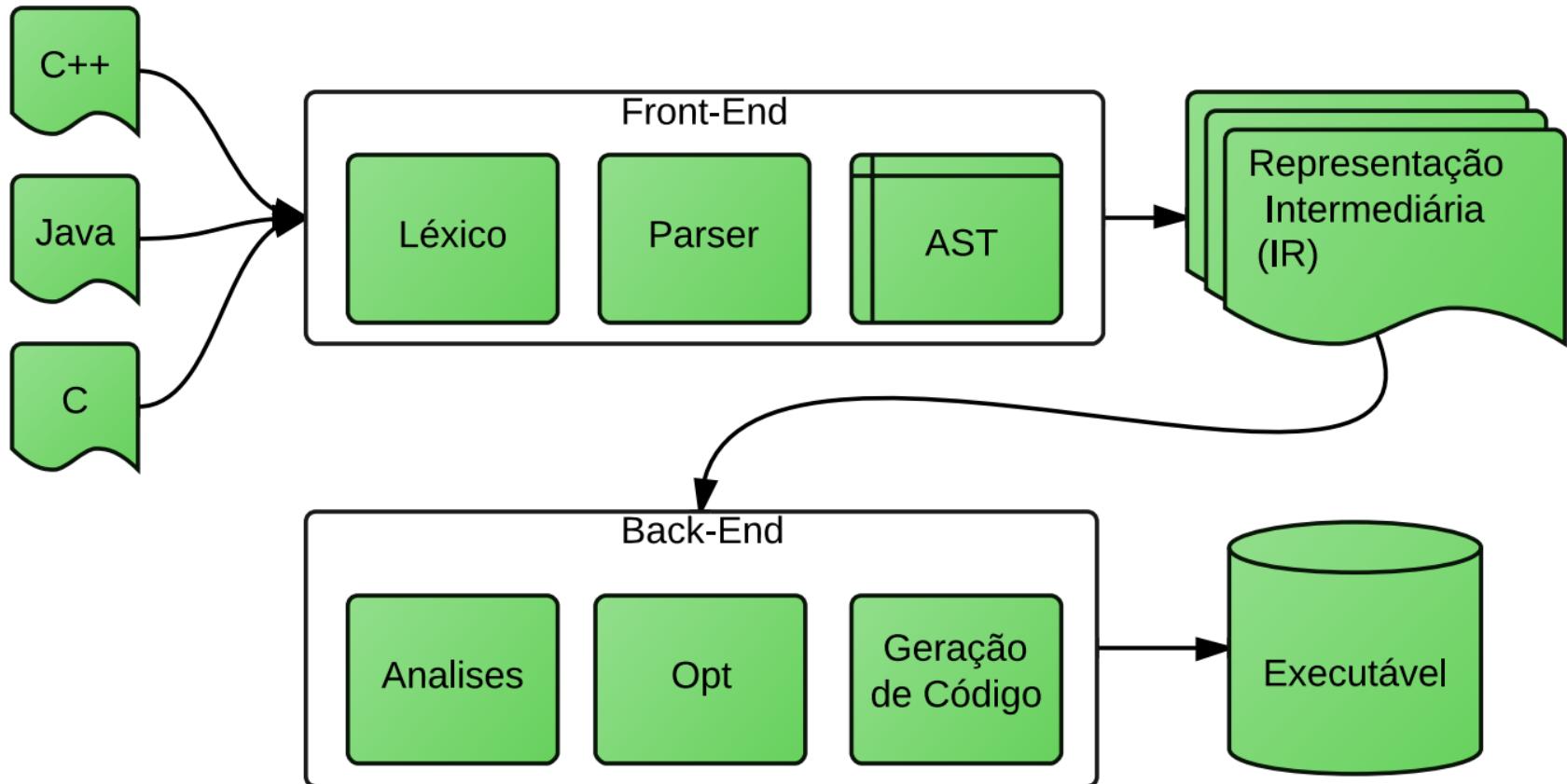


Um pouco de LLVM

Introdução e Ferramentas

Relembrando...



A infraestrutura LLVM

- Em 2000 surge a ideia na Universidade de Illinois:

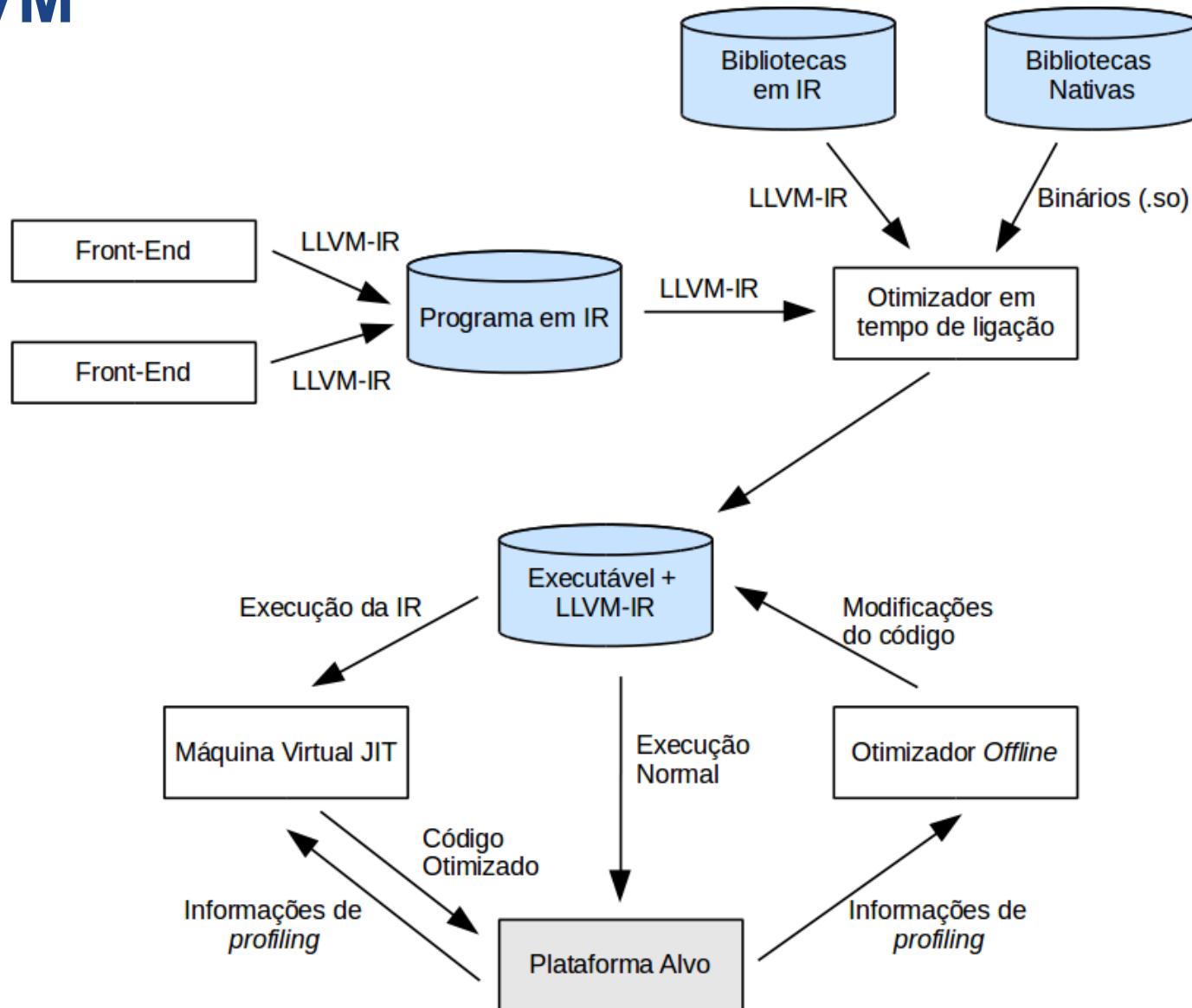
Fazer um novo Back-end para o GCC

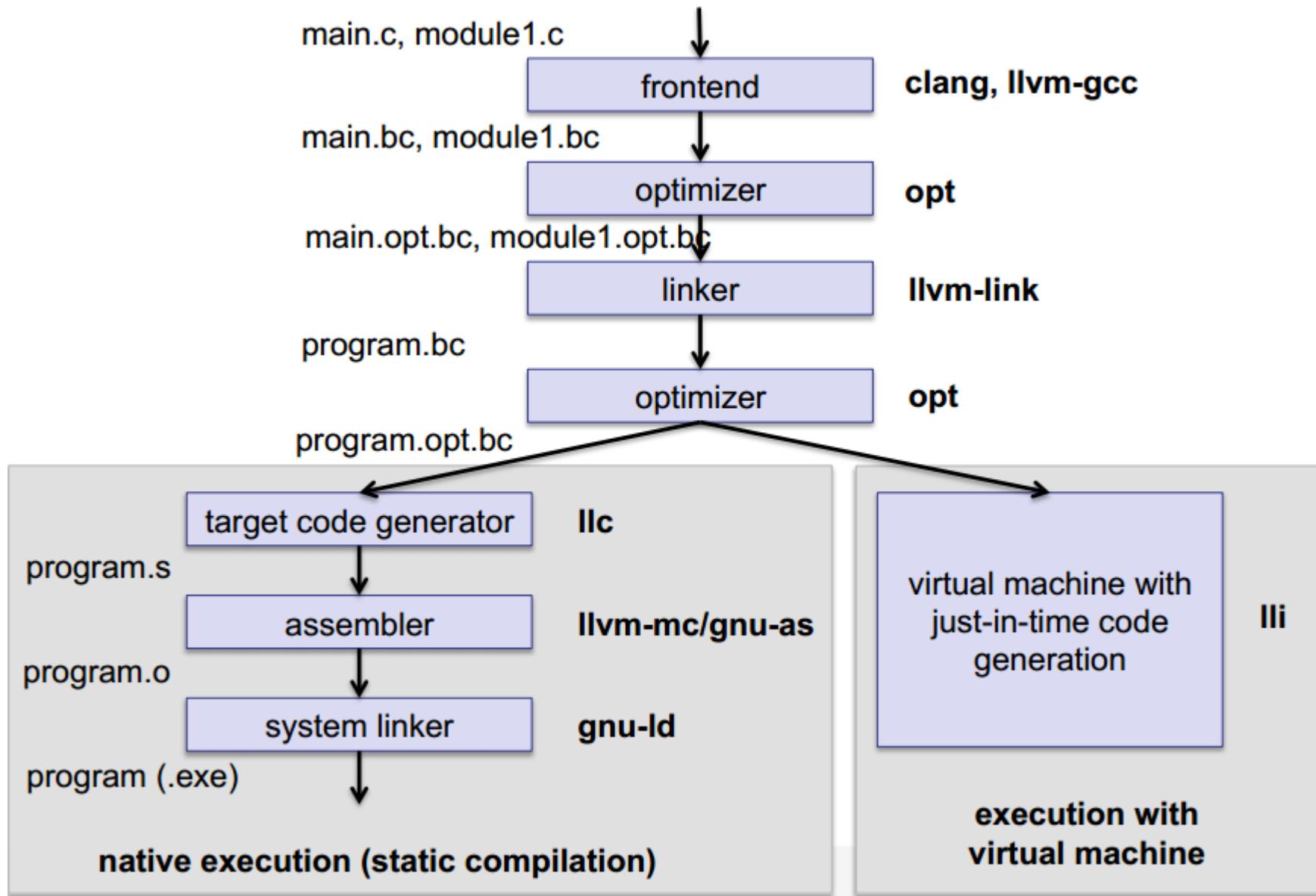
- ***Low Level Virtual Machine***
- Mais bem organizado
- Com componentes facilmente reutilizáveis
- Maior velocidade na construção novos compiladores
- Possibilidade de outros tipos de compilação
 - Baseado em trace
 - Com otimizações em *runtime* (JIT - *Just-in Time*)

Hoje, o que o LLVM é?

- **Um motor de execução poderoso**
 - É o compilador padrão da Apple
 - Tem como *sponsors* várias empresas (Google, Apple, Oracle, ...)
- **Uma coleção de bibliotecas**
 - Análises, otimizações, geração de código, JIT, garbage collection, perfilamento, ...
 - Bem integradas e documentadas
- **Uma coleção de ferramentas**
 - Assemblers, Debugger, Linker, Code Generator, Modular Optimizer, ...
- **Possui seu próprio Front-End: Clang**
 - A IR gerada pelo Clang é a LLVM-IR

LLVM





Comandos úteis

```
# Usando o Clang para gerar executável x86
$ clang main.c -o main

# Clang front-end emitindo arquivo texto LLVM-IR
$ clang main.c -S -emit-llvm -o main.ll

# Monta o arquivo texto LLVM-IR (main.ll) em um arquivo
# bitcode LLVM-IR (main.bc)
$ llvm-as -f main.ll -o main.bc

# Interpreta o bitcode ou o texto LLVM-IR
$ lli main.bc
$ lli main.ll
```

Comandos úteis

```
# A partir do bitcode LLVM-IR (main.bc) é possível gerar
# assembler x86 (main_x86.s)
$ llc main.bc -o main_x86.s

# Gerando código objeto com o GAS
$ as main_x86.s -o main.o

# Linkando as libs no objeto e gerando executável x86
$ gcc main.o -o main

# Executando
$ ./main
```

Nesta fase do curso...

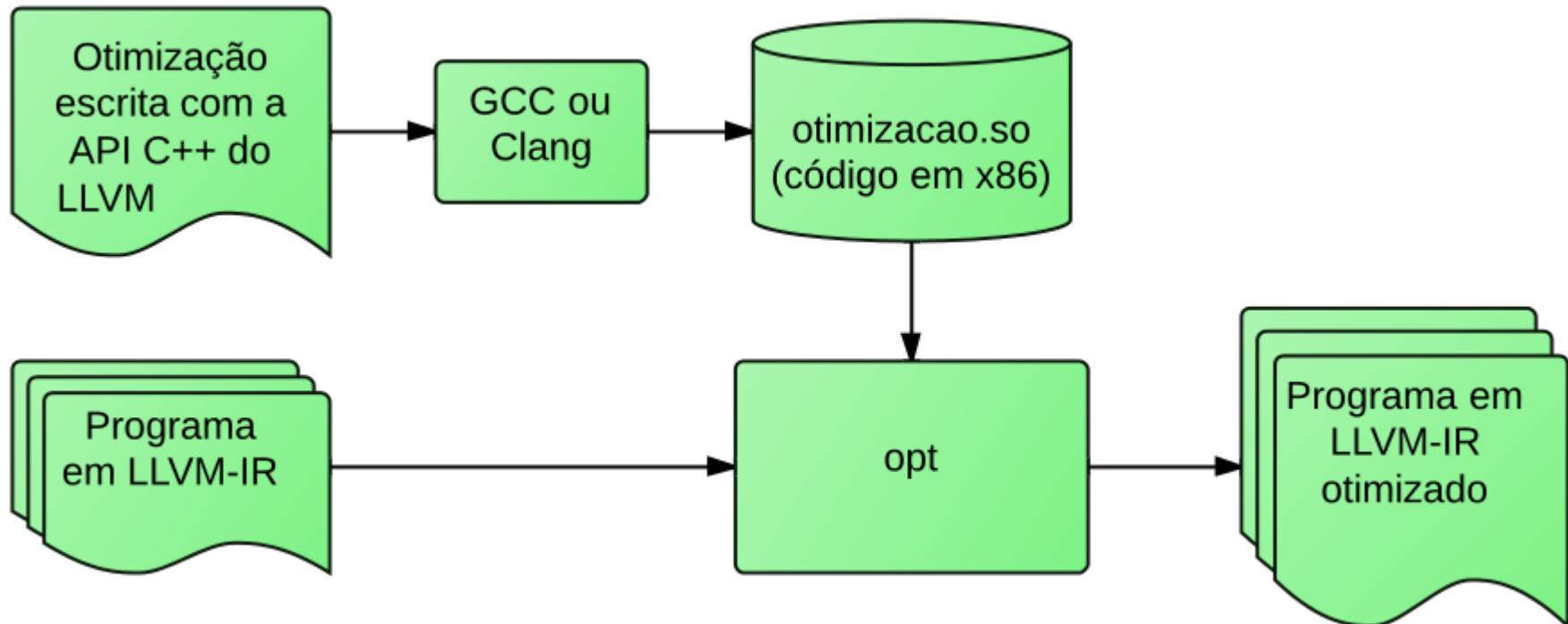
1. Apresentar a IR do LLVM
2. Estudar algumas IR's geradas com o Clang
3. Apresentar as classes em Java responsáveis pelo Front-End (até a AST) da linguagem minijava.
4. **Projeto 2: Gerar LLVM-IR para programas em minijava**
5. **Projeto 3: Implementar otimizações na LLVM-IR**

LLVM-IR: Representação Intermediária

- Fácil leitura
 - Linguagem Tipada
 - Arquitetura Load/Store
 - Procedural
- Independente de arquitetura
- Construída com SSA (*Static Single Assignment*)
- Infinitos registradores
- Três representações:
 - Human readable (file .ll)
 - Binary LLVM bitcode (file .bc)
 - in-Memory C++ data structures

Por que aprender LLVM-IR?

- O projeto será sobre Otimizações em LLVM
- Pra otimizar LLVM-IR precisamos conhecê-la



Instruções em LLVM-IR

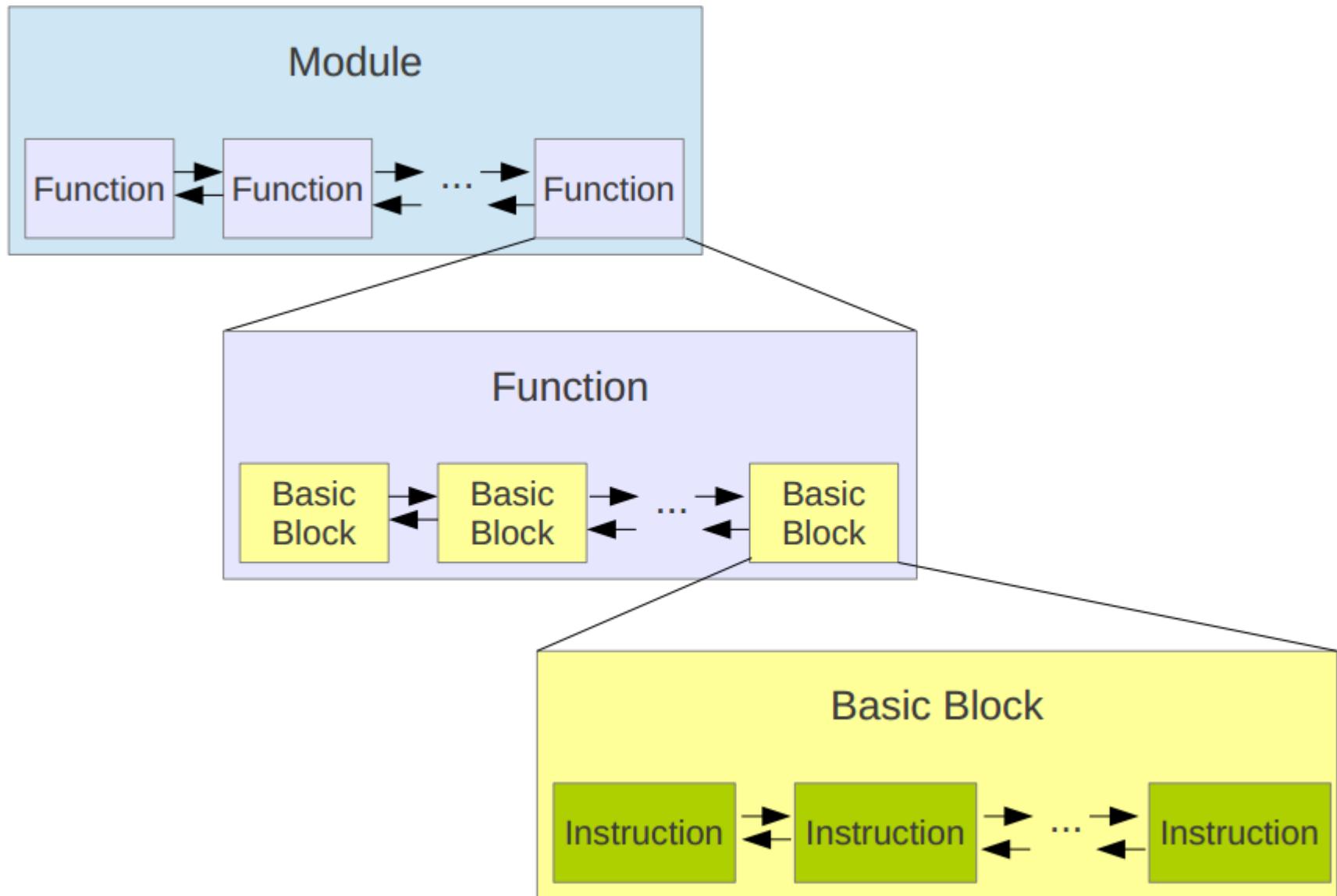
```
; Comment
define void @foo(i32 %a, i32* %b) {
    %var1 = sub i32 2, %a
    %cmp = icmp eq i32 %var1, 0
    br i1 %cmp, label %true, label %end

true:
    %var2 = load i32* %b

end:
    store i32 %var1, i32* %b
    ret void
}
```

Estrutura do programa

- **Module:** contém *Functions* e *GlobalVariables*
- **Function:** contém *BasicBlocks* e *Arguments*
- **BasicBlock:** contém uma lista de *Instructions*
- **Instructions:** *Opcode* + lista de *Operands*



SSA - Static Single Assignment

- Em LLVM-IR, um registrador não pode ser assinalado mais que uma vez:

Um novo registrador para cada resultado!

- Essa restrição facilita as otimizações

```
define i32* @foo(%struct.ST* %s) {
    %t1 = getelementptr %struct.ST* %s, i32 1
    %t2 = getelementptr %struct.ST* %t1, i32 0, i32 2
    %t3 = getelementptr %struct.RT* %t2, i32 0, i32 1
    %t4 = getelementptr [10 x [20 x i32]]* %t3, i32 0, i32 5
    %t5 = getelementptr [20 x i32]* %t4, i32 0, i32 13
    ret i32* %t5
}
```

Tipos em LLVM-IR

- LLVM-IR é tipado: todas as instruções e todos os valores possuem tipos definidos.
- LLVM-IR não é OO: não possui objetos
- Primitivos:
 - **i1, i2, ... i8, ... i16, ... i32**
 - **label, void**
 - **float e double**
- Derivados:
 - Array: **[40 x i32]**
 - Pointers: **[4 x i8]***
 - Structures: **{ i32, (i32)*, i1 }**
 - Function - <tipo_retorno> (<parametro_list>): **i32 (i32)**

Do alto nível para LLVM-IR

Parte 1

Classes

- MiniJava possui classes
- Porém, LLVM-IR não possui suporte a classes
 - Nenhuma linguagem intermediária possui
- Classes viram estruturas contendo seus atributos
 - Ver a classe **LlvmStructure** no MiniJava
- Os métodos viram funções simples
 - O nome da função deve ser único
 - O primeiro argumento da função deve ser um ponteiro para a estrutura da classe

```
class Matematica {  
    int x;  
    public int Soma(int a, int b){  
        int y;  
        y = 10;  
    }  
}
```

%class.Matematica = type { i32 } ; estrutura da classe contendo um inteiro 'x'

; O método Soma virou uma função com um nome único. Neste exemplo, junto
; nome do método com o nome da classe

```
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {  
entry0:
```

```
%a_addr = alloca i32  
store i32 %a, i32 * %a_addr  
%b_addr = alloca i32  
store i32 %b, i32 * %b_addr
```

Reserva um espaço na memória para cada argumento. Como SSA não permite reatribuição, temos usar a memória ao invés de variáveis.

```
%y = alloca i32  
store i32 10, i32 * %y  
ret i32 1
```

Aloco memória para 'y' e uso 'store' para atribuições

```
class Matematica {  
    int x;  
    public int Soma(int a, int b){  
        int y;  
        x = 10;  
    }  
}
```

%class.Matematica = type { i32 } ; estrutura da classe

```
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {  
entry0:  
    %a_addr = alloca i32  
    store i32 %a, i32 * %a_addr  
    %b_addr = alloca i32  
    store i32 %b, i32 * %b_addr  
    %y = alloca i32  
    %tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0  
    store i32 10, i32 * %tmp0  
    ret i32 1  
}
```

getelementptr

Detalhes em: <http://llvm.org/docs/GetElementPtr.html>

- <result> = **getelementptr** <pty>* <ptrval> {, <ty> <idx>}*
- O primeiro argumento sempre é um ponteiro
- Construção mais complexa do LLVM-IR
- Serve para acessar estruturas a partir do primeiro operando

Código em C

```
struct T {
    int f1;
    int f2;
};

void foo(struct T *PT) {
    PT[0].f1 = PT[1].f1 + PT[2].f2;
}
```

foo em LLVM-IR

```
define void %foo(%struct.T* %PT) {
entry:
    %tmp = getelementptr %struct.T* %PT, i32 1, i32 0
    %tmp5 = load i32* %tmp
    %tmp6 = getelementptr %struct.T* %PT, i32 2, i32 1
    %tmp7 = load i32* %tmp6
    %tmp8 = add i32 %tmp7, %tmp5
    %tmp9 = getelementptr %struct.T* %PT, i32 0, i32 0
    store i32 %tmp8, i32* %tmp9
ret void
}
```

Código em C

```
struct RT {
    char A;
    int B[10][20];
    char C;
};

struct ST {
    int X;
    double Y;
    struct RT Z;
};

int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}
```

foo em LLVM-IR

```
define i32* @foo(%struct.ST* %s) {
%tmp = getelementptr %struct.ST* %s, i32 1, i32 2, i32 1,
      i32 5, i32 13
ret i32 $tmp
}
```

Código em C

```
struct RT {
    char A;
    int B[10][20];
    char C;
};

struct ST {
    int X;
    double Y;
    struct RT Z;
};

int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}
```

foo em LLVM-IR

```
define i32* @foo(%struct.ST* %s) {
    %t1 = getelementptr %struct.ST* %s, i32 1
    %t2 = getelementptr %struct.ST* %t1, i32 0, i32 2
    %t3 = getelementptr %struct.RT* %t2, i32 0, i32 1
    %t4 = getelementptr [10 x [20 x i32]]* %t3, i32 0, i32 5
    %t5 = getelementptr [20 x i32]* %t4, i32 0, i32 13
    ret i32* %t5
}
```

Instanciação de Objetos

```
class Teste {
    public static void main(String[] a){
        System.out.println(
            new Matematica().Soma(1,2));
    }
}

class Matematica {
    int x;
    int[] v;
    Complex C;
    public int Soma(int a, int b){
        int y;
        y = 10;
        return 1;
    }
}
class Complex {
    int i;
}
```

```

@.formatting.string = private constant [4 x i8] c"%d\0A\00"
%class.Teste = type {}

%class.Matematica = type { i32, i32 *, %class.Complex * }

%class.Complex = type { i32 }

define i32 @main() {
entry0:
    %tmp0 = alloca i32
    store i32 0, i32 * %tmp0
    %tmp2 = mul i32 20, 1
    %tmp3 = call i8* @malloc( i32 %tmp2)
    %tmp1 = bitcast i8* %tmp3 to %class.Matematica*
    %tmp4 = call i32 @_Soma_Matematica(%class.Matematica * %tmp1, i32 1, i32 2)
    %tmp5 = getelementptr [4 x i8] * @.formatting.string, i32 0, i32 0
    %tmp6 = call i32 (i8 *, ...)* @printf(i8 * %tmp5, i32 %tmp4)
    %tmp7 = load i32 * %tmp0
    ret i32 %tmp7
}

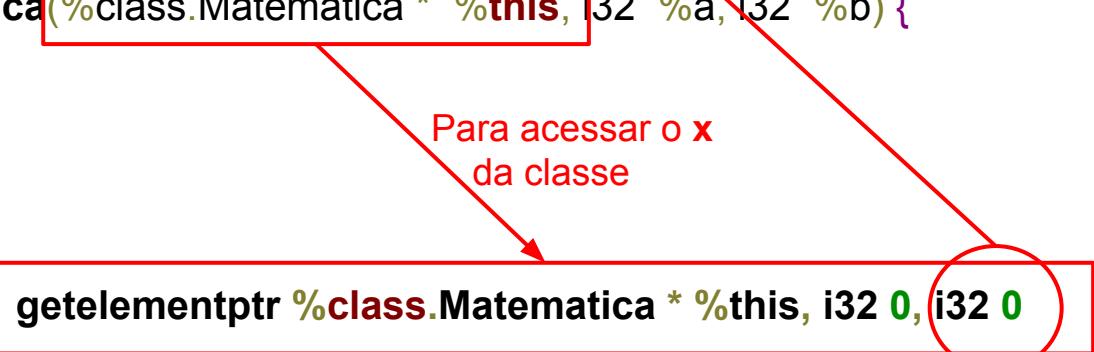
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
    %a_tmp = alloca i32
    store i32 %a, i32 * %a_tmp
    %b_tmp = alloca i32
    store i32 %b, i32 * %b_tmp
    %y = alloca i32
    store i32 10, i32 * %y
    ret i32 1
}

declare i32 @printf(i8 *, ...)
declare i8 * @malloc(i32)

```

Tamanho de %class.Matematica em x86_64
i32: 4 bytes, ptr: 8 bytes

```
@.formatting.string = private constant [4 x i8] c"%d\0A\00"
%class.Teste = type { }
%class.Matematica = type { i32, i32 *, %class.Complex * }
%class.Complex = type { i32 }
define i32 @main() {
entry0:
%tmp0 = alloca i32
store i32 0, i32 * %tmp0
%tmp2 = mul i32 20, 1
%tmp3 = call i8* @malloc( i32 %tmp2)
%tmp1 = bitcast i8* %tmp3 to %class.Matematica*
%tmp4 = call i32 @_Soma_Matematica(%class.Matematica * %tmp1, i32 1, i32 2)
%tmp5 = getelementptr [4 x i8] * @.formatting.string, i32 0, i32 0
%tmp6 = call i32 (i8 *, ...)* @printf(i8 * %tmp5, i32 %tmp4)
%tmp7 = load i32 * %tmp0
ret i32 %tmp7
}
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
%a_tmp = alloca i32
store i32 %a, i32 * %a_tmp
%b_tmp = alloca i32
store i32 %b, i32 * %b_tmp
%y = alloca i32
store i32 10, i32 * %y
ret i32 1
}
declare i32 @printf (i8 *, ...)
declare i8 * @malloc (i32)
```



Do alto nível para LLVM-IR

Parte 2

Herança: acesso aos atributos da classe pai

- Primeiro elemento da classe é sua classe pai

```
class Teste {  
    public static void main(String[] a){  
        System.out.println(new Matematica().Soma(1,2));  
    }  
}  
  
class Matematica extends Complex {  
    int x;  
    public int Soma(int a, int b){  
        i = 10;  
        return 10;  
    }  
}  
  
class Complex {  
    int i;  
}
```

```
@.formatting.string = private constant [4 x i8] c"%d\\0A\\00"
%class.Matematica = type { %class.Complex, i32 }
%class.Teste = type { }
%class.Complex = type { i32 }
define i32 @main() {
...
}
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
%a_tmp = alloca i32
store i32 %a, i32 * %a_tmp
%b_tmp = alloca i32
store i32 %b, i32 * %b_tmp
%tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0
%tmp1 = getelementptr %class.Complex * %tmp0, i32 0, i32 0
store i32 10, i32 * %tmp1
ret i32 10
}
declare i32 @printf (i8 *, ...)
declare i8 * @malloc (i32)
```

Herança: acesso aos métodos da classe pai

- Todo método é uma função global em LLVM-IR
- Apenas chame a função correta

```
class Teste {  
    public static void main(String[] a){  
        System.out.println(new Matematica().Soma(1,2));  
    }  
}  
  
class Matematica extends Complex {  
    int x;  
    public int Soma(int a, int b){  
        return this.Pow(1,2);  
    }  
}  
  
class Complex {  
    int i;  
    public int Pow(int n, int x){  
        return 0;  
    }  
}
```

```
@.formatting.string = private constant [4 x i8] c"%d\0A\00"
%class.Matematica = type { %class.Complex, i32 }
%class.Teste = type { }
%class.Complex = type { i32 }
define i32 @main() {
entry0:
...
}
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
%a_tmp = alloca i32
store i32 %a, i32 * %a_tmp
%b_tmp = alloca i32
store i32 %b, i32 * %b_tmp
%tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0
%tmp1 = call i32 @_Pow_Complex(%class.Complex * %tmp0, i32 1, i32 2)
ret i32 %tmp1
}
define i32 @_Pow_Complex(%class.Complex * %this, i32 %n, i32 %x) {
entry0:
%n_tmp = alloca i32
store i32 %n, i32 * %n_tmp
%x_tmp = alloca i32
store i32 %x, i32 * %x_tmp
ret i32 0
}
declare i32 @printf (i8 *, ...)
declare i8 * @malloc (i32)
```

Herança: o que acontece neste código?

```
class Teste{
    public static void main(String[] a){
        System.out.println(new Zoo().Start());
    }
}
class Zoo {
    public int Start(){
        Tiger t1;
        t1 = new Tiger();
        return this.getInfo(t1);
    }
    public int getInfo(Animal c){
        return c.getWeight();
    }
}
class Tiger extends Animal{
    public int getWeight(){
        return 20;
    }
}
class Animal{
    public int getWeight(){
        return 0;
    }
}
```

Imprime 0 ou 20?

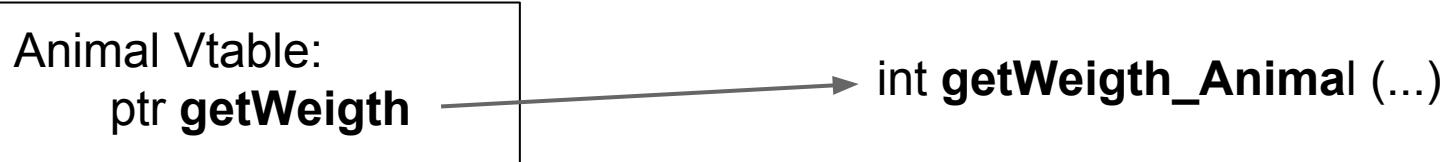
E em C++?

Herança: métodos virtuais

- Em Java: 20
- Em C++: 0
- Em C++ mudando o método **getWeigth** em **Animal** para virtual: 20
- Apesar de Java não ter métodos virtuais, quando há métodos sobrescritos em classes parentes, Java se comporta como C++ com métodos virtuais.
- **Como resolver isso em LLVM-IR?**

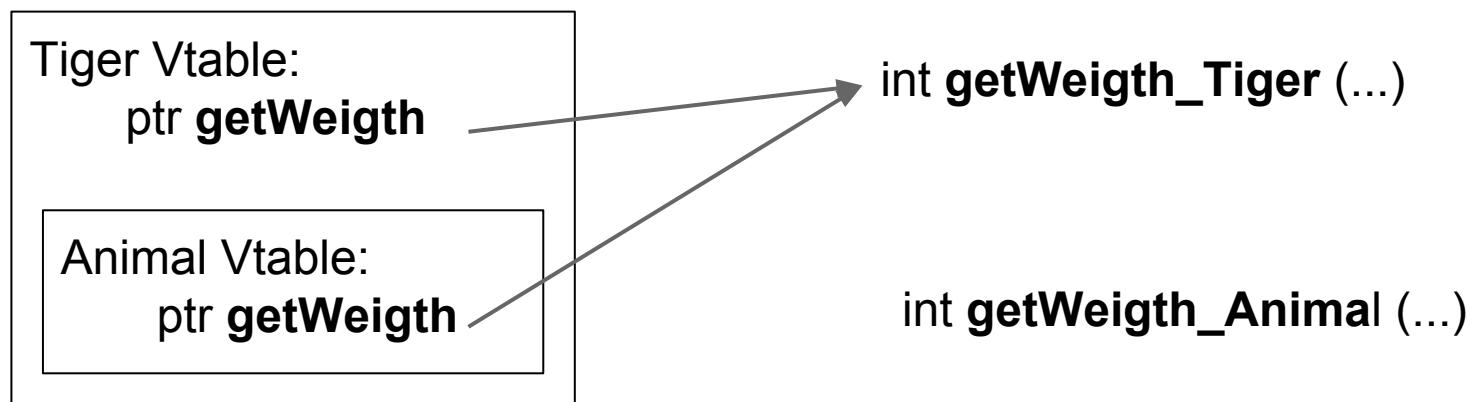
Virtual Method Table: Vtable

- Objetos possuem uma tabela com ponteiros para as funções



Virtual Method Table: Vtable em LLVM-IR

- **Lembrem:** Primeiro campo de uma classe filha é um objeto da classe pai
- O construtor do Objeto liga corretamente os ponteiros as funções



Virtual Method Table: Vtable

- LLVM-IR não tem suporte à classe, mas é possível implementar, como já vimos.
- LLVM-IR também não suporta Vtable, mas também é possível implementar.
- C também não suporta classe nem Vtable, mas é possível implementar ambos.
 - Exemplos interessantes: <http://milotshala.wordpress.com/2012/02/21/virtual-functions-in-c/>

Código com dois métodos virtuais

```
class Teste{
    public static void main(String[] a){
        System.out.println(new Zoo().Start());
    }
}
class Zoo {
    public int Start(){
        Tiger t1;
        int a;
        int b;
        t1 = new Tiger();
        a = this.getInfo(t1);
        b = this.getInfo(t1);
        return a+b;
    }
    public int getInfo(Animal c){
        return c.getWeight();
    }
}
```

```
class Tiger extends Animal{
    public int getWeight() {
        return 20;
    }
    public int getWeight2() {
        return 21;
    }
}
class Animal{
    public int getWeight() {
        return 0;
    }
    public int getWeight2() {
        return 1;
    }
}
```

VTable implementado com Array Estático

- Podemos fazer um Array de ponteiros genéricos (`i8*`), na primeira posição da estrutura da classe. O tamanho do Array é a quantidade de métodos desta classe
- No exemplo dado, todas as classes possuem dois métodos

```
%class.Tiger = type { [2 x i8 *], %class.Animal }
%class.Animal = type { [2 x i8 *] }
%class.Zoo = type { [2 x i8 *] }
%class.Teste = type { }
```

Construtor de Tiger

```
define void @_Tiger_Tiger(%class.Tiger * %this) {
entry0:
; Encontra a classe do Pai
%tmp0 = bitcast %class.Tiger * %this to %class.Animal *

; Construtor do pai (neste caso, sem implementação)
call void @_Animal_Animal(%class.Animal * %tmp0)

; Encontra a VTable na Classe do Pai
%tmp1 = bitcast %class.Animal * %tmp0 to [2 x i8 *] *

; Transforma a função em um ponteiro genérico i8*
%tmp2 = bitcast i32 (%class.Tiger *)* @_getWeight_Tiger to i8 *

; Carrega o primeiro elemento da Tabela, que no caso é o primeiro Método (getWeigth)
%tmp3 = getelementptr [2 x i8 *] * %tmp1, i32 0, i32 0

; Armazena o ponteiro genérico na Tabela
store i8 * %tmp2, i8 * * %tmp3

; Repete para o outro método, alterando o índice para o segundo elemento da Tabela
%tmp4 = bitcast i32 (%class.Tiger *)* @_getWeight2_Tiger to i8 *
%tmp5 = getelementptr [2 x i8 *] * %tmp1, i32 0, i32 1
store i8 * %tmp4, i8 * * %tmp5
ret void
}
```

Chamando um método virtual

```
define i32 @_getInfo_Zoo(%class.Zoo * %this, %class.Animal * %c) {
entry0:
%c_tmp = alloca %class.Animal *
store %class.Animal * %c, %class.Animal * %c_tmp
%tmp0 = load %class.Animal * %c_tmp

; Encontra a tabela de simbolos
%tmp1 = bitcast %class.Animal * %tmp0 to [2 x i8 *] *

; Retorna o ponteiro de funcao de acordo com o índice (neste caso é 0 (getWeigth))
%tmp2 = getelementptr [2 x i8 *] * %tmp1, i32 0, i32 0

; Corrigi o tipo com bitcast
%tmp3 = bitcast i8 * %tmp2 to i32 (%class.Animal *)*
; Carrega o ponteiro da memoria
%tmp4 = load i32 (%class.Animal *)* %tmp3

; Chama o Call
%tmp5 = call i32 %tmp4(%class.Animal * %tmp0)
ret i32 %tmp5
}
```

Código completo (1/4)

```
@.formatting.string = private constant [4 x i8] c"%d\\0A\\00"
%class.Tiger = type { [2 x i8 *], %class.Animal }
%class.Animal = type { [2 x i8 *] }
%class.Zoo = type { [2 x i8 *] }
%class.Teste = type { }
define i32 @main() {
entry0:
%tmp0 = alloca i32
store i32 0, i32 * %tmp0
%tmp2 = call i8* @malloc( i32 16)
%tmp1 = bitcast i8* %tmp2 to %class.Zoo *
call void @_Zoo_Zoo(%class.Zoo * %tmp1)
%tmp3 = call i32 @_Start_Zoo(%class.Zoo * %tmp1)
%tmp4 = getelementptr [4 x i8] * @.formatting.string, i32 0, i32 0
%tmp5 = call i32 (i8 *, ...)* @printf(i8 * %tmp4, i32 %tmp3)
%tmp6 = load i32 * %tmp0
ret i32 %tmp6
}
```

Código completo (2/4)

```
define i32 @_Start_Zoo(%class.Zoo * %this) {
entry0:
%t1 = alloca %class.Tiger *
%a = alloca i32
%b = alloca i32
%tmp1 = call i8* @malloc ( i32 32)
%tmp0 = bitcast i8* %tmp1 to %class.Tiger *
call void @_Tiger_Tiger(%class.Tiger * %tmp0)
store %class.Tiger * %tmp0, %class.Tiger ** %t1
%tmp2 = bitcast %class.Zoo * %this to %class.Zoo *
%tmp3 = load %class.Tiger ** %t1
%tmp4 = bitcast %class.Tiger * %tmp3 to %class.Animal *
%tmp5 = call i32 @_ getInfo_Zoo(%class.Zoo * %tmp2, %class.Animal * %tmp4)
store i32 %tmp5, i32 * %a
%tmp6 = bitcast %class.Zoo * %this to %class.Zoo *
%tmp7 = load %class.Tiger ** %t1
%tmp8 = bitcast %class.Tiger * %tmp7 to %class.Animal *
%tmp9 = call i32 @_ getInfo_Zoo(%class.Zoo * %tmp6, %class.Animal * %tmp8)
store i32 %tmp9, i32 * %b
%tmp10 = load i32 * %a
%tmp11 = load i32 * %b
%tmp12 = add i32 %tmp10, %tmp11
ret i32 %tmp12
}
```

Código completo (3/4)

```
define i32 @_ getInfo_Zoo(%class.Zoo * %this, %class.Animal * %c) {
entry0:
    %c_tmp = alloca %class.Animal *
    store %class.Animal * %c, %class.Animal * %c_tmp
    %tmp0 = load %class.Animal ** %c_tmp
    %tmp1 = bitcast %class.Animal * %tmp0 to [2 x i8 *] *
    %tmp2 = getelementptr [2 x i8 *] * %tmp1, i32 0, i32 0
    %tmp3 = bitcast i8 ** %tmp2 to i32 (%class.Animal *)*
    %tmp4 = load i32 (%class.Animal *)* %tmp3
    %tmp5 = call i32 %tmp4(%class.Animal * %tmp0)
    ret i32 %tmp5
}
define i32 @_ getWeight_Tiger(%class.Tiger * %this) {
entry0:
    ret i32 20
}
define i32 @_ getWeight2_Tiger(%class.Tiger * %this) {
entry0:
    ret i32 21
}
define i32 @_ getWeight_Animal(%class.Animal * %this) {
entry0:
    ret i32 0
}
define i32 @_ getWeight2_Animal(%class.Animal * %this) {
entry0:
    ret i32 1
}
```

Código completo (4/4)

```
define void @_Tiger_Tiger(%class.Tiger * %this) {
entry0:
    %tmp0 = bitcast %class.Tiger * %this to %class.Animal *
    call void @_Animal_Animal(%class.Animal * %tmp0)
    %tmp1 = bitcast %class.Animal * %tmp0 to [2 x i8 *] *
    %tmp2 = bitcast i32 (%class.Tiger *)* @_getWeight_Tiger to i8 *
    %tmp3 = getelementptr [2 x i8 *] * %tmp1, i32 0, i32 0
    store i8 * %tmp2, i8 * * %tmp3
    %tmp4 = bitcast i32 (%class.Tiger *)* @_getWeight2_Tiger to i8 *
    %tmp5 = getelementptr [2 x i8 *] * %tmp1, i32 0, i32 1
    store i8 * %tmp4, i8 * * %tmp5
    ret void
}
define void @_Animal_Animal(%class.Animal * %this) {
entry0:
    ret void
}
define void @_Zoo_Zoo(%class.Zoo * %this) {
entry0:
    ret void
}
define void @_Teste_Teste(%class.Teste * %this) {
entry0:
    ret void
}
declare i32 @printf (i8 *, ...)
declare i8 * @malloc (i32)
```

Para saber mais

- www.llvm.org
- <http://llvm.org/docs/LangRef.html>
- <http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR>
- Bruno Cardoso, Rafael Auler. "*Getting Started with LLVM Core Libraries*", 2014