.

`createStringError` can take `printf` style format specifiers to provide a formatted message:

```
createStringError(errc::executable_format_error,
                  "Bad executable: %s", FileName);
```

Interoperability with std::error_code and ErrorOr

Many existing LLVM APIs use `std::error_code` and its partner `ErrorOr<T>` (which plays the same role as `Expected<T>`, but wraps a `std::error_code` rather than an `Error`). The infectious nature of error types means that an attempt to change one of these functions to return `Error` or `Expected<T>` instead often results in an avalanche of changes to callers, callers of callers, and so on. (The first such attempt, returning an `Error` from MachOObjectFile's constructor, was abandoned after the diff reached 3000 lines, impacted half a dozen libraries, and was still growing).

To solve this problem, the `Error`/`std::error_code` interoperability requirement was introduced. Two pairs of functions allow any `Error` value to be converted to a `std::error_code`, any `Expected<T>` to be converted to an `ErrorOr<T>`, and vice versa:

```
std::error_code errorToErrorCode(Error Err);
Error errorCodeToError(std::error_code EC);

template <typename T> ErrorOr<T> expectedToErrorOr(Expected<T> TOrErr);
template <typename T> Expected<T> errorOrToExpected(ErrorOr<T> TOrEC);
```

Using these APIs it is easy to make surgical patches that update individual functions from `std::error_code` to `Error`, and from `ErrorOr<T>` to `Expected<T>`.

Returning Errors from error handlers

Error recovery attempts may themselves fail. For that reason, `handleErrors` actually recognises three different forms of handler signature:

```
// Error must be handled, no new errors produced:
void(UserDefinedError &E);
```

```
// Error must be handled, new errors can be produced:
Error(UserDefinedError &E);

// Original error can be inspected, then re-wrapped and returned (or a new
// error can be produced):
Error(std::unique_ptr<UserDefinedError> E);
```

Any error returned from a handler will be returned from the `handleErrors` function so that it can be handled itself, or propagated up the stack.

Using ExitOnError to simplify tool code

Library code should never call `exit` for a recoverable error, however in tool code (especially command line tools) this can be a reasonable approach. Calling `exit` upon encountering an error dramatically simplifies control flow as the error no longer needs to be propagated up the stack. This allows code to be written in straight-line style, as long as each fallible call is wrapped in a check and call to exit. The `ExitOnError` class supports this pattern by providing call operators that inspect `Error` values, stripping the error away in the success case and logging to `stderr` then exiting in the failure case.

To use this class, declare a global `ExitOnError` variable in your program:

```
ExitOnError ExitOnErr;
```

Calls to fallible functions can then be wrapped with a call to `ExitOnErr`, turning them into non-failing calls:

```
Error mayFail();
Expected<int> mayFail2();

void foo() {
  ExitOnErr(mayFail());
  int X = ExitOnErr(mayFail2());
}
```

On failure, the error's log message will be written to `stderr`, optionally preceded by a string "banner" that can be set by calling the setBanner method. A mapping can also be supplied from `Error` values to exit codes using the `setExitCodeMapper` method:

```
int main(int argc, char *argv[]) {
  ExitOnErr.setBanner(std::string(argv[0]) + " error:");
  ExitOnErr.setExitCodeMapper(
    [](const Error &Err) {
      if (Err.isA<BadFileFormat>())
        return 2;
      return 1;
    });
```

Use `ExitOnError` in your tool code where possible as it can greatly improve readability.

Using cantFail to simplify safe callsites

Some functions may only fail for a subset of their inputs, so calls using known safe inputs can be assumed to succeed.

The cantFail functions encapsulate this by wrapping an assertion that their argument is a success value and, in the case of Expected<T>, unwrapping the T value:

```
Error onlyFailsForSomeXValues(int X);
Expected<int> onlyFailsForSomeXValues2(int X);

void foo() {
  cantFail(onlyFailsForSomeXValues(KnownSafeValue));
  int Y = cantFail(onlyFailsForSomeXValues2(KnownSafeValue));
  ...
}
```

Like the ExitOnError utility, cantFail simplifies control flow. Their treatment of error cases is very different however: Where ExitOnError is guaranteed to terminate the program on an error input, cantFail simply asserts that the result is success. In debug builds this will result in an assertion failure if an error is encountered. In release builds the behavior of cantFail for failure values is undefined. As such, care must be taken in the use of cantFail: clients must be certain that a cantFail wrapped call really can not fail with the given arguments.

Use of the cantFail functions should be rare in library code, but they are likely to be of more use in tool and unit-test code where inputs and/or mocked-up classes or functions may be known to be safe.

Fallible constructors

Some classes require resource acquisition or other complex initialization that can fail during construction. Unfortunately constructors can't return errors, and having clients test objects after they're constructed to ensure that they're valid is error prone as it's all too easy to forget the test. To work around this, use the named constructor idiom and return an `Expected<T>`:

```
class Foo {
public:

  static Expected<Foo> Create(Resource R1, Resource R2) {
    Error Err = Error::success();
    Foo F(R1, R2, Err);
    if (Err)
      return std::move(Err);
    return std::move(F);
  }

private:

  Foo(Resource R1, Resource R2, Error &Err) {
    ErrorAsOutParameter EAO(&Err);
    if (auto Err2 = R1.acquire()) {
      Err = std::move(Err2);
      return;
    }
    Err = R2.acquire();
  }
};
```

Here, the named constructor passes an `Error` by reference into the actual constructor, which the constructor can then use to return errors. The `ErrorAsOutParameter` utility sets the `Error` value's checked flag on entry to the constructor so that the error can be assigned to, then resets it on exit to force the client (the named constructor) to check the error.

By using this idiom, clients attempting to construct a Foo receive either a well-formed Foo or an Error, never an object in an invalid state.

Propagating and consuming errors based on types

In some contexts, certain types of error are known to be benign. For example, when walking an archive, some clients may be happy to skip over badly formatted object files rather than terminating the walk immediately. Skipping badly formatted objects could be achieved using an elaborate handler method, but the Error.h header provides two utilities that make this idiom much cleaner: the type inspection method, `isA`, and the `consumeError` function:

```
Error walkArchive(Archive A) {
  for (unsigned I = 0; I != A.numMembers(); ++I) {
    auto ChildOrErr = A.getMember(I);
    if (auto Err = ChildOrErr.takeError()) {
      if (Err.isA<BadFileFormat>())
        consumeError(std::move(Err))
      else
        return Err;
```

```
    }
    auto &Child = *ChildOrErr;
    // Use Child
    ...
  }
  return Error::success();
}
```

Concatenating Errors with joinErrors

In the archive walking example above `BadFileFormat` errors are simply consumed and ignored. If the client had wanted report these errors after completing the walk over the archive they could use the `joinErrors` utility:

```
Error walkArchive(Archive A) {
  Error DeferredErrs = Error::success();
  for (unsigned I = 0; I != A.numMembers(); ++I) {
    auto ChildOrErr = A.getMember(I);
    if (auto Err = ChildOrErr.takeError())
      if (Err.isA<BadFileFormat>())
        DeferredErrs = joinErrors(std::move(DeferredErrs), std::move(Err));
      else
        return Err;
    auto &Child = *ChildOrErr;
    // Use Child
    ...
  }
  return DeferredErrs;
}
```

The `joinErrors` routine builds a special error type called `ErrorList`, which holds a list of user defined errors. The `handleErrors` routine recognizes this type and will attempt to handle each of the contained errors in order. If all contained errors can be handled, `handleErrors` will return `Error::success()`, otherwise `handleErrors` will concatenate the remaining errors and return the resulting `ErrorList`.

Building fallible iterators and iterator ranges

The archive walking examples above retrieve archive members by index, however this requires considerable boiler-plate for iteration and error checking. We can clean this up by using the "fallible iterator" pattern, which supports the following natural iteration idiom for fallible containers like Archive:

```
Error Err = Error::success();
for (auto &Child : Ar->children(Err)) {
  // Use Child - only enter the loop when it's valid

  // Allow early exit from the loop body, since we know that Err is success
  // when we're inside the loop.
  if (BailOutOn(Child))
    return;

  ...
}
// Check Err after the loop to ensure it didn't break due to an error.
if (Err)
  return Err;
```

To enable this idiom, iterators over fallible containers are written in a natural style, with their `++` and `--` operators replaced with fallible `Error inc()` and `Error dec()` functions. E.g.:

```
class FallibleChildIterator {
public:
  FallibleChildIterator(Archive &A, unsigned ChildIdx);
  Archive::Child &operator*();
  friend bool operator==(const ArchiveIterator &LHS,
                         const ArchiveIterator &RHS);
```

```
  // operator++/operator-- replaced with fallible increment / decrement:
  Error inc() {
    if (!A.childValid(ChildIdx + 1))
      return make_error<BadArchiveMember>(...);
    ++ChildIdx;
    return Error::success();
  }

  Error dec() { ... }
};
```

Instances of this kind of fallible iterator interface are then wrapped with the fallible_iterator utility which provides `operator++` and `operator--`, returning any errors via a reference passed in to the wrapper at construction time. The fallible_iterator wrapper takes care of (a) jumping to the end of the range on error, and (b) marking the error as checked whenever an iterator is compared to `end` and found to be inequal (in particular: this marks the error as checked throughout the body of a range-based for loop), enabling early exit from the loop without redundant error checking.

Instances of the fallible iterator interface (e.g. FallibleChildIterator above) are wrapped using the `make_fallible_itr` and `make_fallible_end` functions. E.g.:

```
class Archive {
public:
  using child_iterator = fallible_iterator<FallibleChildIterator>;

  child_iterator child_begin(Error &Err) {
    return make_fallible_itr(FallibleChildIterator(*this, 0), Err);
  }

  child_iterator child_end() {
    return make_fallible_end(FallibleChildIterator(*this, size()));
  }

  iterator_range<child_iterator> children(Error &Err) {
    return make_range(child_begin(Err), child_end());
  }
};
```

Using the fallible_iterator utility allows for both natural construction of fallible iterators (using failing `inc` and `dec` operations) and relatively natural use of c++ iterator/loop idioms.

More information on Error and its related utilities can be found in the Error.h header file.

Passing functions and other callable objects

Sometimes you may want a function to be passed a callback object. In order to support lambda expressions and other function objects, you should not use the traditional C approach of taking a function pointer and an opaque cookie:

```
void takeCallback(bool (*Callback)(Function *, void *), void *Cookie);
```

Instead, use one of the following approaches:

Function template

If you don't mind putting the definition of your function into a header file, make it a function template that is templated on the callable type.

```
template<typename Callable>
void takeCallback(Callable Callback) {
  Callback(1, 2, 3);
}
```

The `function_ref` class template

The `function_ref` (doxygen) class template represents a reference to a callable object, templated over the type of the callable. This is a good choice for passing a callback to a function, if you don't need to hold onto the callback after the function returns. In this way, `function_ref` is to `std::function` as `StringRef` is to `std::string`.

`function_ref<Ret(Param1, Param2, ...)>` can be implicitly constructed from any callable object that can be called with arguments of type `Param1`, `Param2`, ..., and returns a value that can be converted to type `Ret`. For example:

```
void visitBasicBlocks(Function *F, function_ref<bool (BasicBlock*)> Callback) {
  for (BasicBlock &BB : *F)
    if (Callback(&BB))
      return;
}
```

can be called using:

```
visitBasicBlocks(F, [&](BasicBlock *BB) {
  if (process(BB))
    return isEmpty(BB);
  return false;
});
```

Note that a `function_ref` object contains pointers to external memory, so it is not generally safe to store an instance of the class (unless you know that the external storage will not be freed). If you need this ability, consider using `std::function`. `function_ref` is small enough that it should always be passed by value.

The `LLVM_DEBUG()` macro and `-debug` option

Often when working on your pass you will put a bunch of debugging printouts and other code into your pass. After you get it working, you want to remove it, but you may need it again in the future (to work out new bugs that you run across).

Naturally, because of this, you don't want to delete the debug printouts, but you don't want them to always be noisy. A standard compromise is to comment them out, allowing you to enable them if you need them in the future.

The `llvm/Support/Debug.h` (doxygen) file provides a macro named `LLVM_DEBUG()` that is a much nicer solution to this problem. Basically, you can put arbitrary code into the argument of the `LLVM_DEBUG` macro, and it is only executed if '`opt`' (or any other tool) is run with the '`-debug`' command line argument:

```
LLVM_DEBUG(dbgs() << "I am here!\n");
```

Then you can run your pass like this:

```
$ opt < a.bc > /dev/null -mypass
<no output>
$ opt < a.bc > /dev/null -mypass -debug
I am here!
```

Using the `LLVM_DEBUG()` macro instead of a home-brewed solution allows you to not have to create "yet another" command line option for the debug output for your pass. Note that `LLVM_DEBUG()` macros are disabled for non-asserts builds, so they do not cause a performance impact at all (for the same reason, they should also not contain side-effects!).

One additional nice thing about the `LLVM_DEBUG()` macro is that you can enable or disable it directly in gdb. Just use "`set DebugFlag=0`" or "`set DebugFlag=1`" from the gdb if the program is running. If the program hasn't been started yet, you can always just run it with `-debug`.

Fine grained debug info with `DEBUG_TYPE` and the `-debug-only` option

Sometimes you may find yourself in a situation where enabling `-debug` just turns on too much information (such as when working on the code generator). If you want to enable debug information with more fine-grained control, you should define the `DEBUG_TYPE` macro and use the `-debug-only` option as follows:

```
#define DEBUG_TYPE "foo"
LLVM_DEBUG(dbgs() << "'foo' debug type\n");
#undef  DEBUG_TYPE
#define DEBUG_TYPE "bar"
LLVM_DEBUG(dbgs() << "'bar' debug type\n");
#undef  DEBUG_TYPE
```

Then you can run your pass like this:

```
$ opt < a.bc > /dev/null -mypass
<no output>
$ opt < a.bc > /dev/null -mypass -debug
'foo' debug type
'bar' debug type
$ opt < a.bc > /dev/null -mypass -debug-only=foo
'foo' debug type
$ opt < a.bc > /dev/null -mypass -debug-only=bar
'bar' debug type
$ opt < a.bc > /dev/null -mypass -debug-only=foo,bar
'foo' debug type
'bar' debug type
```

Of course, in practice, you should only set `DEBUG_TYPE` at the top of a file, to specify the debug type for the entire module. Be careful that you only do this after including Debug.h and not around any #include of headers. Also, you should use names more meaningful than “foo” and “bar”, because there is no system in place to ensure that names do not conflict. If two different modules use the same string, they will all be turned on when the name is specified. This allows, for example, all debug information for instruction scheduling to be enabled with `-debug-only=InstrSched`, even if the source lives in multiple files. The name must not include a comma (,) as that is used to separate the arguments of the `-debug-only` option.

For performance reasons, -debug-only is not available in optimized build (`--enable-optimized`) of LLVM.

The `DEBUG_WITH_TYPE` macro is also available for situations where you would like to set `DEBUG_TYPE`, but only for one specific `DEBUG` statement. It takes an additional first parameter, which is the type to use. For example, the preceding example could be written as:

```
DEBUG_WITH_TYPE("foo", dbgs() << "'foo' debug type\n");
DEBUG_WITH_TYPE("bar", dbgs() << "'bar' debug type\n");
```

The **`Statistic`** class & **`-stats`** option

The `llvm/ADT/Statistic.h` (doxygen) file provides a class named `Statistic` that is used as a unified way to keep track of what the LLVM compiler is doing and how effective various optimizations are. It is useful to see what optimizations are contributing to making a particular program run faster.

Often you may run your pass on some big program, and you’re interested to see how many times it makes a certain transformation. Although you can do this with hand inspection, or some ad-hoc method, this is a real pain and not very useful for big programs. Using the `Statistic` class makes it very easy to keep track of this information, and the calculated information is presented in a uniform manner with the rest of the passes being executed.

There are many examples of `Statistic` uses, but the basics of using it are as follows:

Define your statistic like this:

```
#define DEBUG_TYPE "mypassname"   // This goes before any #includes.
STATISTIC(NumXForms, "The # of times I did stuff");
```

The `STATISTIC` macro defines a static variable, whose name is specified by the first argument. The pass name is taken from the `DEBUG_TYPE` macro, and the description is taken from the second argument. The variable defined ("NumXForms" in this case) acts like an unsigned integer.

Whenever you make a transformation, bump the counter:

```
++NumXForms;   // I did stuff!
```

That's all you have to do. To get '`opt`' to print out the statistics gathered, use the '`-stats`' option:

```
$ opt -stats -mypassname < program.bc > /dev/null
... statistics output ...
```

Note that in order to use the '`-stats`' option, LLVM must be compiled with assertions enabled.

When running `opt` on a C file from the SPEC benchmark suite, it gives a report that looks like this:

```
  7646 bitcodewriter   - Number of normal instructions
   725 bitcodewriter   - Number of oversized instructions
129996 bitcodewriter   - Number of bitcode bytes written
  2817 raise           - Number of insts DCEd or constprop'd
  3213 raise           - Number of cast-of-self removed
  5046 raise           - Number of expression trees converted
    75 raise           - Number of other getelementptr's formed
   138 raise           - Number of load/store peepholes
    42 deadtypeelim    - Number of unused typenames removed from symtab
   392 funcresolve     - Number of varargs functions resolved
    27 globaldce       - Number of global variables removed
     2 adce            - Number of basic blocks removed
   134 cee             - Number of branches revectored
    49 cee             - Number of setcc instruction eliminated
   532 gcse            - Number of loads removed
  2919 gcse            - Number of instructions removed
    86 indvars         - Number of canonical indvars added
    87 indvars         - Number of aux indvars removed
    25 instcombine     - Number of dead inst eliminate
   434 instcombine     - Number of insts combined
   248 licm            - Number of load insts hoisted
  1298 licm            - Number of insts hoisted to a loop pre-header
     3 licm            - Number of insts hoisted to multiple loop preds (bad, no loop pre-heade
    75 mem2reg         - Number of alloca's promoted
  1444 cfgsimplify     - Number of blocks simplified
```

Obviously, with so many optimizations, having a unified framework for this stuff is very nice. Making your pass fit well into the framework makes it more maintainable and useful.

Adding debug counters to aid in debugging your code

Sometimes, when writing new passes, or trying to track down bugs, it is useful to be able to control whether certain things in your pass happen or not. For example, there are times the minimization tooling can only easily give you large testcases. You would like to narrow your bug down to a specific transformation happening or not happening, automatically, using bisection. This is where debug counters help. They provide a framework for making parts of your code only execute a certain number of times.

The `llvm/Support/DebugCounter.h` (doxygen) file provides a class named `DebugCounter` that can be used to create command line counter options that control execution of parts of your code.

Define your DebugCounter like this:

```
DEBUG_COUNTER(DeleteAnInstruction, "passname-delete-instruction",
              "Controls which instructions get delete");
```

The `DEBUG_COUNTER` macro defines a static variable, whose name is specified by the first argument. The name of the counter (which is used on the command line) is specified by the second argument, and the description used in the help is specified by the third argument.

Whatever code you want that control, use `DebugCounter::shouldExecute` to control it.

```
if (DebugCounter::shouldExecute(DeleteAnInstruction))
  I->eraseFromParent();
```

That's all you have to do. Now, using opt, you can control when this code triggers using the '`--debug-counter`' option. There are two counters provided, `skip` and `count`. `skip` is the number of times to skip execution of the codepath. `count` is the number of times, once we are done skipping, to execute the codepath.

```
$ opt --debug-counter=passname-delete-instruction-skip=1,passname-delete-instruction-count=2 -p
```

This will skip the above code the first time we hit it, then execute it twice, then skip the rest of the executions.

So if executed on the following code:

```
%1 = add i32 %a, %b
%2 = add i32 %a, %b
%3 = add i32 %a, %b
%4 = add i32 %a, %b
```

It would delete number `%2` and `%3`.

A utility is provided in *`utils/bisect-skip-count`* to binary search skip and count arguments. It can be used to automatically minimize the skip and count for a debug-counter variable.

Viewing graphs while debugging code

Several of the important data structures in LLVM are graphs: for example CFGs made out of LLVM BasicBlocks, CFGs made out of LLVM MachineBasicBlocks, and Instruction Selection DAGs. In many cases, while debugging various parts of the compiler, it is nice to instantly visualize these graphs.

LLVM provides several callbacks that are available in a debug build to do exactly that. If you call the `Function::viewCFG()` method, for example, the current LLVM tool will pop up a window containing the CFG for the function where each basic block is a node in the graph, and each node contains the instructions in the block. Similarly, there also exists `Function::viewCFGOnly()` (does not include the instructions), the `MachineFunction::viewCFG()` and `MachineFunction::viewCFGOnly()`, and the `SelectionDAG::viewGraph()` methods. Within GDB, for example, you can usually use something like `call DAG.viewGraph()` to pop up a window. Alternatively, you can sprinkle calls to these functions in your code in places you want to debug.

Getting this to work requires a small amount of setup. On Unix systems with X11, install the graphviz toolkit, and make sure 'dot' and 'gv' are in your path. If you are running on macOS, download and install the macOS Graphviz program and add `/Applications/Graphviz.app/Contents/MacOS/` (or wherever you install it) to your path. The programs need not be present when configuring, building or running LLVM and can simply be installed when needed during an active debug session.

`SelectionDAG` has been extended to make it easier to locate interesting nodes in large complex graphs. From gdb, if you `call DAG.setGraphColor(node, "color")`, then the next `call DAG.viewGraph()` would highlight the node in the specified color (choices of colors can be found at colors.) More complex node attributes can be provided with `call DAG.setGraphAttrs(node, "attributes")` (choices can be found at Graph attributes.) If you want to restart and clear all the current graph attributes, then you can `call DAG.clearGraphAttrs()`.

Note that graph visualization features are compiled out of Release builds to reduce file size. This means that you need a Debug+Asserts or Release+Asserts build to use these features.

Picking the Right Data Structure for a Task

LLVM has a plethora of data structures in the `llvm/ADT/` directory, and we commonly use STL data structures. This section describes the trade-offs you should consider when you pick one.

The first step is a choose your own adventure: do you want a sequential container, a set-like container, or a map-like container? The most important thing when choosing a container is the algorithmic properties of how you plan to access the container. Based on that, you should use:

- a map-like container if you need efficient look-up of a value based on another value. Map-like containers also support efficient queries for containment (whether a key is in the map). Map-like containers generally do not support efficient reverse mapping (values to keys). If you need that, use two maps. Some map-like containers also support efficient iteration through the keys in sorted order. Map-like containers are the most expensive sort, only use them if you need one of these capabilities.
- a set-like container if you need to put a bunch of stuff into a container that automatically eliminates duplicates. Some set-like containers support efficient iteration through the elements in sorted order. Set-like containers are more expensive than sequential containers.
- a sequential container provides the most efficient way to add elements and keeps track of the order they are added to the collection. They permit duplicates and support efficient iteration, but do not support efficient look-up based on a key.
- a string container is a specialized sequential container or reference structure that is used for character or byte arrays.
- a bit container provides an efficient way to store and perform set operations on sets of numeric id's, while automatically eliminating duplicates. Bit containers require a maximum of 1 bit for each identifier you want to store.

Once the proper category of container is determined, you can fine tune the memory use, constant factors, and cache behaviors of access by intelligently picking a member of the category. Note that constant factors and cache behavior can be a big deal. If you have a vector that usually only contains a few elements (but could contain many), for example, it's much better to use SmallVector than vector. Doing so avoids (relatively) expensive malloc/free calls, which dwarf the cost of adding the elements to the container.

Sequential Containers (std::vector, std::list, etc)

There are a variety of sequential containers available for you, based on your needs. Pick the first in this section that will do what you want.

llvm/ADT/ArrayRef.h

The `llvm::ArrayRef` class is the preferred class to use in an interface that accepts a sequential list of elements in memory and just reads from them. By taking an `ArrayRef`, the API can be passed a fixed size array, an `std::vector`, an `llvm::SmallVector` and anything else that is contiguous in memory.

Fixed Size Arrays

Fixed size arrays are very simple and very fast. They are good if you know exactly how many elements you have, or you have a (low) upper bound on how many you have.

Heap Allocated Arrays

Heap allocated arrays (`new[] + delete[]`) are also simple. They are good if the number of elements is variable, if you know how many elements you will need before the array is allocated, and if the array is usually large (if not, consider a SmallVector). The cost of a heap allocated array is the cost of the new/delete (aka malloc/free). Also note that if you are allocating an array of a type with a constructor,

the constructor and destructors will be run for every element in the array (re-sizable vectors only construct those elements actually used).

llvm/ADT/TinyPtrVector.h

`TinyPtrVector<Type>` is a highly specialized collection class that is optimized to avoid allocation in the case when a vector has zero or one elements. It has two major restrictions: 1) it can only hold values of pointer type, and 2) it cannot hold a null pointer.

Since this container is highly specialized, it is rarely used.

llvm/ADT/SmallVector.h

`SmallVector<Type, N>` is a simple class that looks and smells just like `vector<Type>`: it supports efficient iteration, lays out elements in memory order (so you can do pointer arithmetic between elements), supports efficient push_back/pop_back operations, supports efficient random access to its elements, etc.

The main advantage of SmallVector is that it allocates space for some number of elements (N) in the object itself. Because of this, if the SmallVector is dynamically smaller than N, no malloc is performed. This can be a big win in cases where the malloc/free call is far more expensive than the code that fiddles around with the elements.

This is good for vectors that are "usually small" (e.g. the number of predecessors/successors of a block is usually less than 8). On the other hand, this makes the size of the SmallVector itself large, so you don't want to allocate lots of them (doing so will waste a lot of space). As such, SmallVectors are most useful when on the stack.

In the absence of a well-motivated choice for the number of inlined elements `N`, it is recommended to use `SmallVector<T>` (that is, omitting the `N`). This will choose a default number of inlined elements reasonable for allocation on the stack (for example, trying to keep `sizeof(SmallVector<T>)` around 64 bytes).

SmallVector also provides a nice portable and efficient replacement for `alloca`.

SmallVector has grown a few other minor advantages over std::vector, causing `SmallVector<Type, 0>` to be preferred over `std::vector<Type>`.

1. std::vector is exception-safe, and some implementations have pessimizations that copy elements when SmallVector would move them.
2. SmallVector understands `std::is_trivially_copyable<Type>` and uses realloc aggressively.
3. Many LLVM APIs take a SmallVectorImpl as an out parameter (see the note below).
4. SmallVector with N equal to 0 is smaller than std::vector on 64-bit platforms, since it uses `unsigned` (instead of `void*`) for its size and capacity.

> Note
>
> Prefer to use `ArrayRef<T>` or `SmallVectorImpl<T>` as a parameter type.
>
> It's rarely appropriate to use `SmallVector<T, N>` as a parameter type. If an API only reads from the vector, it should use ArrayRef. Even if an API updates the vector the "small size" is unlikely to be relevant; such an API should use the `SmallVectorImpl<T>` class, which is the "vector header" (and methods) without the elements allocated after it. Note that `SmallVector<T, N>` inherits from `SmallVectorImpl<T>` so the conversion is implicit and costs nothing. E.g.

```
// DISCOURAGED: Clients cannot pass e.g. raw arrays.
hardcodedContiguousStorage(const SmallVectorImpl<Foo> &In);
// ENCOURAGED: Clients can pass any contiguous storage of Foo.
allowsAnyContiguousStorage(ArrayRef<Foo> In);

void someFunc1() {
  Foo Vec[] = { /* ... */ };
  hardcodedContiguousStorage(Vec); // Error.
  allowsAnyContiguousStorage(Vec); // Works.
}
```

```
// DISCOURAGED: Clients cannot pass e.g. SmallVector<Foo, 8>.
hardcodedSmallSize(SmallVector<Foo, 2> &Out);
// ENCOURAGED: Clients can pass any SmallVector<Foo, N>.
allowsAnySmallSize(SmallVectorImpl<Foo> &Out);

void someFunc2() {
  SmallVector<Foo, 8> Vec;
  hardcodedSmallSize(Vec); // Error.
  allowsAnySmallSize(Vec); // Works.
}
```

Even though it has "`Impl`" in the name, SmallVectorImpl is widely used and is no longer "private to the implementation". A name like `SmallVectorHeader` might be more appropriate.

<vector>

`std::vector<T>` is well loved and respected. However, `SmallVector<T, 0>` is often a better option due to the advantages listed above. std::vector is still useful when you need to store more than `UINT32_MAX` elements or when interfacing with code that expects vectors :).

One worthwhile note about std::vector: avoid code like this:

```
for ( ... ) {
   std::vector<foo> V;
   // make use of V.
}
```

Instead, write this as:

```
std::vector<foo> V;
for ( ... ) {
   // make use of V.
   V.clear();
}
```

Doing so will save (at least) one heap allocation and free per iteration of the loop.

<deque>

`std::deque` is, in some senses, a generalized version of `std::vector`. Like `std::vector`, it provides constant time random access and other similar properties, but it also provides efficient access to the front of the list. It does not guarantee continuity of elements within memory.

In exchange for this extra flexibility, `std::deque` has significantly higher constant factor costs than `std::vector`. If possible, use `std::vector` or something cheaper.

<list>

`std::list` is an extremely inefficient class that is rarely useful. It performs a heap allocation for every element inserted into it, thus having an extremely high constant factor, particularly for small data types. `std::list` also only supports bidirectional iteration, not random access iteration.

In exchange for this high cost, std::list supports efficient access to both ends of the list (like `std::deque`, but unlike `std::vector` or `SmallVector`). In addition, the iterator invalidation characteristics of std::list are stronger than that of a vector class: inserting or removing an element into the list does not invalidate iterator or pointers to other elements in the list.

llvm/ADT/ilist.h

`ilist<T>` implements an 'intrusive' doubly-linked list. It is intrusive, because it requires the element to store and provide access to the prev/next pointers for the list.

`ilist` has the same drawbacks as `std::list`, and additionally requires an `ilist_traits` implementation for the element type, but it provides some novel characteristics. In particular, it can efficiently store

polymorphic objects, the traits class is informed when an element is inserted or removed from the list, and `ilists` are guaranteed to support a constant-time splice operation.

These properties are exactly what we want for things like `Instructions` and basic blocks, which is why these are implemented with `ilists`.

Related classes of interest are explained in the following subsections:

- ilist_traits
- iplist
- llvm/ADT/ilist_node.h
- Sentinels

llvm/ADT/PackedVector.h

Useful for storing a vector of values using only a few number of bits for each value. Apart from the standard operations of a vector-like container, it can also perform an 'or' set operation.

For example:

```
enum State {
    None = 0x0,
    FirstCondition = 0x1,
    SecondCondition = 0x2,
    Both = 0x3
};

State get() {
    PackedVector<State, 2> Vec1;
    Vec1.push_back(FirstCondition);

    PackedVector<State, 2> Vec2;
    Vec2.push_back(SecondCondition);

    Vec1 |= Vec2;
    return Vec1[0]; // returns 'Both'.
}
```

ilist_traits

`ilist_traits<T>` is `ilist<T>`'s customization mechanism. `iplist<T>` (and consequently `ilist<T>`) publicly derive from this traits class.

iplist

`iplist<T>` is `ilist<T>`'s base and as such supports a slightly narrower interface. Notably, inserters from `T&` are absent.

`ilist_traits<T>` is a public base of this class and can be used for a wide variety of customizations.

llvm/ADT/ilist_node.h

`ilist_node<T>` implements the forward and backward links that are expected by the `ilist<T>` (and analogous containers) in the default manner.

`ilist_node<T>`s are meant to be embedded in the node type `T`, usually `T` publicly derives from `ilist_node<T>`.

Sentinels

`ilists` have another specialty that must be considered. To be a good citizen in the C++ ecosystem, it needs to support the standard container operations, such as `begin` and `end` iterators, etc. Also, the `operator--` must work correctly on the `end` iterator in the case of non-empty `ilists`.

The only sensible solution to this problem is to allocate a so-called sentinel along with the intrusive list, which serves as the `end` iterator, providing the back-link to the last element. However conforming to the C++ convention it is illegal to `operator++` beyond the sentinel and it also must not be dereferenced.

These constraints allow for some implementation freedom to the `ilist` how to allocate and store the sentinel. The corresponding policy is dictated by `ilist_traits<T>`. By default a `T` gets heap-allocated whenever the need for a sentinel arises.

While the default policy is sufficient in most cases, it may break down when `T` does not provide a default constructor. Also, in the case of many instances of `ilists`, the memory overhead of the associated sentinels is wasted. To alleviate the situation with numerous and voluminous `T`-sentinels, sometimes a trick is employed, leading to ghostly sentinels.

Ghostly sentinels are obtained by specially-crafted `ilist_traits<T>` which superpose the sentinel with the `ilist` instance in memory. Pointer arithmetic is used to obtain the sentinel, which is relative to the `ilist`'s `this` pointer. The `ilist` is augmented by an extra pointer, which serves as the back-link of the sentinel. This is the only field in the ghostly sentinel which can be legally accessed.

Other Sequential Container options

Other STL containers are available, such as `std::string`.

There are also various STL adapter classes such as `std::queue`, `std::priority_queue`, `std::stack`, etc. These provide simplified access to an underlying container but don't affect the cost of the container itself.

String-like containers

There are a variety of ways to pass around and use strings in C and C++, and LLVM adds a few new options to choose from. Pick the first option on this list that will do what you need, they are ordered according to their relative cost.

Note that it is generally preferred to not pass strings around as `const char*`'s. These have a number of problems, including the fact that they cannot represent embedded nul ("0") characters, and do not have a length available efficiently. The general replacement for '`const char*`' is StringRef.

For more information on choosing string containers for APIs, please see Passing Strings.

llvm/ADT/StringRef.h

The StringRef class is a simple value class that contains a pointer to a character and a length, and is quite related to the ArrayRef class (but specialized for arrays of characters). Because StringRef carries a length with it, it safely handles strings with embedded nul characters in it, getting the length does not require a strlen call, and it even has very convenient APIs for slicing and dicing the character range that it represents.

StringRef is ideal for passing simple strings around that are known to be live, either because they are C string literals, std::string, a C array, or a SmallVector. Each of these cases has an efficient implicit conversion to StringRef, which doesn't result in a dynamic strlen being executed.

StringRef has a few major limitations which make more powerful string containers useful:

1. You cannot directly convert a StringRef to a 'const char*' because there is no way to add a trailing nul (unlike the .c_str() method on various stronger classes).
2. StringRef doesn't own or keep alive the underlying string bytes. As such it can easily lead to dangling pointers, and is not suitable for embedding in datastructures in most cases (instead, use an std::string or something like that).
3. For the same reason, StringRef cannot be used as the return value of a method if the method "computes" the result string. Instead, use std::string.

4. StringRef's do not allow you to mutate the pointed-to string bytes and it doesn't allow you to insert or remove bytes from the range. For editing operations like this, it interoperates with the Twine class.

Because of its strengths and limitations, it is very common for a function to take a StringRef and for a method on an object to return a StringRef that points into some string that it owns.

llvm/ADT/Twine.h

The Twine class is used as an intermediary datatype for APIs that want to take a string that can be constructed inline with a series of concatenations. Twine works by forming recursive instances of the Twine datatype (a simple value object) on the stack as temporary objects, linking them together into a tree which is then linearized when the Twine is consumed. Twine is only safe to use as the argument to a function, and should always be a const reference, e.g.:

```
void foo(const Twine &T);
...
StringRef X = ...
unsigned i = ...
foo(X + "." + Twine(i));
```

This example forms a string like "blarg.42" by concatenating the values together, and does not form intermediate strings containing "blarg" or "blarg.".

Because Twine is constructed with temporary objects on the stack, and because these instances are destroyed at the end of the current statement, it is an inherently dangerous API. For example, this simple variant contains undefined behavior and will probably crash:

```
void foo(const Twine &T);
...
StringRef X = ...
unsigned i = ...
const Twine &Tmp = X + "." + Twine(i);
foo(Tmp);
```

... because the temporaries are destroyed before the call. That said, Twine's are much more efficient than intermediate std::string temporaries, and they work really well with StringRef. Just be aware of their limitations.

llvm/ADT/SmallString.h

SmallString is a subclass of SmallVector that adds some convenience APIs like += that takes StringRef's. SmallString avoids allocating memory in the case when the preallocated space is enough to hold its data, and it calls back to general heap allocation when required. Since it owns its data, it is very safe to use and supports full mutation of the string.

Like SmallVector's, the big downside to SmallString is their sizeof. While they are optimized for small strings, they themselves are not particularly small. This means that they work great for temporary scratch buffers on the stack, but should not generally be put into the heap: it is very rare to see a SmallString as the member of a frequently-allocated heap data structure or returned by-value.

std::string

The standard C++ std::string class is a very general class that (like SmallString) owns its underlying data. sizeof(std::string) is very reasonable so it can be embedded into heap data structures and returned by-value. On the other hand, std::string is highly inefficient for inline editing (e.g. concatenating a bunch of stuff together) and because it is provided by the standard library, its performance characteristics depend a lot of the host standard library (e.g. libc++ and MSVC provide a highly optimized string class, GCC contains a really slow implementation).

The major disadvantage of std::string is that almost every operation that makes them larger can allocate memory, which is slow. As such, it is better to use SmallVector or Twine as a scratch buffer,

but then use std::string to persist the result.

Set-Like Containers (std::set, SmallSet, SetVector, etc)

Set-like containers are useful when you need to canonicalize multiple values into a single representation. There are several different choices for how to do this, providing various trade-offs.

A sorted 'vector'

If you intend to insert a lot of elements, then do a lot of queries, a great approach is to use an std::vector (or other sequential container) with std::sort+std::unique to remove duplicates. This approach works really well if your usage pattern has these two distinct phases (insert then query), and can be coupled with a good choice of sequential container.

This combination provides the several nice properties: the result data is contiguous in memory (good for cache locality), has few allocations, is easy to address (iterators in the final vector are just indices or pointers), and can be efficiently queried with a standard binary search (e.g. `std::lower_bound`; if you want the whole range of elements comparing equal, use `std::equal_range`).

llvm/ADT/SmallSet.h

If you have a set-like data structure that is usually small and whose elements are reasonably small, a `SmallSet<Type, N>` is a good choice. This set has space for N elements in place (thus, if the set is dynamically smaller than N, no malloc traffic is required) and accesses them with a simple linear search. When the set grows beyond N elements, it allocates a more expensive representation that guarantees efficient access (for most types, it falls back to std::set, but for pointers it uses something far better, SmallPtrSet.

The magic of this class is that it handles small sets extremely efficiently, but gracefully handles extremely large sets without loss of efficiency.

llvm/ADT/SmallPtrSet.h

`SmallPtrSet` has all the advantages of `SmallSet` (and a `SmallSet` of pointers is transparently implemented with a `SmallPtrSet`). If more than N insertions are performed, a single quadratically probed hash table is allocated and grows as needed, providing extremely efficient access (constant time insertion/deleting/queries with low constant factors) and is very stingy with malloc traffic.

Note that, unlike std::set, the iterators of `SmallPtrSet` are invalidated whenever an insertion occurs. Also, the values visited by the iterators are not visited in sorted order.

llvm/ADT/StringSet.h

`StringSet` is a thin wrapper around StringMap<char>, and it allows efficient storage and retrieval of unique strings.

Functionally analogous to `SmallSet<StringRef>`, `StringSet` also supports iteration. (The iterator dereferences to a `StringMapEntry<char>`, so you need to call `i->getKey()` to access the item of the StringSet.) On the other hand, `StringSet` doesn't support range-insertion and copy-construction, which SmallSet and SmallPtrSet do support.

llvm/ADT/DenseSet.h

DenseSet is a simple quadratically probed hash table. It excels at supporting small values: it uses a single allocation to hold all of the pairs that are currently inserted in the set. DenseSet is a great way to unique small values that are not simple pointers (use SmallPtrSet for pointers). Note that DenseSet has the same requirements for the value type that DenseMap has.

llvm/ADT/SparseSet.h

SparseSet holds a small number of objects identified by unsigned keys of moderate size. It uses a lot of memory, but provides operations that are almost as fast as a vector. Typical keys are physical registers, virtual registers, or numbered basic blocks.

SparseSet is useful for algorithms that need very fast clear/find/insert/erase and fast iteration over small sets. It is not intended for building composite data structures.

llvm/ADT/SparseMultiSet.h

SparseMultiSet adds multiset behavior to SparseSet, while retaining SparseSet's desirable attributes. Like SparseSet, it typically uses a lot of memory, but provides operations that are almost as fast as a vector. Typical keys are physical registers, virtual registers, or numbered basic blocks.

SparseMultiSet is useful for algorithms that need very fast clear/find/insert/erase of the entire collection, and iteration over sets of elements sharing a key. It is often a more efficient choice than using composite data structures (e.g. vector-of-vectors, map-of-vectors). It is not intended for building composite data structures.

llvm/ADT/FoldingSet.h

FoldingSet is an aggregate class that is really good at uniquing expensive-to-create or polymorphic objects. It is a combination of a chained hash table with intrusive links (uniqued objects are required to inherit from FoldingSetNode) that uses SmallVector as part of its ID process.

Consider a case where you want to implement a "getOrCreateFoo" method for a complex object (for example, a node in the code generator). The client has a description of what it wants to generate (it knows the opcode and all the operands), but we don't want to 'new' a node, then try inserting it into a set only to find out it already exists, at which point we would have to delete it and return the node that already exists.

To support this style of client, FoldingSet perform a query with a FoldingSetNodeID (which wraps SmallVector) that can be used to describe the element that we want to query for. The query either returns the element matching the ID or it returns an opaque ID that indicates where insertion should take place. Construction of the ID usually does not require heap traffic.

Because FoldingSet uses intrusive links, it can support polymorphic objects in the set (for example, you can have SDNode instances mixed with LoadSDNodes). Because the elements are individually allocated, pointers to the elements are stable: inserting or removing elements does not invalidate any pointers to other elements.

<set>

`std::set` is a reasonable all-around set class, which is decent at many things but great at nothing. std::set allocates memory for each element inserted (thus it is very malloc intensive) and typically stores three pointers per element in the set (thus adding a large amount of per-element space overhead). It offers guaranteed log(n) performance, which is not particularly fast from a complexity standpoint (particularly if the elements of the set are expensive to compare, like strings), and has extremely high constant factors for lookup, insertion and removal.

The advantages of std::set are that its iterators are stable (deleting or inserting an element from the set does not affect iterators or pointers to other elements) and that iteration over the set is guaranteed to be in sorted order. If the elements in the set are large, then the relative overhead of the pointers and malloc traffic is not a big deal, but if the elements of the set are small, std::set is almost never a good choice.

llvm/ADT/SetVector.h

LLVM's `SetVector<Type>` is an adapter class that combines your choice of a set-like container along with a Sequential Container The important property that this provides is efficient insertion with uniquing (duplicate elements are ignored) with iteration support. It implements this by inserting

elements into both a set-like container and the sequential container, using the set-like container for uniquing and the sequential container for iteration.

The difference between SetVector and other sets is that the order of iteration is guaranteed to match the order of insertion into the SetVector. This property is really important for things like sets of pointers. Because pointer values are non-deterministic (e.g. vary across runs of the program on different machines), iterating over the pointers in the set will not be in a well-defined order.

The drawback of SetVector is that it requires twice as much space as a normal set and has the sum of constant factors from the set-like container and the sequential container that it uses. Use it only if you need to iterate over the elements in a deterministic order. SetVector is also expensive to delete elements out of (linear time), unless you use its "pop_back" method, which is faster.

`SetVector` is an adapter class that defaults to using `std::vector` and a size 16 `SmallSet` for the underlying containers, so it is quite expensive. However, `"llvm/ADT/SetVector.h"` also provides a `SmallSetVector` class, which defaults to using a `SmallVector` and `SmallSet` of a specified size. If you use this, and if your sets are dynamically smaller than `N`, you will save a lot of heap traffic.

llvm/ADT/UniqueVector.h

UniqueVector is similar to SetVector but it retains a unique ID for each element inserted into the set. It internally contains a map and a vector, and it assigns a unique ID for each value inserted into the set.

UniqueVector is very expensive: its cost is the sum of the cost of maintaining both the map and vector, it has high complexity, high constant factors, and produces a lot of malloc traffic. It should be avoided.

llvm/ADT/ImmutableSet.h

ImmutableSet is an immutable (functional) set implementation based on an AVL tree. Adding or removing elements is done through a Factory object and results in the creation of a new ImmutableSet object. If an ImmutableSet already exists with the given contents, then the existing one is returned; equality is compared with a FoldingSetNodeID. The time and space complexity of add or remove operations is logarithmic in the size of the original set.

There is no method for returning an element of the set, you can only check for membership.

Other Set-Like Container Options

The STL provides several other options, such as std::multiset and the various "hash_set" like containers (whether from C++ TR1 or from the SGI library). We never use hash_set and unordered_set because they are generally very expensive (each insertion requires a malloc) and very non-portable.

std::multiset is useful if you're not interested in elimination of duplicates, but has all the drawbacks of std::set. A sorted vector (where you don't delete duplicate entries) or some other approach is almost always better.

Map-Like Containers (std::map, DenseMap, etc)

Map-like containers are useful when you want to associate data to a key. As usual, there are a lot of different ways to do this. :)

A sorted 'vector'

If your usage pattern follows a strict insert-then-query approach, you can trivially use the same approach as sorted vectors for set-like containers. The only difference is that your query function (which uses std::lower_bound to get efficient log(n) lookup) should only compare the key, not both the key and value. This yields the same advantages as sorted vectors for sets.

llvm/ADT/StringMap.h

Strings are commonly used as keys in maps, and they are difficult to support efficiently: they are variable length, inefficient to hash and compare when long, expensive to copy, etc. StringMap is a specialized container designed to cope with these issues. It supports mapping an arbitrary range of bytes to an arbitrary other object.

The StringMap implementation uses a quadratically-probed hash table, where the buckets store a pointer to the heap allocated entries (and some other stuff). The entries in the map must be heap allocated because the strings are variable length. The string data (key) and the element object (value) are stored in the same allocation with the string data immediately after the element object. This container guarantees the "`(char*)(&Value+1)`" points to the key string for a value.

The StringMap is very fast for several reasons: quadratic probing is very cache efficient for lookups, the hash value of strings in buckets is not recomputed when looking up an element, StringMap rarely has to touch the memory for unrelated objects when looking up a value (even when hash collisions happen), hash table growth does not recompute the hash values for strings already in the table, and each pair in the map is store in a single allocation (the string data is stored in the same allocation as the Value of a pair).

StringMap also provides query methods that take byte ranges, so it only ever copies a string if a value is inserted into the table.

StringMap iteration order, however, is not guaranteed to be deterministic, so any uses which require that should instead use a std::map.

llvm/ADT/IndexedMap.h

IndexedMap is a specialized container for mapping small dense integers (or values that can be mapped to small dense integers) to some other type. It is internally implemented as a vector with a mapping function that maps the keys to the dense integer range.

This is useful for cases like virtual registers in the LLVM code generator: they have a dense mapping that is offset by a compile-time constant (the first virtual register ID).

llvm/ADT/DenseMap.h

DenseMap is a simple quadratically probed hash table. It excels at supporting small keys and values: it uses a single allocation to hold all of the pairs that are currently inserted in the map. DenseMap is a great way to map pointers to pointers, or map other small types to each other.

There are several aspects of DenseMap that you should be aware of, however. The iterators in a DenseMap are invalidated whenever an insertion occurs, unlike map. Also, because DenseMap allocates space for a large number of key/value pairs (it starts with 64 by default), it will waste a lot of space if your keys or values are large. Finally, you must implement a partial specialization of DenseMapInfo for the key that you want, if it isn't already supported. This is required to tell DenseMap about two special marker values (which can never be inserted into the map) that it needs internally.

DenseMap's find_as() method supports lookup operations using an alternate key type. This is useful in cases where the normal key type is expensive to construct, but cheap to compare against. The DenseMapInfo is responsible for defining the appropriate comparison and hashing methods for each alternate key type used.

llvm/IR/ValueMap.h

ValueMap is a wrapper around a DenseMap mapping `Value*`s (or subclasses) to another type. When a Value is deleted or RAUW'ed, ValueMap will update itself so the new version of the key is mapped to the same value, just as if the key were a WeakVH. You can configure exactly how this happens, and what else happens on these two events, by passing a `Config` parameter to the ValueMap template.

llvm/ADT/IntervalMap.h

IntervalMap is a compact map for small keys and values. It maps key intervals instead of single keys, and it will automatically coalesce adjacent intervals. When the map only contains a few intervals, they are stored in the map object itself to avoid allocations.

The IntervalMap iterators are quite big, so they should not be passed around as STL iterators. The heavyweight iterators allow a smaller data structure.

<map>

std::map has similar characteristics to std::set: it uses a single allocation per pair inserted into the map, it offers log(n) lookup with an extremely large constant factor, imposes a space penalty of 3 pointers per pair in the map, etc.

std::map is most useful when your keys or values are very large, if you need to iterate over the collection in sorted order, or if you need stable iterators into the map (i.e. they don't get invalidated if an insertion or deletion of another element takes place).

llvm/ADT/MapVector.h

`MapVector<KeyT,ValueT>` provides a subset of the DenseMap interface. The main difference is that the iteration order is guaranteed to be the insertion order, making it an easy (but somewhat expensive) solution for non-deterministic iteration over maps of pointers.

It is implemented by mapping from key to an index in a vector of key,value pairs. This provides fast lookup and iteration, but has two main drawbacks: the key is stored twice and removing elements takes linear time. If it is necessary to remove elements, it's best to remove them in bulk using `remove_if()`.

llvm/ADT/IntEqClasses.h

IntEqClasses provides a compact representation of equivalence classes of small integers. Initially, each integer in the range 0..n-1 has its own equivalence class. Classes can be joined by passing two class representatives to the join(a, b) method. Two integers are in the same class when findLeader() returns the same representative.

Once all equivalence classes are formed, the map can be compressed so each integer 0..n-1 maps to an equivalence class number in the range 0..m-1, where m is the total number of equivalence classes. The map must be uncompressed before it can be edited again.

llvm/ADT/ImmutableMap.h

ImmutableMap is an immutable (functional) map implementation based on an AVL tree. Adding or removing elements is done through a Factory object and results in the creation of a new ImmutableMap object. If an ImmutableMap already exists with the given key set, then the existing one is returned; equality is compared with a FoldingSetNodeID. The time and space complexity of add or remove operations is logarithmic in the size of the original map.

Other Map-Like Container Options

The STL provides several other options, such as std::multimap and the various "hash_map" like containers (whether from C++ TR1 or from the SGI library). We never use hash_set and unordered_set because they are generally very expensive (each insertion requires a malloc) and very non-portable.

std::multimap is useful if you want to map a key to multiple values, but has all the drawbacks of std::map. A sorted vector or some other approach is almost always better.

Bit storage containers (BitVector, SparseBitVector, CoalescingBitVector)

There are three bit storage containers, and choosing when to use each is relatively straightforward.

One additional option is `std::vector<bool>`: we discourage its use for two reasons 1) the implementation in many common compilers (e.g. commonly available versions of GCC) is extremely inefficient and 2) the C++ standards committee is likely to deprecate this container and/or change it significantly somehow. In any case, please don't use it.

BitVector

The BitVector container provides a dynamic size set of bits for manipulation. It supports individual bit setting/testing, as well as set operations. The set operations take time O(size of bitvector), but operations are performed one word at a time, instead of one bit at a time. This makes the BitVector very fast for set operations compared to other containers. Use the BitVector when you expect the number of set bits to be high (i.e. a dense set).

SmallBitVector

The SmallBitVector container provides the same interface as BitVector, but it is optimized for the case where only a small number of bits, less than 25 or so, are needed. It also transparently supports larger bit counts, but slightly less efficiently than a plain BitVector, so SmallBitVector should only be used when larger counts are rare.

At this time, SmallBitVector does not support set operations (and, or, xor), and its operator[] does not provide an assignable lvalue.

SparseBitVector

The SparseBitVector container is much like BitVector, with one major difference: Only the bits that are set, are stored. This makes the SparseBitVector much more space efficient than BitVector when the set is sparse, as well as making set operations O(number of set bits) instead of O(size of universe). The downside to the SparseBitVector is that setting and testing of random bits is O(N), and on large SparseBitVectors, this can be slower than BitVector. In our implementation, setting or testing bits in sorted order (either forwards or reverse) is O(1) worst case. Testing and setting bits within 128 bits (depends on size) of the current bit is also O(1). As a general statement, testing/setting bits in a SparseBitVector is O(distance away from last set bit).

CoalescingBitVector

The CoalescingBitVector container is similar in principle to a SparseBitVector, but is optimized to represent large contiguous ranges of set bits compactly. It does this by coalescing contiguous ranges of set bits into intervals. Searching for a bit in a CoalescingBitVector is O(log(gaps between contiguous ranges)).

CoalescingBitVector is a better choice than BitVector when gaps between ranges of set bits are large. It's a better choice than SparseBitVector when find() operations must have fast, predictable performance. However, it's not a good choice for representing sets which have lots of very short ranges. E.g. the set *{2*x : x in [0, n)}* would be a pathological input.

Debugging

A handful of GDB pretty printers are provided for some of the core LLVM libraries. To use them, execute the following (or add it to your `~/.gdbinit`):

```
source /path/to/llvm/src/utils/gdb-scripts/prettyprinters.py
```

It also might be handy to enable the print pretty option to avoid data structures being printed as a big block of text.

Helpful Hints for Common Operations

This section describes how to perform some very simple transformations of LLVM code. This is meant to give examples of common idioms used, showing the practical side of LLVM transformations.

Because this is a "how-to" section, you should also read about the main classes that you will be working with. The Core LLVM Class Hierarchy Reference contains details and descriptions of the main classes that you should know about.

Basic Inspection and Traversal Routines

The LLVM compiler infrastructure have many different data structures that may be traversed. Following the example of the C++ standard template library, the techniques used to traverse these various data structures are all basically the same. For an enumerable sequence of values, the `XXXbegin()` function (or method) returns an iterator to the start of the sequence, the `XXXend()` function returns an iterator pointing to one past the last valid element of the sequence, and there is some `XXXiterator` data type that is common between the two operations.

Because the pattern for iteration is common across many different aspects of the program representation, the standard template library algorithms may be used on them, and it is easier to remember how to iterate. First we show a few common examples of the data structures that need to be traversed. Other data structures are traversed in very similar ways.

Iterating over the **BasicBlock** in a **Function**

It's quite common to have a `Function` instance that you'd like to transform in some way; in particular, you'd like to manipulate its `BasicBlock`s. To facilitate this, you'll need to iterate over all of the `BasicBlock`s that constitute the `Function`. The following is an example that prints the name of a `BasicBlock` and the number of `Instruction`s it contains:

```
Function &Func = ...
for (BasicBlock &BB : Func)
  // Print out the name of the basic block if it has one, and then the
  // number of instructions that it contains
  errs() << "Basic block (name=" << BB.getName() << ") has "
             << BB.size() << " instructions.\n";
```

Iterating over the **Instruction** in a **BasicBlock**

Just like when dealing with `BasicBlock`s in `Function`s, it's easy to iterate over the individual instructions that make up `BasicBlock`s. Here's a code snippet that prints out each instruction in a `BasicBlock`:

```
BasicBlock& BB = ...
for (Instruction &I : BB)
   // The next statement works since operator<<(ostream&,...)
   // is overloaded for Instruction&
   errs() << I << "\n";
```

However, this isn't really the best way to print out the contents of a `BasicBlock`! Since the ostream operators are overloaded for virtually anything you'll care about, you could have just invoked the print routine on the basic block itself: `errs() << BB << "\n";`.

Iterating over the **Instruction** in a **Function**

If you're finding that you commonly iterate over a `Function`'s `BasicBlock`s and then that `BasicBlock`'s `Instruction`s, `InstIterator` should be used instead. You'll need to include `llvm/IR/InstIterator.h` (doxygen) and then instantiate `InstIterator`s explicitly in your code. Here's a small example that shows how to dump all instructions in a function to the standard error stream:

```
#include "llvm/IR/InstIterator.h"

// F is a pointer to a Function instance
for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
  errs() << *I << "\n";
```

Easy, isn't it? You can also use `InstIterator`s to fill a work list with its initial contents. For example, if you wanted to initialize a work list to contain all instructions in a `Function` F, all you would need to do is something like:

```
std::set<Instruction*> worklist;
// or better yet, SmallPtrSet<Instruction*, 64> worklist;

for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
  worklist.insert(&*I);
```

The STL set `worklist` would now contain all instructions in the `Function` pointed to by F.

Turning an iterator into a class pointer (and vice-versa)

Sometimes, it'll be useful to grab a reference (or pointer) to a class instance when all you've got at hand is an iterator. Well, extracting a reference or a pointer from an iterator is very straight-forward. Assuming that `i` is a `BasicBlock::iterator` and `j` is a `BasicBlock::const_iterator`:

```
Instruction& inst = *i;   // Grab reference to instruction reference
Instruction* pinst = &*i; // Grab pointer to instruction reference
const Instruction& inst = *j;
```

However, the iterators you'll be working with in the LLVM framework are special: they will automatically convert to a ptr-to-instance type whenever they need to. Instead of dereferencing the iterator and then taking the address of the result, you can simply assign the iterator to the proper pointer type and you get the dereference and address-of operation as a result of the assignment (behind the scenes, this is a result of overloading casting mechanisms). Thus the second line of the last example,

```
Instruction *pinst = &*i;
```

is semantically equivalent to

```
Instruction *pinst = i;
```

It's also possible to turn a class pointer into the corresponding iterator, and this is a constant time operation (very efficient). The following code snippet illustrates use of the conversion constructors provided by LLVM iterators. By using these, you can explicitly grab the iterator of something without actually obtaining it via iteration over some structure:

```
void printNextInstruction(Instruction* inst) {
  BasicBlock::iterator it(inst);
  ++it; // After this line, it refers to the instruction after *inst
  if (it != inst->getParent()->end()) errs() << *it << "\n";
}
```

Unfortunately, these implicit conversions come at a cost; they prevent these iterators from conforming to standard iterator conventions, and thus from being usable with standard algorithms and containers. For example, they prevent the following code, where `B` is a `BasicBlock`, from compiling:

```
llvm::SmallVector<llvm::Instruction *, 16>(B->begin(), B->end());
```

Because of this, these implicit conversions may be removed some day, and `operator*` changed to return a pointer instead of a reference.

Finding call sites: a slightly more complex example

Say that you're writing a FunctionPass and would like to count all the locations in the entire module (that is, across every `Function`) where a certain function (i.e., some `Function *`) is already in scope. As you'll learn later, you may want to use an `InstVisitor` to accomplish this in a much more straight-

forward manner, but this example will allow us to explore how you'd do it if you didn't have `InstVisitor` around. In pseudo-code, this is what we want to do:

```
initialize callCounter to zero
for each Function f in the Module
  for each BasicBlock b in f
    for each Instruction i in b
      if (i a Call and calls the given function)
        increment callCounter
```

And the actual code is (remember, because we're writing a `FunctionPass`, our `FunctionPass`-derived class simply has to override the `runOnFunction` method):

```
Function* targetFunc = ...;

class OurFunctionPass : public FunctionPass {
  public:
    OurFunctionPass(): callCounter(0) { }

    virtual runOnFunction(Function& F) {
      for (BasicBlock &B : F) {
        for (Instruction &I: B) {
          if (auto *CB = dyn_cast<CallBase>(&I)) {
            // We know we've encountered some kind of call instruction (call,
            // invoke, or callbr), so we need to determine if it's a call to
            // the function pointed to by m_func or not.
            if (CB->getCalledFunction() == targetFunc)
              ++callCounter;
          }
        }
      }
    }

  private:
    unsigned callCounter;
};
```

Iterating over def-use & use-def chains

Frequently, we might have an instance of the `Value` class (doxygen) and we want to determine which `Users` use the `Value`. The list of all `Users` of a particular `Value` is called a def-use chain. For example, let's say we have a `Function*` named `F` to a particular function `foo`. Finding all of the instructions that use `foo` is as simple as iterating over the def-use chain of `F`:

```
Function *F = ...;

for (User *U : F->users()) {
  if (Instruction *Inst = dyn_cast<Instruction>(U)) {
    errs() << "F is used in instruction:\n";
    errs() << *Inst << "\n";
  }
```

Alternatively, it's common to have an instance of the `User` Class (doxygen) and need to know what `Values` are used by it. The list of all `Values` used by a `User` is known as a use-def chain. Instances of class `Instruction` are common `User` s, so we might want to iterate over all of the values that a particular instruction uses (that is, the operands of the particular `Instruction`):

```
Instruction *pi = ...;

for (Use &U : pi->operands()) {
  Value *v = U.get();
  // ...
}
```

Declaring objects as `const` is an important tool of enforcing mutation free algorithms (such as analyses, etc.). For this purpose above iterators come in constant flavors as `Value::const_use_iterator`

and `Value::const_op_iterator`. They automatically arise when calling `use/op_begin()` on `const Value*`s or `const User*`s respectively. Upon dereferencing, they return `const Use*`s. Otherwise the above patterns remain unchanged.

Iterating over predecessors & successors of blocks

Iterating over the predecessors and successors of a block is quite easy with the routines defined in `"llvm/IR/CFG.h"`. Just use code like this to iterate over all predecessors of BB:

```
#include "llvm/IR/CFG.h"
BasicBlock *BB = ...;

for (BasicBlock *Pred : predecessors(BB)) {
  // ...
}
```

Similarly, to iterate over successors use `successors`.

Making simple changes

There are some primitive transformation operations present in the LLVM infrastructure that are worth knowing about. When performing transformations, it's fairly common to manipulate the contents of basic blocks. This section describes some of the common methods for doing so and gives example code.

Creating and inserting new `Instruction`s

Instantiating Instructions

Creation of `Instructions` is straight-forward: simply call the constructor for the kind of instruction to instantiate and provide the necessary parameters. For example, an `AllocaInst` only requires a (const-ptr-to) `Type`. Thus:

```
auto *ai = new AllocaInst(Type::Int32Ty);
```

will create an `AllocaInst` instance that represents the allocation of one integer in the current stack frame, at run time. Each `Instruction` subclass is likely to have varying default parameters which change the semantics of the instruction, so refer to the doxygen documentation for the subclass of Instruction that you're interested in instantiating.

Naming values

It is very useful to name the values of instructions when you're able to, as this facilitates the debugging of your transformations. If you end up looking at generated LLVM machine code, you definitely want to have logical names associated with the results of instructions! By supplying a value for the `Name` (default) parameter of the `Instruction` constructor, you associate a logical name with the result of the instruction's execution at run time. For example, say that I'm writing a transformation that dynamically allocates space for an integer on the stack, and that integer is going to be used as some kind of index by some other code. To accomplish this, I place an `AllocaInst` at the first point in the first `BasicBlock` of some `Function`, and I'm intending to use it within the same `Function`. I might do:

```
auto *pa = new AllocaInst(Type::Int32Ty, 0, "indexLoc");
```

where `indexLoc` is now the logical name of the instruction's execution value, which is a pointer to an integer on the run time stack.

Inserting instructions

There are essentially three ways to insert an `Instruction` into an existing sequence of instructions that form a `BasicBlock`:

- Insertion into an explicit instruction list

  Given a `BasicBlock* pb`, an `Instruction* pi` within that `BasicBlock`, and a newly-created instruction we wish to insert before `*pi`, we do the following:

  ```
  BasicBlock *pb = ...;
  Instruction *pi = ...;
  auto *newInst = new Instruction(...);

  pb->getInstList().insert(pi, newInst); // Inserts newInst before pi in pb
  ```

  Appending to the end of a `BasicBlock` is so common that the `Instruction` class and `Instruction`-derived classes provide constructors which take a pointer to a `BasicBlock` to be appended to. For example code that looked like:

  ```
  BasicBlock *pb = ...;
  auto *newInst = new Instruction(...);

  pb->getInstList().push_back(newInst); // Appends newInst to pb
  ```

  becomes:

  ```
  BasicBlock *pb = ...;
  auto *newInst = new Instruction(..., pb);
  ```

  which is much cleaner, especially if you are creating long instruction streams.

- Insertion into an implicit instruction list

  `Instruction` instances that are already in `BasicBlock`s are implicitly associated with an existing instruction list: the instruction list of the enclosing basic block. Thus, we could have accomplished the same thing as the above code without being given a `BasicBlock` by doing:

  ```
  Instruction *pi = ...;
  auto *newInst = new Instruction(...);

  pi->getParent()->getInstList().insert(pi, newInst);
  ```

  In fact, this sequence of steps occurs so frequently that the `Instruction` class and `Instruction`-derived classes provide constructors which take (as a default parameter) a pointer to an `Instruction` which the newly-created `Instruction` should precede. That is, `Instruction` constructors are capable of inserting the newly-created instance into the `BasicBlock` of a provided instruction, immediately before that instruction. Using an `Instruction` constructor with a `insertBefore` (default) parameter, the above code becomes:

  ```
  Instruction* pi = ...;
  auto *newInst = new Instruction(..., pi);
  ```

  which is much cleaner, especially if you're creating a lot of instructions and adding them to `BasicBlock`s.

- Insertion using an instance of `IRBuilder`

  Inserting several `Instruction`s can be quite laborious using the previous methods. The `IRBuilder` is a convenience class that can be used to add several instructions to the end of a `BasicBlock` or before a particular `Instruction`. It also supports constant folding and renaming named registers (see `IRBuilder`'s template arguments).

  The example below demonstrates a very simple use of the `IRBuilder` where three instructions are inserted before the instruction `pi`. The first two instructions are Call instructions and third instruction multiplies the return value of the two calls.

  ```
  Instruction *pi = ...;
  IRBuilder<> Builder(pi);
  CallInst* callOne = Builder.CreateCall(...);
  ```

```
CallInst* callTwo = Builder.CreateCall(...);
Value* result = Builder.CreateMul(callOne, callTwo);
```

The example below is similar to the above example except that the created `IRBuilder` inserts instructions at the end of the `BasicBlock` `pb`.

```
BasicBlock *pb = ...;
IRBuilder<> Builder(pb);
CallInst* callOne = Builder.CreateCall(...);
CallInst* callTwo = Builder.CreateCall(...);
Value* result = Builder.CreateMul(callOne, callTwo);
```

See Kaleidoscope Tutorial for a practical use of the `IRBuilder`.

Deleting Instructions

Deleting an instruction from an existing sequence of instructions that form a BasicBlock is very straight-forward: just call the instruction's `eraseFromParent()` method. For example:

```
Instruction *I = .. ;
I->eraseFromParent();
```

This unlinks the instruction from its containing basic block and deletes it. If you'd just like to unlink the instruction from its containing basic block but not delete it, you can use the `removeFromParent()` method.

Replacing an Instruction with another Value

Replacing individual instructions

Including "llvm/Transforms/Utils/BasicBlockUtils.h" permits use of two very useful replace functions: `ReplaceInstWithValue` and `ReplaceInstWithInst`.

Deleting Instructions

- `ReplaceInstWithValue`

  This function replaces all uses of a given instruction with a value, and then removes the original instruction. The following example illustrates the replacement of the result of a particular `AllocaInst` that allocates memory for a single integer with a null pointer to an integer.

  ```
  AllocaInst* instToReplace = ...;
  BasicBlock::iterator ii(instToReplace);

  ReplaceInstWithValue(instToReplace->getParent()->getInstList(), ii,
                       Constant::getNullValue(PointerType::getUnqual(Type::Int32Ty)));
  ```

- `ReplaceInstWithInst`

  This function replaces a particular instruction with another instruction, inserting the new instruction into the basic block at the location where the old instruction was, and replacing any uses of the old instruction with the new instruction. The following example illustrates the replacement of one `AllocaInst` with another.

  ```
  AllocaInst* instToReplace = ...;
  BasicBlock::iterator ii(instToReplace);

  ReplaceInstWithInst(instToReplace->getParent()->getInstList(), ii,
                      new AllocaInst(Type::Int32Ty, 0, "ptrToReplacedInt"));
  ```

Replacing multiple uses of Users and Values

You can use `Value::replaceAllUsesWith` and `User::replaceUsesOfWith` to change more than one use at a time. See the doxygen documentation for the Value Class and User Class, respectively, for more information.

Deleting GlobalVariables

Deleting a global variable from a module is just as easy as deleting an Instruction. First, you must have a pointer to the global variable that you wish to delete. You use this pointer to erase it from its parent, the module. For example:

```
GlobalVariable *GV = .. ;

GV->eraseFromParent();
```

Threads and LLVM

This section describes the interaction of the LLVM APIs with multithreading, both on the part of client applications, and in the JIT, in the hosted application.

Note that LLVM's support for multithreading is still relatively young. Up through version 2.5, the execution of threaded hosted applications was supported, but not threaded client access to the APIs. While this use case is now supported, clients must adhere to the guidelines specified below to ensure proper operation in multithreaded mode.

Note that, on Unix-like platforms, LLVM requires the presence of GCC's atomic intrinsics in order to support threaded operation. If you need a multithreading-capable LLVM on a platform without a suitably modern system compiler, consider compiling LLVM and LLVM-GCC in single-threaded mode, and using the resultant compiler to build a copy of LLVM with multithreading support.

Ending Execution with `llvm_shutdown()`

When you are done using the LLVM APIs, you should call `llvm_shutdown()` to deallocate memory used for internal structures.

Lazy Initialization with `ManagedStatic`

`ManagedStatic` is a utility class in LLVM used to implement static initialization of static resources, such as the global type tables. In a single-threaded environment, it implements a simple lazy initialization scheme. When LLVM is compiled with support for multi-threading, however, it uses double-checked locking to implement thread-safe lazy initialization.

Achieving Isolation with `LLVMContext`

`LLVMContext` is an opaque class in the LLVM API which clients can use to operate multiple, isolated instances of LLVM concurrently within the same address space. For instance, in a hypothetical compile-server, the compilation of an individual translation unit is conceptually independent from all the others, and it would be desirable to be able to compile incoming translation units concurrently on independent server threads. Fortunately, `LLVMContext` exists to enable just this kind of scenario!

Conceptually, `LLVMContext` provides isolation. Every LLVM entity (`Modules`, `Values`, `Types`, `Constants`, etc.) in LLVM's in-memory IR belongs to an `LLVMContext`. Entities in different contexts cannot interact with each other: `Modules` in different contexts cannot be linked together, `Functions` cannot be added to `Modules` in different contexts, etc. What this means is that is safe to compile on multiple threads simultaneously, as long as no two threads operate on entities within the same context.

In practice, very few places in the API require the explicit specification of a `LLVMContext`, other than the `Type` creation/lookup APIs. Because every `Type` carries a reference to its owning context, most other entities can determine what context they belong to by looking at their own `Type`. If you are adding new entities to LLVM IR, please try to maintain this interface design.

Threads and the JIT

LLVM's "eager" JIT compiler is safe to use in threaded programs. Multiple threads can call `ExecutionEngine::getPointerToFunction()` or `ExecutionEngine::runFunction()` concurrently, and multiple threads can run code output by the JIT concurrently. The user must still ensure that only one thread accesses IR in a given `LLVMContext` while another thread might be modifying it. One way to do that is to always hold the JIT lock while accessing IR outside the JIT (the JIT modifies the IR by adding `CallbackVH`s). Another way is to only call `getPointerToFunction()` from the `LLVMContext`'s thread.

When the JIT is configured to compile lazily (using `ExecutionEngine::DisableLazyCompilation(false)`), there is currently a race condition in updating call sites after a function is lazily-jitted. It's still possible to use the lazy JIT in a threaded program if you ensure that only one thread at a time can call any particular lazy stub and that the JIT lock guards any IR access, but we suggest using only the eager JIT in threaded programs.

Advanced Topics

This section describes some of the advanced or obscure API's that most clients do not need to be aware of. These API's tend manage the inner workings of the LLVM system, and only need to be accessed in unusual circumstances.

The `ValueSymbolTable` class

The `ValueSymbolTable` (doxygen) class provides a symbol table that the Function and Module classes use for naming value definitions. The symbol table can provide a name for any Value.

Note that the `SymbolTable` class should not be directly accessed by most clients. It should only be used when iteration over the symbol table names themselves are required, which is very special purpose. Note that not all LLVM Values have names, and those without names (i.e. they have an empty name) do not exist in the symbol table.

Symbol tables support iteration over the values in the symbol table with `begin/end/iterator` and supports querying to see if a specific name is in the symbol table (with `lookup`). The `ValueSymbolTable` class exposes no public mutator methods, instead, simply call `setName` on a value, which will autoinsert it into the appropriate symbol table.

The `User` and owned `Use` classes' memory layout

The `User` (doxygen) class provides a basis for expressing the ownership of `User` towards other Value instances. The `Use` (doxygen) helper class is employed to do the bookkeeping and to facilitate O(1) addition and removal.

Interaction and relationship between `User` and `Use` objects

A subclass of `User` can choose between incorporating its `Use` objects or refer to them out-of-line by means of a pointer. A mixed variant (some `Use` s inline others hung off) is impractical and breaks the invariant that the `Use` objects belonging to the same `User` form a contiguous array.

We have 2 different layouts in the `User` (sub)classes:

- Layout a)

  The `Use` object(s) are inside (resp. at fixed offset) of the `User` object and there are a fixed number of them.

- Layout b)

  The `Use` object(s) are referenced by a pointer to an array from the `User` object and there may be a variable number of them.

As of v2.4 each layout still possesses a direct pointer to the start of the array of `Use`s. Though not mandatory for layout a), we stick to this redundancy for the sake of simplicity. The `User` object also

stores the number of `Use` objects it has. (Theoretically this information can also be calculated given the scheme presented below.)

Special forms of allocation operators (`operator new`) enforce the following memory layouts:

- Layout a) is modelled by prepending the `User` object by the `Use[]` array.

```
...---.---.---.---.-------...
  | P | P | P | P | User
'''---'---'---'---'-------'''
```

- Layout b) is modelled by pointing at the `Use[]` array.

```
.-------...
| User
'-------'''
    |
    v
    .---.---.---.---...
    | P | P | P | P |
    '---'---'---'---'''
```

(In the above figures 'P' stands for the `Use**` that is stored in each `Use` object in the member `Use::Prev`)

Designing Type Hierarchies and Polymorphic Interfaces

There are two different design patterns that tend to result in the use of virtual dispatch for methods in a type hierarchy in C++ programs. The first is a genuine type hierarchy where different types in the hierarchy model a specific subset of the functionality and semantics, and these types nest strictly within each other. Good examples of this can be seen in the `Value` or `Type` type hierarchies.

A second is the desire to dispatch dynamically across a collection of polymorphic interface implementations. This latter use case can be modeled with virtual dispatch and inheritance by defining an abstract interface base class which all implementations derive from and override. However, this implementation strategy forces an "is-a" relationship to exist that is not actually meaningful. There is often not some nested hierarchy of useful generalizations which code might interact with and move up and down. Instead, there is a singular interface which is dispatched across a range of implementations.

The preferred implementation strategy for the second use case is that of generic programming (sometimes called "compile-time duck typing" or "static polymorphism"). For example, a template over some type parameter `T` can be instantiated across any particular implementation that conforms to the interface or concept. A good example here is the highly generic properties of any type which models a node in a directed graph. LLVM models these primarily through templates and generic programming. Such templates include the `LoopInfoBase` and `DominatorTreeBase`. When this type of polymorphism truly needs dynamic dispatch you can generalize it using a technique called concept-based polymorphism. This pattern emulates the interfaces and behaviors of templates using a very limited form of virtual dispatch for type erasure inside its implementation. You can find examples of this technique in the `PassManager.h` system, and there is a more detailed introduction to it by Sean Parent in several of his talks and papers:

1. Inheritance Is The Base Class of Evil - The GoingNative 2013 talk describing this technique, and probably the best place to start.
2. Value Semantics and Concepts-based Polymorphism - The C++Now! 2012 talk describing this technique in more detail.
3. Sean Parent's Papers and Presentations - A GitHub project full of links to slides, video, and sometimes code.

When deciding between creating a type hierarchy (with either tagged or virtual dispatch) and using templates or concepts-based polymorphism, consider whether there is some refinement of an abstract base class which is a semantically meaningful type on an interface boundary. If anything more

refined than the root abstract interface is meaningless to talk about as a partial extension of the semantic model, then your use case likely fits better with polymorphism and you should avoid using virtual dispatch. However, there may be some exigent circumstances that require one technique or the other to be used.

If you do need to introduce a type hierarchy, we prefer to use explicitly closed type hierarchies with manual tagged dispatch and/or RTTI rather than the open inheritance model and virtual dispatch that is more common in C++ code. This is because LLVM rarely encourages library consumers to extend its core types, and leverages the closed and tag-dispatched nature of its hierarchies to generate significantly more efficient code. We have also found that a large amount of our usage of type hierarchies fits better with tag-based pattern matching rather than dynamic dispatch across a common interface. Within LLVM we have built custom helpers to facilitate this design. See this document's section on isa and dyn_cast and our detailed document which describes how you can implement this pattern for use with the LLVM helpers.

ABI Breaking Checks

Checks and asserts that alter the LLVM C++ ABI are predicated on the preprocessor symbol *LLVM_ENABLE_ABI_BREAKING_CHECKS* - LLVM libraries built with *LLVM_ENABLE_ABI_BREAKING_CHECKS* are not ABI compatible LLVM libraries built without it defined. By default, turning on assertions also turns on *LLVM_ENABLE_ABI_BREAKING_CHECKS* so a default +Asserts build is not ABI compatible with a default -Asserts build. Clients that want ABI compatibility between +Asserts and -Asserts builds should use the CMake build system to set *LLVM_ENABLE_ABI_BREAKING_CHECKS* independently of *LLVM_ENABLE_ASSERTIONS*.

The Core LLVM Class Hierarchy Reference

`#include "llvm/IR/Type.h"`

header source: Type.h

doxygen info: Type Classes

The Core LLVM classes are the primary means of representing the program being inspected or transformed. The core LLVM classes are defined in header files in the `include/llvm/IR` directory, and implemented in the `lib/IR` directory. It's worth noting that, for historical reasons, this library is called `libLLVMCore.so`, not `libLLVMIR.so` as you might expect.

The Type class and Derived Types

`Type` is a superclass of all type classes. Every `Value` has a `Type`. `Type` cannot be instantiated directly but only through its subclasses. Certain primitive types (`VoidType`, `LabelType`, `FloatType` and `DoubleType`) have hidden subclasses. They are hidden because they offer no useful functionality beyond what the `Type` class offers except to distinguish themselves from other subclasses of `Type`.

All other types are subclasses of `DerivedType`. Types can be named, but this is not a requirement. There exists exactly one instance of a given shape at any one time. This allows type equality to be performed with address equality of the Type Instance. That is, given two `Type*` values, the types are identical if the pointers are identical.

Important Public Methods

- `bool isIntegerTy() const`: Returns true for any integer type.
- `bool isFloatingPointTy()`: Return true if this is one of the five floating point types.
- `bool isSized()`: Return true if the type has known size. Things that don't have a size are abstract types, labels and void.

Important Derived Types

`IntegerType`

Subclass of DerivedType that represents integer types of any bit width. Any bit width between `IntegerType::MIN_INT_BITS` (1) and `IntegerType::MAX_INT_BITS` (~8 million) can be represented.

- `static const IntegerType* get(unsigned NumBits)`: get an integer type of a specific bit width.
- `unsigned getBitWidth() const`: Get the bit width of an integer type.

`SequentialType`

This is subclassed by ArrayType and VectorType.

- `const Type * getElementType() const`: Returns the type of each of the elements in the sequential type.
- `uint64_t getNumElements() const`: Returns the number of elements in the sequential type.

`ArrayType`

This is a subclass of SequentialType and defines the interface for array types.

`PointerType`

Subclass of Type for pointer types.

`VectorType`

Subclass of SequentialType for vector types. A vector type is similar to an ArrayType but is distinguished because it is a first class type whereas ArrayType is not. Vector types are used for vector operations and are usually small vectors of an integer or floating point type.

`StructType`

Subclass of DerivedTypes for struct types.

`FunctionType`

Subclass of DerivedTypes for function types.

- `bool isVarArg() const`: Returns true if it's a vararg function.
- `const Type * getReturnType() const`: Returns the return type of the function.
- `const Type * getParamType (unsigned i)`: Returns the type of the ith parameter.
- `const unsigned getNumParams() const`: Returns the number of formal parameters.

The `Module` class

`#include "llvm/IR/Module.h"`

header source: Module.h

doxygen info: Module Class

The `Module` class represents the top level structure present in LLVM programs. An LLVM module is effectively either a translation unit of the original program or a combination of several translation units merged by the linker. The `Module` class keeps track of a list of Functions, a list of GlobalVariables, and a SymbolTable. Additionally, it contains a few helpful member functions that try to make common operations easy.

Important Public Members of the `Module` class

- `Module::Module(std::string name = "")`

  Constructing a Module is easy. You can optionally provide a name for it (probably based on the name of the translation unit).

- `Module::iterator` - Typedef for function list iterator
  `Module::const_iterator` - Typedef for const_iterator.
  `begin()`, `end()`, `size()`, `empty()`

  These are forwarding methods that make it easy to access the contents of a `Module` object's Function list.

- `Module::FunctionListType &getFunctionList()`

  Returns the list of Functions. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `Module::global_iterator` - Typedef for global variable list iterator
  `Module::const_global_iterator` - Typedef for const_iterator.
  `global_begin()`, `global_end()`, `global_size()`, `global_empty()`

  These are forwarding methods that make it easy to access the contents of a `Module` object's GlobalVariable list.

- `Module::GlobalListType &getGlobalList()`

  Returns the list of GlobalVariables. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `SymbolTable *getSymbolTable()`

  Return a reference to the SymbolTable for this `Module`.

- `Function *getFunction(StringRef Name) const`

  Look up the specified function in the `Module` SymbolTable. If it does not exist, return `null`.

- `FunctionCallee getOrInsertFunction(const std::string &Name, const FunctionType *T)`

  Look up the specified function in the `Module` SymbolTable. If it does not exist, add an external declaration for the function and return it. Note that the function signature already present may not match the requested signature. Thus, in order to enable the common usage of passing the result directly to EmitCall, the return type is a struct of `{FunctionType *T, Constant *FunctionPtr}`, rather than simply the `Function*` with potentially an unexpected signature.

- `std::string getTypeName(const Type *Ty)`

  If there is at least one entry in the SymbolTable for the specified Type, return it. Otherwise return the empty string.

- `bool addTypeName(const std::string &Name, const Type *Ty)`

  Insert an entry in the SymbolTable mapping `Name` to `Ty`. If there is already an entry for this name, true is returned and the SymbolTable is not modified.

The `Value` class

`#include "llvm/IR/Value.h"`

header source: Value.h

doxygen info: Value Class

The `Value` class is the most important class in the LLVM Source base. It represents a typed value that may be used (among other things) as an operand to an instruction. There are many different types of `Values`, such as Constants, Arguments. Even Instructions and Functions are `Values`.

A particular `Value` may be used many times in the LLVM representation for a program. For example, an incoming argument to a function (represented with an instance of the Argument class) is "used" by every instruction in the function that references the argument. To keep track of this relationship, the `Value` class keeps a list of all of the `Users` that is using it (the User class is a base class for all nodes in

the LLVM graph that can refer to `Values`). This use list is how LLVM represents def-use information in the program, and is accessible through the `use_*` methods, shown below.

Because LLVM is a typed representation, every LLVM `Value` is typed, and this Type is available through the `getType()` method. In addition, all LLVM values can be named. The "name" of the `Value` is a symbolic string printed in the LLVM code:

```
%foo = add i32 1, 2
```

The name of this instruction is "foo". NOTE that the name of any value may be missing (an empty string), so names should ONLY be used for debugging (making the source code easier to read, debugging printouts), they should not be used to keep track of values or map between them. For this purpose, use a `std::map` of pointers to the `Value` itself instead.

One important aspect of LLVM is that there is no distinction between an SSA variable and the operation that produces it. Because of this, any reference to the value produced by an instruction (or the value available as an incoming argument, for example) is represented as a direct pointer to the instance of the class that represents this value. Although this may take some getting used to, it simplifies the representation and makes it easier to manipulate.

Important Public Members of the `Value` class

- `Value::use_iterator` - Typedef for iterator over the use-list
  `Value::const_use_iterator` - Typedef for const_iterator over the use-list
  `unsigned use_size()` - Returns the number of users of the value.
  `bool use_empty()` - Returns true if there are no users.
  `use_iterator use_begin()` - Get an iterator to the start of the use-list.
  `use_iterator use_end()` - Get an iterator to the end of the use-list.
  `User *use_back()` - Returns the last element in the list.

  These methods are the interface to access the def-use information in LLVM. As with all other iterators in LLVM, the naming conventions follow the conventions defined by the STL.
- `Type *getType() const` This method returns the Type of the Value.
- `bool hasName() const`
  `std::string getName() const`
  `void setName(const std::string &Name)`

  This family of methods is used to access and assign a name to a `Value`, be aware of the precaution above.
- `void replaceAllUsesWith(Value *V)`

  This method traverses the use list of a `Value` changing all Users of the current value to refer to "V" instead. For example, if you detect that an instruction always produces a constant value (for example through constant folding), you can replace all uses of the instruction with the constant like this:

  ```
  Inst->replaceAllUsesWith(ConstVal);
  ```

The `User` class

`#include "llvm/IR/User.h"`

header source: User.h

doxygen info: User Class

Superclass: Value

The `User` class is the common base class of all LLVM nodes that may refer to `Values`. It exposes a list of "Operands" that are all of the `Values` that the User is referring to. The `User` class itself is a subclass

of `Value`.

The operands of a `User` point directly to the LLVM `Value` that it refers to. Because LLVM uses Static Single Assignment (SSA) form, there can only be one definition referred to, allowing this direct connection. This connection provides the use-def information in LLVM.

Important Public Members of the `User` class

The `User` class exposes the operand list in two ways: through an index access interface and through an iterator based interface.

- `Value *getOperand(unsigned i)`
  `unsigned getNumOperands()`

  These two methods expose the operands of the `User` in a convenient form for direct access.

- `User::op_iterator` - Typedef for iterator over the operand list
  `op_iterator op_begin()` - Get an iterator to the start of the operand list.
  `op_iterator op_end()` - Get an iterator to the end of the operand list.

  Together, these methods make up the iterator based interface to the operands of a `User`.

The `Instruction` class

`#include "llvm/IR/Instruction.h"`

header source: Instruction.h

doxygen info: Instruction Class

Superclasses: User, Value

The `Instruction` class is the common base class for all LLVM instructions. It provides only a few methods, but is a very commonly used class. The primary data tracked by the `Instruction` class itself is the opcode (instruction type) and the parent BasicBlock the `Instruction` is embedded into. To represent a specific type of instruction, one of many subclasses of `Instruction` are used.

Because the `Instruction` class subclasses the User class, its operands can be accessed in the same way as for other `User`s (with the `getOperand()`/`getNumOperands()` and `op_begin()`/`op_end()` methods). An important file for the `Instruction` class is the `llvm/Instruction.def` file. This file contains some meta-data about the various different types of instructions in LLVM. It describes the enum values that are used as opcodes (for example `Instruction::Add` and `Instruction::ICmp`), as well as the concrete sub-classes of `Instruction` that implement the instruction (for example BinaryOperator and CmpInst). Unfortunately, the use of macros in this file confuses doxygen, so these enum values don't show up correctly in the doxygen output.

Important Subclasses of the `Instruction` class

- `BinaryOperator`

  This subclasses represents all two operand instructions whose operands must be the same type, except for the comparison instructions.

- `CastInst` This subclass is the parent of the 12 casting instructions. It provides common operations on cast instructions.

- `CmpInst`

  This subclass represents the two comparison instructions, ICmpInst (integer operands), and FCmpInst (floating point operands).

Important Public Members of the `Instruction` class

- `BasicBlock *getParent()`

  Returns the BasicBlock that this `Instruction` is embedded into.

- `bool mayWriteToMemory()`

  Returns true if the instruction writes to memory, i.e. it is a `call`, `free`, `invoke`, or `store`.

- `unsigned getOpcode()`

  Returns the opcode for the `Instruction`.

- `Instruction *clone() const`

  Returns another instance of the specified instruction, identical in all ways to the original except that the instruction has no parent (i.e. it's not embedded into a BasicBlock), and it has no name.

The `Constant` class and subclasses

Constant represents a base class for different types of constants. It is subclassed by ConstantInt, ConstantArray, etc. for representing the various types of Constants. GlobalValue is also a subclass, which represents the address of a global variable or function.

Important Subclasses of Constant

- ConstantInt : This subclass of Constant represents an integer constant of any width.
  - `const APInt& getValue() const`: Returns the underlying value of this constant, an APInt value.
  - `int64_t getSExtValue() const`: Converts the underlying APInt value to an int64_t via sign extension. If the value (not the bit width) of the APInt is too large to fit in an int64_t, an assertion will result. For this reason, use of this method is discouraged.
  - `uint64_t getZExtValue() const`: Converts the underlying APInt value to a uint64_t via zero extension. IF the value (not the bit width) of the APInt is too large to fit in a uint64_t, an assertion will result. For this reason, use of this method is discouraged.
  - `static ConstantInt* get(const APInt& Val)`: Returns the ConstantInt object that represents the value provided by `Val`. The type is implied as the IntegerType that corresponds to the bit width of `Val`.
  - `static ConstantInt* get(const Type *Ty, uint64_t Val)`: Returns the ConstantInt object that represents the value provided by `Val` for integer type `Ty`.
- ConstantFP : This class represents a floating point constant.
  - `double getValue() const`: Returns the underlying value of this constant.
- ConstantArray : This represents a constant array.
  - `const std::vector<Use> &getValues() const`: Returns a vector of component constants that makeup this array.
- ConstantStruct : This represents a constant struct.
  - `const std::vector<Use> &getValues() const`: Returns a vector of component constants that makeup this array.
- GlobalValue : This represents either a global variable or a function. In either case, the value is a constant fixed address (after linking).

The `GlobalValue` class

`#include "llvm/IR/GlobalValue.h"`

header source: GlobalValue.h

doxygen info: GlobalValue Class

Superclasses: Constant, User, Value

Global values (GlobalVariables or Functions) are the only LLVM values that are visible in the bodies of all Functions. Because they are visible at global scope, they are also subject to linking with other globals defined in different translation units. To control the linking process, `GlobalValues` know their

linkage rules. Specifically, `GlobalValues` know whether they have internal or external linkage, as defined by the `LinkageTypes` enumeration.

If a `GlobalValue` has internal linkage (equivalent to being `static` in C), it is not visible to code outside the current translation unit, and does not participate in linking. If it has external linkage, it is visible to external code, and does participate in linking. In addition to linkage information, `GlobalValues` keep track of which Module they are currently part of.

Because `GlobalValues` are memory objects, they are always referred to by their address. As such, the Type of a global is always a pointer to its contents. It is important to remember this when using the `GetElementPtrInst` instruction because this pointer must be dereferenced first. For example, if you have a `GlobalVariable` (a subclass of `GlobalValue`) that is an array of 24 ints, type `[24 x i32]`, then the `GlobalVariable` is a pointer to that array. Although the address of the first element of this array and the value of the `GlobalVariable` are the same, they have different types. The `GlobalVariable`'s type is `[24 x i32]`. The first element's type is `i32`. Because of this, accessing a global value requires you to dereference the pointer with `GetElementPtrInst` first, then its elements can be accessed. This is explained in the LLVM Language Reference Manual.

Important Public Members of the `GlobalValue` class

- ```
  bool hasInternalLinkage() const
  bool hasExternalLinkage() const
  void setInternalLinkage(bool HasInternalLinkage)
  ```

  These methods manipulate the linkage characteristics of the `GlobalValue`.

- ```
  Module *getParent()
  ```

  This returns the Module that the GlobalValue is currently embedded into.

The `Function` class

```
#include "llvm/IR/Function.h"
```

header source: Function.h

doxygen info: Function Class

Superclasses: GlobalValue, Constant, User, Value

The `Function` class represents a single procedure in LLVM. It is actually one of the more complex classes in the LLVM hierarchy because it must keep track of a large amount of data. The `Function` class keeps track of a list of BasicBlocks, a list of formal Arguments, and a SymbolTable.

The list of BasicBlocks is the most commonly used part of `Function` objects. The list imposes an implicit ordering of the blocks in the function, which indicate how the code will be laid out by the backend. Additionally, the first BasicBlock is the implicit entry node for the `Function`. It is not legal in LLVM to explicitly branch to this initial block. There are no implicit exit nodes, and in fact there may be multiple exit nodes from a single `Function`. If the BasicBlock list is empty, this indicates that the `Function` is actually a function declaration: the actual body of the function hasn't been linked in yet.

In addition to a list of BasicBlocks, the `Function` class also keeps track of the list of formal Arguments that the function receives. This container manages the lifetime of the Argument nodes, just like the BasicBlock list does for the BasicBlocks.

The SymbolTable is a very rarely used LLVM feature that is only used when you have to look up a value by name. Aside from that, the SymbolTable is used internally to make sure that there are not conflicts between the names of Instructions, BasicBlocks, or Arguments in the function body.

Note that `Function` is a GlobalValue and therefore also a Constant. The value of the function is its address (after linking) which is guaranteed to be constant.

Important Public Members of the `Function`

- `Function(const FunctionType *Ty, LinkageTypes Linkage, const std::string &N = "", Module* Parent = 0)`

  Constructor used when you need to create new `Function`s to add the program. The constructor must specify the type of the function to create and what type of linkage the function should have. The FunctionType argument specifies the formal arguments and return value for the function. The same FunctionType value can be used to create multiple functions. The `Parent` argument specifies the Module in which the function is defined. If this argument is provided, the function will automatically be inserted into that module's list of functions.

- `bool isDeclaration()`

  Return whether or not the `Function` has a body defined. If the function is "external", it does not have a body, and thus must be resolved by linking with a function defined in a different translation unit.

- `Function::iterator` - Typedef for basic block list iterator
  `Function::const_iterator` - Typedef for const_iterator.
  `begin()`, `end()`, `size()`, `empty()`

  These are forwarding methods that make it easy to access the contents of a `Function` object's BasicBlock list.

- `Function::BasicBlockListType &getBasicBlockList()`

  Returns the list of BasicBlocks. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `Function::arg_iterator` - Typedef for the argument list iterator
  `Function::const_arg_iterator` - Typedef for const_iterator.
  `arg_begin()`, `arg_end()`, `arg_size()`, `arg_empty()`

  These are forwarding methods that make it easy to access the contents of a `Function` object's Argument list.

- `Function::ArgumentListType &getArgumentList()`

  Returns the list of Argument. This is necessary to use when you need to update the list or perform a complex action that doesn't have a forwarding method.

- `BasicBlock &getEntryBlock()`

  Returns the entry `BasicBlock` for the function. Because the entry block for the function is always the first block, this returns the first block of the `Function`.

- `Type *getReturnType()`
  `FunctionType *getFunctionType()`

  This traverses the Type of the `Function` and returns the return type of the function, or the FunctionType of the actual function.

- `SymbolTable *getSymbolTable()`

  Return a pointer to the SymbolTable for this `Function`.

The `GlobalVariable` class

`#include "llvm/IR/GlobalVariable.h"`

header source: GlobalVariable.h

doxygen info: GlobalVariable Class

Superclasses: GlobalValue, Constant, User, Value

Global variables are represented with the (surprise surprise) `GlobalVariable` class. Like functions, `GlobalVariables` are also subclasses of GlobalValue, and as such are always referenced by their address (global values must live in memory, so their "name" refers to their constant address). See

GlobalValue for more on this. Global variables may have an initial value (which must be a Constant), and if they have an initializer, they may be marked as "constant" themselves (indicating that their contents never change at runtime).

Important Public Members of the `GlobalVariable` class

- `GlobalVariable(const Type *Ty, bool isConstant, LinkageTypes &Linkage, Constant *Initializer = 0, const std::string &Name = "", Module* Parent = 0)`

  Create a new global variable of the specified type. If `isConstant` is true then the global variable will be marked as unchanging for the program. The Linkage parameter specifies the type of linkage (internal, external, weak, linkonce, appending) for the variable. If the linkage is InternalLinkage, WeakAnyLinkage, WeakODRLinkage, LinkOnceAnyLinkage or LinkOnceODRLinkage, then the resultant global variable will have internal linkage. AppendingLinkage concatenates together all instances (in different translation units) of the variable into a single variable but is only applicable to arrays. See the LLVM Language Reference for further details on linkage types. Optionally an initializer, a name, and the module to put the variable into may be specified for the global variable as well.

- `bool isConstant() const`

  Returns true if this is a global variable that is known not to be modified at runtime.

- `bool hasInitializer()`

  Returns true if this `GlobalVariable` has an initializer.

- `Constant *getInitializer()`

  Returns the initial value for a `GlobalVariable`. It is not legal to call this method if there is no initializer.

The `BasicBlock` class

`#include "llvm/IR/BasicBlock.h"`

header source: BasicBlock.h

doxygen info: BasicBlock Class

Superclass: Value

This class represents a single entry single exit section of the code, commonly known as a basic block by the compiler community. The `BasicBlock` class maintains a list of Instructions, which form the body of the block. Matching the language definition, the last element of this list of instructions is always a terminator instruction.

In addition to tracking the list of instructions that make up the block, the `BasicBlock` class also keeps track of the Function that it is embedded into.

Note that `BasicBlock`s themselves are Values, because they are referenced by instructions like branches and can go in the switch tables. `BasicBlock`s have type `label`.

Important Public Members of the `BasicBlock` class

- `BasicBlock(const std::string &Name = "", Function *Parent = 0)`

  The `BasicBlock` constructor is used to create new basic blocks for insertion into a function. The constructor optionally takes a name for the new block, and a Function to insert it into. If the `Parent` parameter is specified, the new `BasicBlock` is automatically inserted at the end of the specified Function, if not specified, the BasicBlock must be manually inserted into the Function.

- `BasicBlock::iterator` – Typedef for instruction list iterator
  `BasicBlock::const_iterator` – Typedef for const_iterator.

`begin()`, `end()`, `front()`, `back()`, `size()`, `empty()` STL-style functions for accessing the instruction list.

These methods and typedefs are forwarding functions that have the same semantics as the standard library methods of the same names. These methods expose the underlying instruction list of a basic block in a way that is easy to manipulate. To get the full complement of container operations (including operations to update the list), you must use the `getInstList()` method.

- `BasicBlock::InstListType &getInstList()`

  This method is used to get access to the underlying container that actually holds the Instructions. This method must be used when there isn't a forwarding function in the `BasicBlock` class for the operation that you would like to perform. Because there are no forwarding functions for "updating" operations, you need to use this if you want to update the contents of a `BasicBlock`.

- `Function *getParent()`

  Returns a pointer to Function the block is embedded into, or a null pointer if it is homeless.

- `Instruction *getTerminator()`

  Returns a pointer to the terminator instruction that appears at the end of the `BasicBlock`. If there is no terminator instruction, or if the last instruction in the block is not a terminator, then a null pointer is returned.

The `Argument` class

This subclass of Value defines the interface for incoming formal arguments to a function. A Function maintains a list of its formal arguments. An argument has a pointer to the parent Function.