# Introduction

In this document we will take a look at how to map various classic high-level programming language constructs to LLVM IR. The purpose of the document is to make the learning curve less steep for aspiring LLVM users.

For the sake of simplicity, we'll be working with a 32-bit target machines so that pointers and word-sized operands are 32-bits.

Also, for the sake of readability we do not mangle (encode) names. Rather, they are given simple, easy-to-read names that reflect their purpose. A production compiler for any language but C would generally need to mangle the names so as to avoid conflicts between symbols.

# A Quick Primer

Here are a few things that you should know before reading this document:

1. LLVM IR is not machine code, but sort of the step just above assembly.
2. LLVM IR is highly typed so expect to be told when you do something wrong.
3. LLVM IR does not differentiate between signed and unsigned integers.
4. LLVM IR assumes two's complement signed integers so that say `trunc` works equally well on signed and unsigned integers.
5. Global symbols begin with an ampersand (@).
6. Local symbols begin with a percent symbol (%).

7. All symbols must be declared or defined before use.
8. Don't worry that the LLVM IR at times can see somewhat lengthy when it comes to expressing something; the optimizer will ensure the output is well optimized and you'll often see two or three LLVM IR instructions be coalesced into a single machine code instruction.
9. If in doubt, consult the LangRef. If there is a conflict between the Language Reference and this document, the latter is wrong!
10. All LLVM IR examples have been compiled successfully but not run.

Finally: This is an informal guide, not a formal scientific paper. So your mileage may vary. Some will find this document highly useful, others will have no use for it. Hopefully, though, it will aid you in getting up to speed with LLVM.

# Mapping Basic Constructs into LLVM IR

In this chapter, we'll look at the most basic and simple constructs that are part of nearly all imperative/OOP languages out there.

## Global Variables

Global varibles are trivial to implement in LLVM IR:

```
int variable = 14;

int test()
{
   return variable;
}
```

```
@variable = global i32 14

define i32 @test() nounwind {
   %1 = load i32* @variable
   ret i32 %1
}
```

Please notice that LLVM views global variables as pointers; so you must explicitly dereference the global variable using the `load` instruction when accessing its value.

## Local Variables

There are basically two kinds of local variables in LLVM:

1. Register-allocated local variables (temporaries).
2. Stack-allocated local variables.

The former is created by introducing a new symbol for the variable:

```
%1 = ... result of some computation ...
```

The latter is created by allocating the variable on the stack:

```
%2 = alloca i32
```

Please notice that `alloca` yields a pointer to the allocated type. As is generally the case in LLVM, you must explicitly use a `load` or `store` instruction to read or write the value respectively.

## Constants

There are two different kinds of constants:

1. Constants that do *not* occupy allocated memory.
2. Constants that *do* occupy allocated memory.

The former are always expanded inline by the compiler as there seems to be no LLVM IR equivalent of those. In other words, the compiler simply inserts the constant value wherever it is being used in a

computation.

Constants that do occupy memory are defined using the `constant` keyword:

```
@hello = internal constant [6 x i8] c"hello\00"
%struct = type { i32, i8 }
@struct_constant = internal constant %struct { i32 16, i8 4 }
```

## Function Prototypes

A function prototype, aka a profile, is translated into an equivalent `declare` declaration in LLVM IR:

```
int Bar(int value);
```

Becomes:

```
declare i32 @Bar(i32 %value)
```

Or you can leave out the descriptive parameter name:

```
declare i32 @Bar(i32)
```

## Function Definitions

The translation of function definitions depends on a range of factors, ranging from the calling convention in use, whether the function is exception-aware or not, and whether the function is to be publicly available outside the module.

### Simple Functions

The most basic model is:

```
int Bar(void)
{
   return 17;
}
```

Becomes:

```
define i32 @Bar() nounwind
{
   ret i32 17
}
```

## Function Pointers

Function pointers are expressed almost like in C and C++:

```
int (*Function)(char *buffer);
```

Becomes:

```
@Function = global i32(i8*)* null
```

## Casts

Casts are basically six different types of casts:

1. Bitwise casts (type casts).
2. Zero-extending casts (unsigned upcast).
3. Sign-extending casts (signed upcast).
4. Truncating casts (signed and unsigned downcast).
5. Floating-point extending casts (float upcast).
6. Floating-point reducing casts (float downcasts).

### Bitwise Casts

A bitwise cast (`bitcast`) basically reinterprets a given bit pattern without changing any bits in the operand. For instance, you could make a bitcast of a pointer to byte into a pointer to some structure as follows:

```
typedef struct
{
   int a;
} Foo;

extern void *malloc(size_t size);
extern void *free(void *value);

void allocate()
{
   Foo *foo = (Foo *) malloc(sizeof(Foo));
   foo.a = 12;
   free(foo);
}
```

Becomes:

```
%Foo = type { i32 }

declare i8* @malloc(i32)
declare void @malloc(i8*)

define void @allocate() nounwind {
   %1 = call i8* @malloc(i32 4)
   %foo = bitcast i8* %1 to %Foo*
   %2 = getelementptr inbounds %Foo* %foo, i32 0, i32 0
   store i32 2, i32* %2
   ret i32 %3
}
```

## Zero-Extending Casts (Unsigned Upcasts)

To upcast an unsigned value like in the example below:

```
uint8 byte = 117;
uint32 word;

void main()
{
   /* The compiler automatically upcasts the byte to a word. */
   word = byte;
}
```

You use the `zext` instruction:

```
@byte = global i8 117
@word = global i32 0

define void @main() nounwind {
   %1 = load i8* @byte
   %2 = zext i8 %1 to i32
   store i32 %2, i32* @word
   ret void
}
```

## Truncating Casts (Signed and Unsigned Downcasts)

Both signed and unsigned integers use the same instruction, `trunc`, to reduce the size of the number in question. This is because LLVM IR assumes that all signed integer values are in two's complement format for which reason `trunc` is sufficient to handle both cases:

```
@int = global i32 -1
@char = global i8 0

define void @main() nounwind {
   %1 = load i32* @int
   %2 = trunc i32 %1 to i8
   store i8 %2, i8* @char
   ret void
}
```

### Sign-Extending Casts (Signed Upcasts)

To upcast a signed value, you replace the `zext` instruction with the `sext` instruction and everything else works just like in the previous section:

```
@char = global i8 -17
@int  = global i32 0

define void @main() nounwind {
   %1 = load i8* @char
   %2 = sext i8 %1 to i32
   store i32 %2, i32* @int
   ret void
}
```

## Incomplete Structure Types

Incomplete types are very useful for hiding the details of what a given structure has of fields. A well-designed C interface can be made so that no details of the structure are revealed to the client so that the client cannot inspect or modify private members inside the structure:

```
void Bar(struct Foo *);
```

Becomes:

```
%Foo = type opaque
declare void @Bar(%Foo)
```

## Structures

LLVM IR already includes the concept of structures so there isn't much to do:

```
struct Foo
{
   size_t _length;
};
```

It is only a matter of discarding the actual field names and then index by numerals starting from zero:

```
%Foo = type { i32 }
```

# Mapping Advanced Constructs to LLVM IR

In this chapter, we'll look at various non-OOP constructs that are highly useful and are becoming more and more widespread in use.

## Lambda Functions

A lambda function is basically an anonymous function with the added spice that it may freely refer to the local variables (including argument variables) in the containing function. Lambdas are implemented just like Pascal's nested functions, except the compiler is responsible for generating an internal name for the lambda function. There are a few different ways of implementing lambda functions (see Wikipedia on nested functions for more information).

I'll give an example in pseudo-C++ because C++ does not incorporate lambda functions:

```
int foo(int a)
{
   auto function = lambda(int x) { return x + a; }
   return function(10);
}
```

Here the "problem" is that the lambda function references a local variable, namely a, even though it is a function of its own. The easiest and most generic way of implementing lambda functions is to pass in a special pointer, say `parent`, that points to the frame of the caller of the lambda function. Another implementation is to parameterize the local variables that the lambda function uses and pass those into it.

The first solution looks something like this:

```
define i32 @lambda(i8* %parent, i32 %x) nounwind {
   %1 = bitcast i8* %parent to i32*
   %2 = load i32* %1       ; %2 = a
   %3 = add i32 %x, %2
   ret i32 %3
}

declare i8* @llvm.frameaddress(i32 %level)

define i32 @foo(i32 %a) nounwind {
   %1 = call i8* @llvm.frameaddress(i32 0)
   %2 = call i32 @lambda(i8* %1, i32 10)
   ret i32 %2
}
```

The other solution is straightforward and looks like this:

```
define i32 @lambda(i32 %a, i32 %x) nounwind {
   %1 = add i32 %a, %x
   ret i32 %1
}

define i32 @foo(i32 %a) nounwind {
   %1 = call i32 @lambda(i32 %a, i32 10)
   ret i32 %1
}
```

Given the fact that the reference manual warns about the use of the `@llvm.frameaddress` intrinsic function, the second solution is to be preferred.

## Closures

**TODO:** Describe closures.

## Generators

**TODO:** Describe generators.

# Mapping Exception Handling to LLVM IR

Exceptions can be implemented in one of three ways:

1. The simple way by using a propagated return value.
2. The bulky way by using `setjmp` and `longjmp`.
3. The efficient way by using zero overhead stack unwinding.

Please notice that many compiler developers with respect for themselves won't accept the first method as a proper way of handling exceptions. However, it is unbeatable in terms of simplicity and

can likely help people to understand that implementing exceptions does not need to be a nightmare.

The second method is used by some production compilers, but it has a large overhead both in terms of code bloat and the cost of a `try-catch` statement also becomes quite high because all CPU registers are saved using `setjmp` whenever a `try` statement is encountered.

The third method is very advanced but in return is zero cost in terms of not adding any explicit code to save registers or check return values. I do feel, however, that the third method is hard on the often limited caches that are on contemporary CPUs: The code size simply seems to explode when you use the second or third method, something that it doesn't with the first method.

Furthermore, the third form also mandates that some sort of stack frame is set up in functions that can throw exceptions, something that the simple method does not require. I personally believe that the reason C is still in use in many contemporary projects, such as the Linux kernel, is that C *is* faster than C++ when you compare return-value-checking C code with exception-throwing C++ code. I have never formally measured this, but my experience tells me that there is a reason why some C++ programmers avoid exceptions, this to such an extent that you often have to enable exception handling support with an explicit compiler option.

## Exception Handling by Propagated Return Value

This method basically is a compiler-generated way of implicitly checking each function's return value. Its main advantage is that it is simple - at the cost of mostly unproductive checks of return values.

```
void Bar()
{
   Foo foo;
   try
   {
      foo.SetLength(17);
      throw new Error("Out of sensible things to do!");
   }
   catch (Error *that)
   {
      foo.SetLength(24);
      delete that;
   }
}
```

This maps to the following code:

```
struct Exception *Bar()
{
   /* standard prologue */
   struct Exception *status = NULL;

   struct Foo foo;
   Foo_Create(&foo);

   /* "try" statement becomes this: */

   /* Body of "try" statement becomes this: */
   Foo_SetLength(&foo, 17);
   status = (struct Exception *) malloc(sizeof(struct Exception));
   Exception_Create(status, "Out of sensible things to do!");

   /* "catch" block becomes this: */
   if (status != null)
   {
      if (inheritsfrom(status, "Exception"))
      {
         Foo_SetLength(&foo, 24);
         Exception_Delete(status);
```

```
        free(status);
        status = NULL;
     }
   }

   Foo_Delete(&foo);

   return status;
}
```

```
%Exception = type { i32, i8* }

@.Exception_class_name = internal constant [10 x i8] c"Exception\00"

@.message = internal constant [30 x i8] c"Out of sensible things to do!\00"

declare i8* @malloc(i32)
declare void @free(i8*)

declare i32 @RuntimeObjectInherits(%Exception* %child, i8* %basename)

define %Exception* @Bar() nounwind {
   %foo = alloca %Foo, align 4
   call void @Foo_Create_Default(%Foo* %foo)

   ; "try" statement becomes this:

   ; Body of "try" statement becomes this:
   call void @Foo_SetLength(%Foo* %foo, i32 17)
   %1 = call i8* @malloc(i32 8)
   %status = bitcast i8* %1 to %Exception*
   %2 = getelementptr inbounds [30 x i8]* @.message, i32 0, i32 0
   call void @Exception_Create_String(%Exception* %status, i8* %2)

   %catch = icmp ne %Exception* %status, null
   br i1 %catch, label %.catch_begin, label %.catch_close

.catch_begin:
   %3 = getelementptr inbounds [10 x i8]* @.Exception_class_Name, i32 0, i32
   %4 = call i32 @RuntimeObjectInherits(%Exception* %status, i8* %3)
   %match = icmp eq i32 %4, 1
   br i1 %match, label %exception_begin, label %exception_close

.exception_begin:
   call void @Foo_SetLength(%Foo* %foo, i32 24)
   call void @Exception_Delete(%Exception* %status)
   %5 = bitcast %Exception* %status to i8*
   call void @free(i8* %5)

.exception_close:

.catch_close:
```

**TODO:** Finish up classic "return value propagated exception handling".

## `setjmp`/`longjmp` Exception Handling

The basic idea behind the `setjmp` and `longjmp` exception handling scheme is that you save the CPU state whenever you encounter a `try` keyword and then do a `longjmp` whenever you throw an exception. If there are few `try` blocks in the program, as is typically the case, the cost of this method is not as high as it might seem. However, often there are implicit exception handlers due to the need to release local resources such as class instances allocated on the stack and then the cost can become

quite high.

```
#include <stdio.h>

class Exception
{
public:
   Exception(const char *text)
   {
      _text = text;
   }

   const char *GetText() const
   {
      return _text;
   }

private:
   const char *_text;
};

int main(int argc, const char *argv[])
{
   int result = EXIT_FAILURE;

   try
   {
      if (argc == 1)
         throw Exception("Syntax: 'program' source-file target-file");

      result = EXIT_SUCCESS;
   }
   catch (Exception that)
   {
      puts(that.GetText());
   }

   return result;
}
```

This translates into something like this:

```
declare int @printf(i8*, ...)

; jmp_buf is very platform dependent, this is for illustration only...
%jmp_buf = type { i32 }
declare int @setjmp(%jmp_buf* %env)
declare void @longjmp(%jmp_buf* %env)

%Exception = type { i8* }

define void @Exception_Create(%Exception* %this, i32 %code, i8* %text) nounwi
   %1 = getelementptr %Exception* %this, i32 0, i32 0    ; %1 = &%this._text
   store i8** %1, %text
   ret void
}

define i8* @Exception_GetText(%Exception* %this) nounwind {
   %1 = getelementptr %Exception* %this, i32, i32 0      ; %1 = &%this._text
   %2 = load i8** %1
   ret i8* %2
}
```

```
define i32 @main(i32 %argc, i8** %argv) nounwind {
   ; "try" keyword expands into this:
   %1 = alloca %jmp_buf
   %2 = call i32 @setjmp(%jmp_buf* %1)

   ; if actual call to setjmp, the result is zero.
   ; if a longjmp, the result is non-zero.
   %3 = icmp eq i32 %2, 0
   br i1 %3, label %.saved, label %.catch

.saved:
   ; the body of the "try" statement expands to this:
   %4 = icmp eq i32 %argc, 1
   br i1 %4, label .if_begin, label .if_close

.if_begin:
   %5 = alloca %Exception

.catch:
```

**TODO:** Finish up `setjmp`/`longjmp` example.

## Zero Cost Exception Handling

**TODO:** Explain how to implement exception handling using zero cost exception handling.

# Mapping Object-Oriented Constructs to LLVM IR

In this chapter we'll look at various object-oriented constructs and see how they can be mapped to LLVM IR.

## Classes

A class is basically nothing more than a structure with an associated set of functions that take an implicit first parameter, namely a pointer to the structure. Therefore, is is very trivial to map a class to LLVM IR:

```
#include <stddef.h>

class Foo
{
public:
   Foo()
   {
      _length = 0;
   }

   size_t GetLength() const
   {
      return _length;
   }

   void SetLength(size_t value)
   {
      _length = value;
   }

private:
   size_t _length;
};
```

We first transform this code into two separate pieces:

1. The structure definition.
2. The list of methods, including the constructor.

```
; The structure definition for class Foo.
%Foo = type { i32 }

; The default constructor for class Foo.
define void @Foo_Create_Default(%Foo* %this) nounwind {
   %1 = getelementptr inbounds %Foo* %this, i32 0, i32 0
   store i32 0, i32* %1, align 4
   ret void
}

; The Foo::GetLength() method.
define void @Foo_GetLength(%Foo* %this) nounwind {
   %1 = getelementptr inbounds %Foo* %this, i32 0, i32 0
   %2 = load i32* %this, align 4
   ret i32 %2
}

; The Foo::SetLength() method.
define void @Foo_SetLength(%Foo* %this, i32 %value) nounwind {
   %1 = getelementptr inbounds %Foo* %this, i32 0, i32 0
   store i32 %value, i32* %1, align 4
   ret void
}
```

Then we make sure that the constructor (`Foo_Create_Default`) is invoked whenever an instance of the structure is created:

```
Foo foo;
```

```
%foo = alloca %Foo, align 4
call void @Foo_Create_Default(%Foo* %foo)
```

## Virtual Methods

A virtual method is basically no more than a compiler-controlled function pointer. Each virtual method is recorded in the `vtable`, which is a structure of all the function pointers needed by a given class:

```
class Foo
{
public:
   virtual int GetLengthTimesTwo() const
   {
      return _length * 2;
   }

   void SetLength(size_t value)
   {
      _length = value;
   }

private:
   int _length;
};

Foo foo;
foo.SetLength(4);
return foo.GetLengthTimesTwo();
```

This becomes:

```
%Foo_vtable_type = type { i32(%Foo*)* }
```

```
%Foo = type { %Foo_vtable_type*, i32 }

define i32 @Foo_GetLengthTimesTwo(%Foo* %this) nounwind {
   %1 = getelementptr inbounds %Foo* %this, i32 0, i32 1
   %2 = load i32* %1, align 4
   %3 = mul i32 %2, 2
   ret i32 %3
}

@Foo_vtable_data = global %Foo_vtable_type {
   i32(%Foo*)* @Foo_GetLengthTimesTwo
}

define void @Foo_Create_Default(%Foo* %this) nounwind {
   %1 = getelementptr inbounds %Foo* %this, i32 0, i32 0
   store %Foo_vtable_type* @Foo_vtable_data, %Foo_vtable_type** %1, align 4
   %2 = getelementptr inbounds %Foo* %this, i32 0, i32 1
   store i32 0, i32* %2, align 4
   ret void
}

define void @Foo_SetLength(%Foo* %this, i32 %value) nounwind {
   %1 = getelementptr inbounds %Foo* %this, i32 0, i32 1
   store i32 %value, i32* %1, align 4
   ret void
}

define i32 @main(i32 %argc, i8** %argv) nounwind {
   %foo = alloca %Foo, align 4
   call void @Foo_Create_Default(%Foo* %foo)
   call void @Foo_SetLength(%Foo* %foo, i32 4)
   %1 = getelementptr inbounds %Foo* %foo, i32 0, i32 0
   %2 = load %Foo_vtable_type** %1
   %3 = getelementptr inbounds %Foo_vtable_type* %2, i32 0, i32 0
   %4 = load i32(%Foo*)** %3
   %5 = call i32 %4(%Foo* %foo)
   ret i32 %5
}
```

Please notice that some C++ compilers store `_vtable` at a negative offset into the structure so that things like `memcpy(this, 0, sizeof(*this))` work, even though such commands should always be avoided in an OOP context.

## Single Inheritance

Single inheritance is very straightforward: Each "structure" (class) is laid out in memory after one another in declaration order.

```
class Base
{
public:
   void SetA(int value)
   {
      _a = value;
   }

private:
   int _a;
};

class Derived: public Base
```

```
{
public:
   void SetB(int value)
   {
      SetA(value);
      _b = value;
   }

protected:
   int _b;
}
```

Here, a and b will be laid out to follow one another in memory so that inheriting from a class is simply a matter of declaring a the base class as a first member in the inheriting class:

```
%Base = type {
   i32         ; '_a' in class Base
}

define void @Base_SetA(%Base* %this, i32 %value) nounwind {
   %1 = getelementptr %Base* %this, i32 0, i32 0
   store i32 %value, i32* %1
   ret void
}

%Derived = type {
   i32,        ; '_a' from class Base
   i32         ; '_b' from class Derived
}

define void @Derived_SetB(%Derived* %this, i32 %value) nounwind {
   %1 = bitcast %Derived* %this to %Base*
   call void @Base_SetA(%Base* %1, i32 %value)
   %2 = getelementptr %Derived* %this, i32 0, i32 1
   store i32 %value, i32* %2
   ret void
}
```

So the base class simply becomes plain members of the type declaration for the derived class.

And then the compiler must insert appropriate type casts whenever the derived class is being referenced as its base class as shown above with the `bitcast` operator.

## Multiple Inheritance

Multiple inheritance is not that difficult, either, it is merely a question of laying out the multiply inherited "structures" in order inside each derived class.

```
class BaseA
{
public:
   void SetA(int value)
   {
      _a = value;
   }

private:
   int _a;
};

class BaseB
{
public:
```

```
   void SetB(int value)
   {
      SetA(value);
      _b = value;
   }

private:
   int _b;
};

class Derived:
   public BaseA,
   public BaseB
{
public:
   void SetC(int value)
   {
      SetB(value);
      _c = value;
   }

private:
   int _c;
};
```

This is equivalent to the following LLVM IR:

```
%BaseA = type {
   i32          ; '_a' from BaseA
}

define void @BaseA_SetA(%BaseA* %this, i32 %value) nounwind {
   %1 = getelementptr %BaseA* %this, i32 0, i32 0
   store i32 %value, i32* %1
   ret void
}

%BaseB = type {
   i32,         ; '_a' from BaseA
   i32          ; '_b' from BaseB
}

define void @BaseB_SetB(%BaseB* %this, i32 %value) nounwind {
   %1 = bitcast %BaseB* %this to %BaseA*
   call void @BaseA_SetA(%BaseA* %1, i32 %value)
   %2 = getelementptr %BaseB* %this, i32 0, i32 1
   store i32 %value, i32* %2
   ret void
}

%Derived = type {
   i32,         ; '_a' from BaseA
   i32,         ; '_b' from BaseB
   i32          ; '_c' from Derived
}

define void @Derived_SetC(%Derived* %this, i32 %value) nounwind {
   %1 = bitcast %Derived* %this to %BaseB*
   call void @BaseB_SetB(%BaseB* %1, i32 %value)
   %2 = getelementptr %Derived* %this, i32 0, i32 2
   store i32 %value, i32* %2
   ret void
```

```
}
```

And the compiler then supplies the needed type casts and pointer arithmentic whenever `baseB` is being referenced as an instance of `BaseB`. Please notice that all it takes is a `bitcast` from one class to another as well as an adjustment of the last argument to `getelementptr`.

## Virtual Inheritance

Virtual inheritance is actually quite simple as it dictates that identical base classes are to be merged into a single occurence. For instance, given this:

```
class BaseA
{
public:
   int a;
};

class BaseB: public BaseA
{
public:
   int b;
};

class BaseC: public BaseA
{
public:
   int c;
};

class Derived:
   public virtual BaseB,
   public virtual BaseC
{
   int d;
};
```

`Derived` will only contain a single instance of `BaseA` even if its inheritance graph dictates that it should have two instances. The result looks something like this:

```
class Derived
{
public:
   int a;
   int b;
   int c;
   int d;
};
```

So the second instance of `a` is silently ignored because it would cause multiple instances of `BaseA` to exist in `Derived`, which would clearly cause lots of confusion and ambiguities.

## Interfaces

An interface is basically nothing more than a base class with no data members, where all the methods are pure virtual (i.e. has no body).

As such, we've already described how to convert an interface to LLVM IR - it is done precisely the same way that you convert a virtual member function to LLVM IR.

## Class Equivalence Test

There are basically two ways of doing this:

1. If you can guarantee that each class a unique `vtable`, you can simply compare the pointers to the `vtable`.
2. If you cannot guarantee that each class has a unique `vtable` (because different `vtables` may have been merged by the linker), you need to add a unique field to the `vtable` so that you can compare that instead.

The first variant goes roughly as follows (assuming identical strings aren't merged by the compiler, something that they are most of the time):

```
bool equal = (typeid(first) == typeid(other));
```

```
%object_vtable_type = type { i8* }
%object_vtable_data = internal constant { [8 x i8]* c"object\00" }

define i1 @typeequals(%object* %first, %object* %other) {
   %1 = getelementptr %object* %first, i32 0, i32 0
   %2 = load
   %2 = getelementptr %object* %other, i32 0, i32 0
```

As far as I know, RTTI is simply done by adding two fields to the `_vtable` structure: `parent` and `signature`. The former is a pointer to the vtable of the parent class and the latter is the mangled (encoded) name of the class. To see if a given class is another class, you simply compare the `signature` fields. To see if a given class is a derived class of some other class, you simply walk the chain of `parent` fields, while checking if you have found a matching signature.

## Class Inheritance Test

A class inheritance test is basically a question of the form:

Is class X identical to or derived from class Y?

To answer that question, we can use one of two methods:

1. The naive implementation where we search upwards in the chain of parents.
2. The faster implementation where we search a preallocated list of parents.

The naive implementation works as follows:

```
define @naive_instanceof(%object* %first, %object* %other) nounwind {
   ; compare the two instances
   %first1 = getelementptr %object %first, i32 0, i32 0
   %first2 = load %object* %first1
   %other1 = getelementptr %object %other, i32 0, i32 0
   %other2 = load %object* %other2
   %equal = icmp eq i32 %first2, %other2
   br i1 %equal, label @.match, label @.mismatch
.match:

   %2 = getelementptr %object %
   ; ascend up the chain of parents
```

## The new Operator

The new operator is generally nothing more than a type-safe version of the C `malloc` function - in some implementations of C++, they may even be called interchangeably without causing unseen or unwanted side-effects.

All calls of the form `new X` are mapped into:

```
declare i8* @malloc(i32) nounwind

%X = type { i8 }

define void @X_Create_Default(%X* %this) nounwind {
```

```
   %1 = getelementptr %X* %this, i32 0, i32 0
   store i8 0, i8* %1, align 4
   ret void
}

define void @test() nounwind {
   %1 = call i8* @malloc(i32 1)
   %2 = bitcast i8* %1 to %X*
   call void @X_Create_Default(%X* %2)
   ret void
}
```

Calls of the form `new X(Y, Z)` are the same, except `Y` and `Z` are passed into the constructor.

# Interfacing to the Operating System

I'll divide this chapter into two sections:

1. How to Interface to POSIX Operating Systems.
2. How to Interface to the Windows Operating System.

## How to Interface to POSIX Operating Systems

On POSIX, the presence of the C run-time library is an unavoidable fact for which reason it makes a lot of sense to directly call such C run-time functions.

### Sample "Hello World" Application

On POSIX, it is really very easy to create the `Hello world` program:

```
declare i32 @puts(i8* nocapture) nounwind

@.hello = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

define i32 @main(i32 %argc, i8** %argv) {
   %1 = getelementptr [13 x i8]* @.hello, i32 0, i32 0
   call i32 @puts(i8* %1)
   ret i32 0
}
```

## How to Interface to the Windows Operating System

On Windows, the C run-time library is mostly considered of relevance to the C and C++ languages only, so you have a plethora (thousands) of standard system interfaces that any client application may use.

### Sample "Hello World" Application

`Hello world` on Windows is nowhere as straightforward as on POSIX:

```
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64
target triple = "i686-pc-win32"

%struct._OVERLAPPED = type { i32, i32, %union.anon, i8* }
%union.anon = type { %struct.anon }
%struct.anon = type { i32, i32 }

declare dllimport x86_stdcallcc i8* @"\01_GetStdHandle@4"(i32) #1

declare dllimport x86_stdcallcc i32 @"\01_WriteFile@20"(i8*, i8*, i32, i32*,

@hello = internal constant [13 x i8] c"Hello world\0A\00", align 1
```

```
define i32 @main(i32 %argc, i8** %argv) nounwind {
   %1 = call i8* @"\01_GetStdHandle@4"(i32 -11)    ; -11 = STD_OUTPUT_HANDLE
   %2 = getelementptr [13 x i8]* @hello, i32 0, i32 0
   %3 = call i32 @"\01_WriteFile@20"(i8* %1, i8* %2, i32 12, i32* null, %stru
   ; todo: Check that %4 is not equal to -1 (INVALID_HANDLE_VALUE)
   ret i32 0
}

attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"
   "no-infs-fp-math"="fa lse" "no-nans-fp-math"="false" "stack-protector-buff
   "use-soft-float"="false"
}
```

**TODO:** What are the `\01` prefixes on the Windows names for? Do they represent Windows' way of exporting symbols or are they exclusive to Clang and LLVM?

# Appendix A: How to Implement a String Type in LLVM

There are two ways to implement a string type in LLVM:

1. To write the implementation in LLVM IR.
2. To write the implementation in a higher-level language that generates IR.

I'd personally much prefer to use the second method, but for the sake of the example, I'll go ahead and illustrate a simple but useful string type in LLVM IR. It assumes a 32-bit architecture, so please replace all occurences of `i32` with `i64` if you are targetting a 64-bit architecture.

It all boils down to making a suitable type definition for the class and then define a rich set of functions to operate on the type definition:

```
; The actual type definition for our 'String' type.
%String = type {
   i8*,     ; buffer: pointer to the character buffer
   i32,     ; length: the number of chars in the buffer
   i32,     ; maxlen: the maximum number of chars in the buffer
   i32      ; factor: the number of chars to preallocate when growing
}

define void @String_Create_Default(%String* %this) nounwind {
   ; Initialize 'buffer'.
   %1 = getelementptr %String* %this, i32 0, i32 0
   store i8* null, i8** %1, align 4

   ; Initialize 'length'.
   %2 = getelementptr %String* %this, i32 0, i32 1
   store i32 0, i32* %2, align 4

   ; Initialize 'maxlen'.
   %3 = getelementptr %String* %this, i32 0, i32 2
   store i32 0, i32* %3, align 4

  ; Initialize 'factor'.
  %4 = getelementptr %String* %this, i32 0, i32 3
  store i32 16, i32* %4, align 4

  ret void
}

declare i8* @malloc(i32)
declare void @free(i8*)
```

```
declare i8* @memcpy(i8*, i8*, i32)

define void @String_Delete(%String* %this) nounwind {
  ; Check if we need to call 'free'.
   %1 = getelementptr %String* %this, i32 0, i32 0
  %2 = load i8** %1
  %3 = icmp ne i8* %2, null
  br i1 %3, label %free_begin, label %free_close

free_begin:
  call void @free(i8* %2)
  br label %free_close

free_close:
  ret void
}

define void @String_Resize(%String* %this, i32 %value) {
   ; %output = malloc(%value)
   %output = call i8* @malloc(i32 %value)

  ; todo: check return value

  ; %buffer = this->buffer
  %1 = getelementptr %String* %this, i32 0, i32 0
  %buffer = load i8** %1

   ; %length = this->length
  %2 = getelementptr %String* %this, i32 0, i32 1
  %length = load i32* %2

  ; memcpy(%output, %buffer, %length)
  %3 = call i8* @memcpy(i8* %output, i8* %buffer, i32 %length)

  ; free(%buffer)
  call void @free(i8* %buffer)

   ; this->buffer = %output
  store i8* %output, i8** %1, align 4

  ret void
}

define void @String_Add_Char(%String* %this, i8 %value) {
  ; Check if we need to grow the string.
  %1 = getelementptr %String* %this, i32 0, i32 1
  %length = load i32* %1
  %2 = getelementptr %String* %this, i32 0, i32 2
  %maxlen = load i32* %2
  ; if length == maxlen:
  %3 = icmp eq i32 %length, %maxlen
  br i1 %3, label %grow_begin, label %grow_close

grow_begin:
  %4 = getelementptr %String* %this, i32 0, i32 3
  %factor = load i32* %4
  %5 = add i32 %maxlen, %factor
  call void @String_Resize(%String* %this, i32 %5)
  br label %grow_close

grow_close:
  %6 = getelementptr %String* %this, i32 0, i32 0
```

```
  %buffer = load i8** %6
  %7 = getelementptr i8* %buffer, i32 %length
  store i8 %value, i8* %7

  ret void
}
```

# Resources


1. Modern Compiler Implementation in Java, 2nd Edition.
2. Alex Darby's series of articles on low-level stuff.


# Epilogue

If you discover any errors in this document or you need more information than given here, please write to the friendly LLVM developers and they'll surely help you out or add the requested info to this document.