Compilation Principle
编 译 原 理

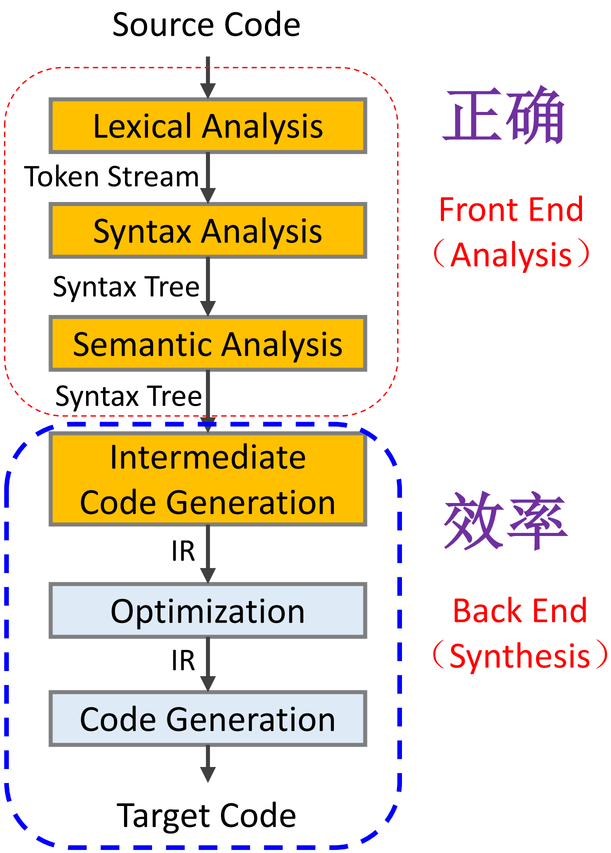
第16讲：中间代码(1)

张献伟

xianweiz.github.io

DCS290, 5/6/2021

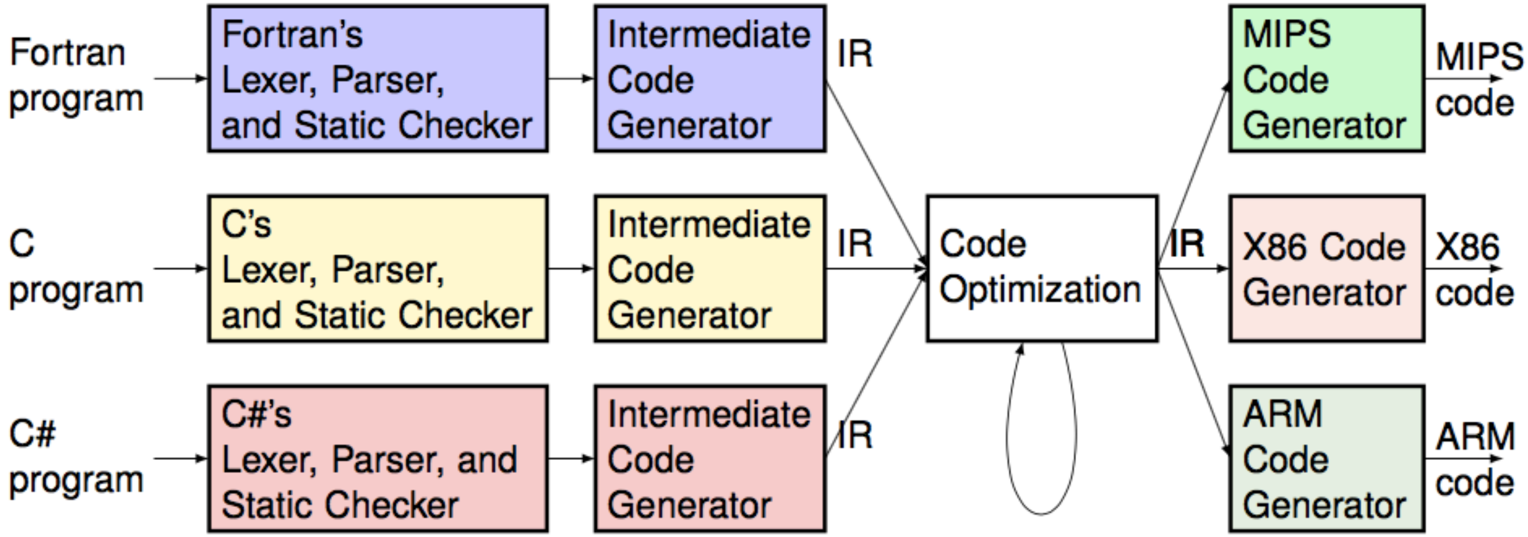
Compilation Phases[编译阶段]



- **Lexical**: source code → tokens
  - RE, NFA, DFA, …
  - Is the program lexically well-formed?
    - E.g., x#y = 1
- **Syntax**: tokens → AST or parse tree
  - CFG, LL(1), LALR(1), …
  - Is the input program syntactically well-formed?
    - E.g., x = 1 y = 2
- **Semantic**: AST → AST +symbol table
  - SDD, SDT, typing, scoping, …
  - Does the input program has a well-defined meaning?
    - E.g., int x; y = x(1)

Modern Compilers

- Compilation flow [编译流程]
  - First, translate the source program to some form of intermediate representation (IR, 中间表示)
  - Then convert from there into machine code
- IR provides advantages [IR的优势]
  - Increased abstraction, cleaner separation, and retargeting, etc



Different IRs for Different Stages

- Modern compilers use different IRs at different stages

- **High-Level** IR: close to high-level language
  - Examples: Abstract Syntax Tree, Parse Tree
  - Language dependent (a high-level IR for each language)
  - Purpose: semantic analysis of program
- **Low-Level** IR: close to assembly
  - Examples: Three address code[三地址码], Static Single Assignment[静态单赋值]
  - Essentially an instruction set[指令集] for an abstract machine
  - Language and machine independent (one common IR)
  - Purpose: compiler optimizations to make code efficient
    - All optimizations written in this IR is automatically applicable to all languages and machines

Different IRs for Different Stages (cont.)

- **Machine-Level** IR
  - Examples: x86 IR, ARM IR, MIPS IR
  - Actual instructions for a concrete machine ISA
  - Machine dependent (a machine-level IR for each ISA)
  - Purpose: code generation / CPU register allocation
    - (Optional) Machine-level optimizations (e.g. strength reduction: x / 2 → x » 1)
- Possible to have one IR (AST) — some compilers do
  - Generate machine code from AST after semantic analysis
  - Makes sense if compilation time is the primary concern (e.g. JIT)
    - Skip the IR generation step

- So why have multiple IRs?

Why Multiple IRs?

- Why multiple IRs?
  - Better to have an appropriate IR for the task at hand [针对性]
    - Semantic analysis much easier with AST
    - Compiler optimizations much easier with low-level IR
    - Register allocation only possible with machine-level IR
  - Easier to add a new front-end (language) or back-end (ISA) [易于扩展]
    - Front-end: a new AST → low-level IR converter
    - Back-end: a new low-level IR → machine IR converter
    - Low-level IR acts as a bridge between multiple front-ends and back-ends, such that they can be reused
- If one IR (AST), and adding a new front-end ...
  - Reimplement all compiler optimizations for new AST
  - A new AST → machine code converter for each ISA
  - Same goes for adding a new back-end

Three-Address Code[三地址码]

- High-level assembly where each operation has **at most three** operands. Generic form is X = Y op Z [最多3个操作数]
  - where X, Y, Z can be variables, constants, or compiler-generated temporaries holding intermediate values
- Characteristics [特性]
  - Assembly code for an 'abstract machine'
  - Long expressions are converted to multiple instructions
  - Control flow statements are converted to jumps [控制流->跳转]
  - Machine independent
    - Operations are generic (not tailored to specific machine)
    - Function calls represented as generic call nodes
    - Uses symbolic names rather than register names (actual locations of symbols are yet to be determined)
- Design goal: for easier machine-independent optimization

Three-Address Code Example

- For example, x * y + x * y is translated to

  t1 = x * y ; t1, t2, t3 are temporary variables

  t2 = x * y

  t3 = t1 + t2

  – Can be generated through a depth-first traversal of AST
  – Internal nodes in AST are translated to temporary variables
- Notice: repetition of x * y [重复]
  – Can be later eliminated through a compiler optimization called <u>common subexpression elimination</u> (CSE): [通用子表达式消除]

    t1 = x * y

    t3 = t1 + t1

  – Using 3-address code rather than AST makes it:
    - Easier to spot opportunities (just find matching RHSs)
    - Easier to manipulate IR (AST is much more cumbersome)

Three-Address Statements

- Assignment statement [二元赋值]

  x = y op z

  where op is an arithmetic or logical operation (binary operation)
- Assignment statement [一元赋值]

  x = op y

  where op is an unary operation such as -, not, shift
- Copy statement [拷贝]

  x = y
- Unconditional jump statement [无条件跳转]

  goto L

  where L is label

Three-Address Statements (cont.)

- Conditional jump statement [条件跳转]

  if (x relop y) goto L

  where relop is a relational operator such as =,≠, >, < J

- Procedural call statement [过程调用]

  param x_1, ..., param x_n, call F_y, n

  As an example, foo(x_1, x_2, x_3) is translated to

  param x_1

  param x_2

  param x_3

  call foo, 3

- Procedural call return statement [过程调用返回]

  return y

  where y is the return value (if applicable)

Three-Address Statements (cont.)

- Indexed assignment statement [索引]

  x = y[i]

  or

  y[i] = x

  where x is a scalar variable and y is an array variable

- Address and pointer operation statement [地址和指针]

  x = & y ; a pointer x is set to address of y

  y = * x ; y is set to the value of location

  ; pointed to by pointer x

  *y = x ; location pointed to by y is assigned x

Example

```
i = 1
do {
   a[i] = x * 5;
   i ++;
} while (i <= 10);
```

Source program

$i = 1$

$L:\ t_1 = x * 5$

$t_2 = \&a$

$t_3 = \mathrm{sizeof(int)}$

$t_4 = t_3 * i$

$t_5 = t_2 + t_4$

$*t_5 = t_1$

$i = i + 1$

$\text{if } i <= 10 \text{ goto } L$

a[i]

Three-address code

Implementation of TAC

- 3 possible ways (and more)
  - quadruples [四元式]
  - triples [三元式]
  - indirect triples [间接三元式]
- Trade-offs between, space, speed, ease of manipulation
- Using quadruples [四元式]

  op arg1, arg2, result

  - There are four(4) fields at maximum
  - arg1 and arg2 are optional, depending on the *op*
  - Examples:
    - x = a + b => + a, b, x
    - x = - y => - y, , x
    - goto L => goto , , L

Using Triples[三元式]

- Triple: quadruple without the result field
  - Result field is implicitly index of instruction
  - Result referred to by index of instructions computing it
  - Example: a = b * (-c) + b * (-c)

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	-	c		t3	-	c	
(3)	*	b	t3	t4	*	b	(2)
(4)	+	t2	t4	t5	+	(1)	(3)
(5)	=	t5		a	=	a	(4)

More About Triples

- What if LHS of assignment is not a var but an expression?
  - Array location (e.g. x[i] = y)
  - Pointer location (e.g. *(x+i) = y)
  - Struct field location (e.g. x.i = y)
- Compute memory address of LHS location beforehand
- Example: triples for array assignment statement

```
x[i] = y
```

  - is translated to

```
(0): [] x i          // Compute address of x[i] location
(1): = (0) y         // Assign y to that location
```

  - Complex LHS may require more triples to compute address

Using Indirect Triples[间接三元式]

- Problem with triples
  - Compiler optimizations often involve moving instructions
  - Hard to move instructions because numbering will change, even for instructions not involved in optimization
  - See below CSE performed on the second (-c) * b:

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
~~(2)~~	~~-~~	~~c~~		~~t3~~	~~-~~	~~c~~	
~~(3)~~	~~*~~	~~b~~	~~t3~~	~~t4~~	~~*~~	~~b~~	~~(2)~~
~~(4)~~(2)	+	t2	~~t4~~ t2	t5	+	(1)	~~(3)~~(1)
~~(5)~~(3)	=	t5		a	=	a	(4)

Using Indirect Triples[间接三元式]

- Problem with triples
  - Compiler optimizations often involve moving instructions
  - Hard to move instructions because numbering will change, even for instructions not involved in optimization
  - See below CSE performed on second (-c) * b:

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	+	t2	t2	t5	+	(1)	(1)
(3)	=	t5		a	=	a	(4)

Instruction (3) refers to (4) which is no longer there.

Using Indirect Triples (cont.)

- Triples are stored in a triple 'database'
- IR is a listing of pointers to triples in database
  - Can reorder listing without changing numbering in database
- Pointer indirection overhead but allows easy code motion

	Listing
	(ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(2)
(3)	(3)
(4)	(4)
(5)	(5)

	Database		
	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

After CSE Optimization

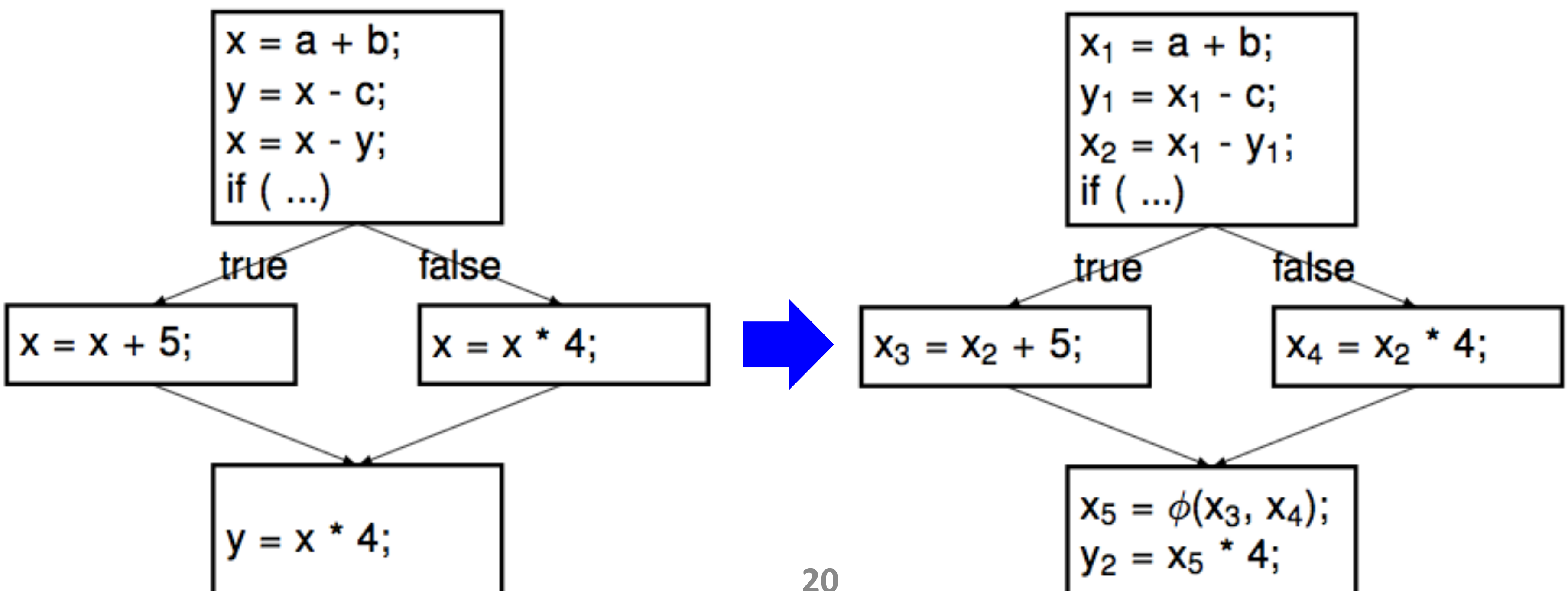
- After CSE, empty entries in database can be reused
  - Code in triple database becomes non-contiguous over time
  - That's fine since the listing is the code, not the database

	Listing
	(ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(4)
(3)	(5)

	Database		
	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	empty		
(3)	empty		
(4)	+	(1)	(1)
(5)	=	a	(4)

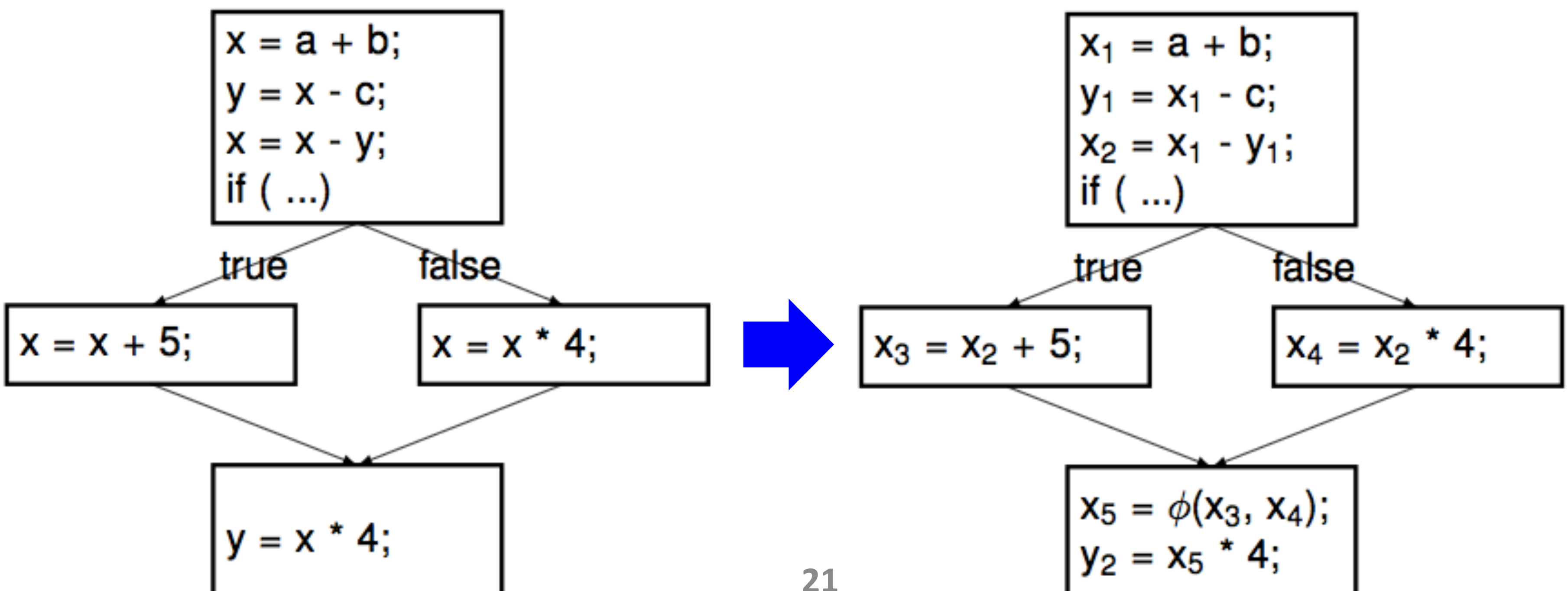
Single Static Assignment[静态单赋值]

- Every variable is assigned to exactly once statically
  - Give variable different version name on every assignment
    - e.g. $x \rightarrow x_1, x_2, \ldots, x_5$ for each static assignment of x
  - Now value of each variable guaranteed not to change
  - On a control flow merge, ϕ-function combines two versions
    - e.g. $x_5 = \phi(x_3, x_4)$: means x_5 is either x_3 or x_4



Benefits of SSA

- SSA is an IR that facilitates certain code optimizations
  - SSA tells you when an optimization shouldn't happen
  - Suppose compiler performs CSE on previous example:
    - Without SSA, (incorrectly) tempted to eliminate second x * 4
    - With SSA, x_2 * 4 and x_5 * 4 are clearly different values



Benefits of SSA (cont.)

- SSA is an IR that facilitates certain code optimizations
  - SSA tells you when an optimization should happen
  - Suppose compiler performs dead code elimination (DCE): (DCE removes code that computes dead values)

```
x = a + b;
x = c - d;
y = x * b;
```

$\Rightarrow$

$x_1 = a + b;$
$x_2 = c - d;$
$y_1 = x_2 * b;$

  - Without SSA, not clear whether there are dead values
  - With SSA, x_1 is never used and clearly a dead value
- Why does SSA work so well with compiler optimizations?
  - SSA makes flow of values explicit in the IR
  - Without SSA, need a separate dataflow graph
  - Will discuss more in **Compiler Optimization** section

SSA Orthogonal to IR Implementation

- SSA is expressed most commonly as 3-address code
- We learned 3 ways to implement 3-address code
  - quadruples
  - triples
  - indirect triples
- How you implement is orthogonal to SSA representation
  - After variable renaming, any 3-address code becomes SSA
- SSA is used widely in modern compilers:
  - GCC (GNU C Compiler)
  - LLVM (Low Level Virtual Machine) Compiler
  - Oracle Java JIT Compiler
  - Google Chrome JavaScript JIT Compiler
  - PyPy Python JIT Compiler

Generating Code

using Syntax Directed Translation

Syntax Directed Translation[语法制导翻译]

- Syntax directed translation used again for code generation
  - Since code generation is also dependent on syntax
  - Code generation is translating syntactic structures to code
- What language structures do we need to translate?
  - Definitions (variables, functions, ...)
  - Assignment statements
  - Control flow statements (if-then-else, for-loop, ...)
  - ...
- We are going to use the following strategy:
  - Specify SDD semantic rules (without ordering)
  - Convert SDD rules to SDT actions (with ordering)
    - In the process, we will discover SDD has non-*L-attributes*
    - We will also discuss what to do with those non-*L-attributes*

Code Generation Overview[代码生成]

- Program code is a collection of functions
  - By now, all functions are listed in symbol table

- Goal is to generate code for each function in that list
- Generating code for a function involves two steps:
  - Processing variable definitions [变量定义]
    - Involves laying out variables in memory
  - Processing statements [语句]
    - Involves generating instructions for statements

- We will start with <u>processing variable definitions</u>

Processing Variable Definitions

- To lay out a variable, both location and width are needed
  - Location: where variable is located in memory
  - Width: how much space variable takes up in memory
- Attributes for variable definition:
  - **T V** e.g. int x;
  - **T**: non-terminal for type name
    - **T.type**: type (int, float, ...)
    - **T.width**: width of type in bytes (e.g. 4 for int)
  - **V**: non-terminal for variable name
    - **V.type**: type (int, float, ...)
    - **V.width**: width of variable according to type
    - **V.offset**: offset of variable in memory
  - But offset from what...?

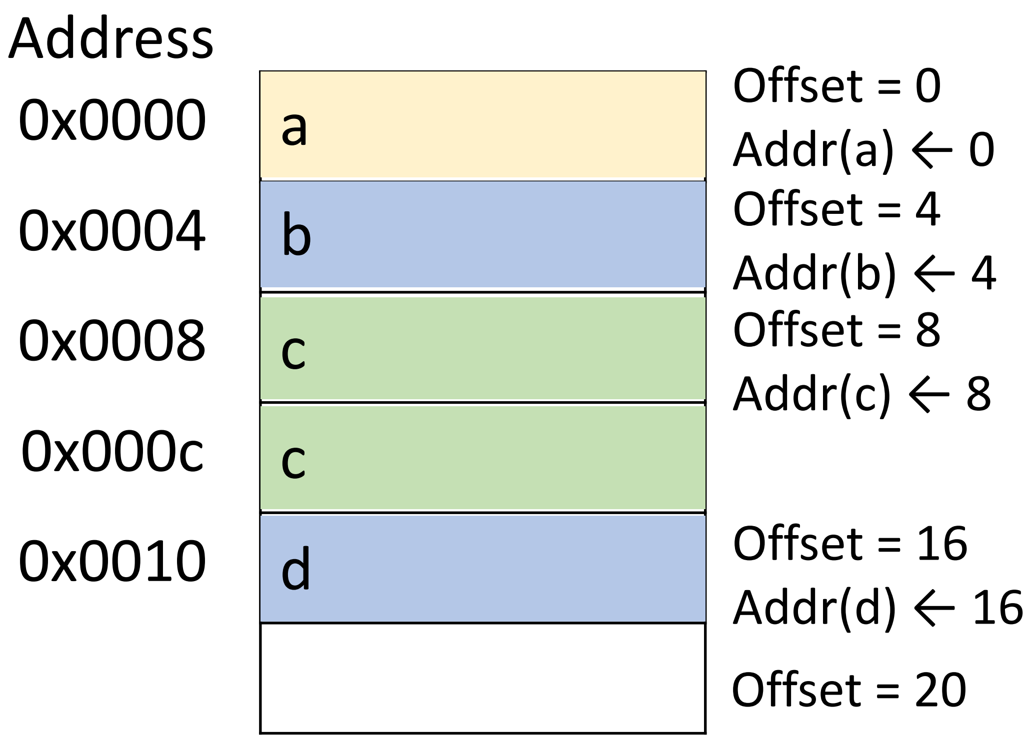
Calculate Variable Location from Offset

- Naive method: reserve a big memory section for all data
  - Size data section to be large enough to contain all variables
  - Location = var offset + base of data section
- Naive method wastes a lot of memory
  - Vars with limited scope need to live only briefly in memory
    - E.g. function variables need to last only for duration of call
- **Solution**: allocate memory briefly for each scope
  - Allocate when entering scope, free when exiting scope
  - Variables in same scope are allocated / freed together
  - Location = var offset + base of scope memory section
  - Will discuss more later in **Runtime Management**

Storage Layout of Variables in a Function

- When there are multiple variables defined in a function,
  - Compiler lays out variables in memory <u>sequentially</u>
  - Current <u>offset</u> used to place variable x in memory
    - address(x) ← offset
    - offset += sizeof(x.type)

```
void foo() {
  int a;
  int b;
  long long c;
  int d;
}
```



More about Storage Layout

- Allocation alignment[对齐]
  - Enforce addr(x) % sizeof(x.type) == 0
  - Most machine architectures are designed such that computation is most efficient at sizeof(x.type) boundaries
    - E.g. Most machines are designed to load integer values at integer word boundaries
    - If not on word boundary, need to load two words and shift & concatenate → inefficient

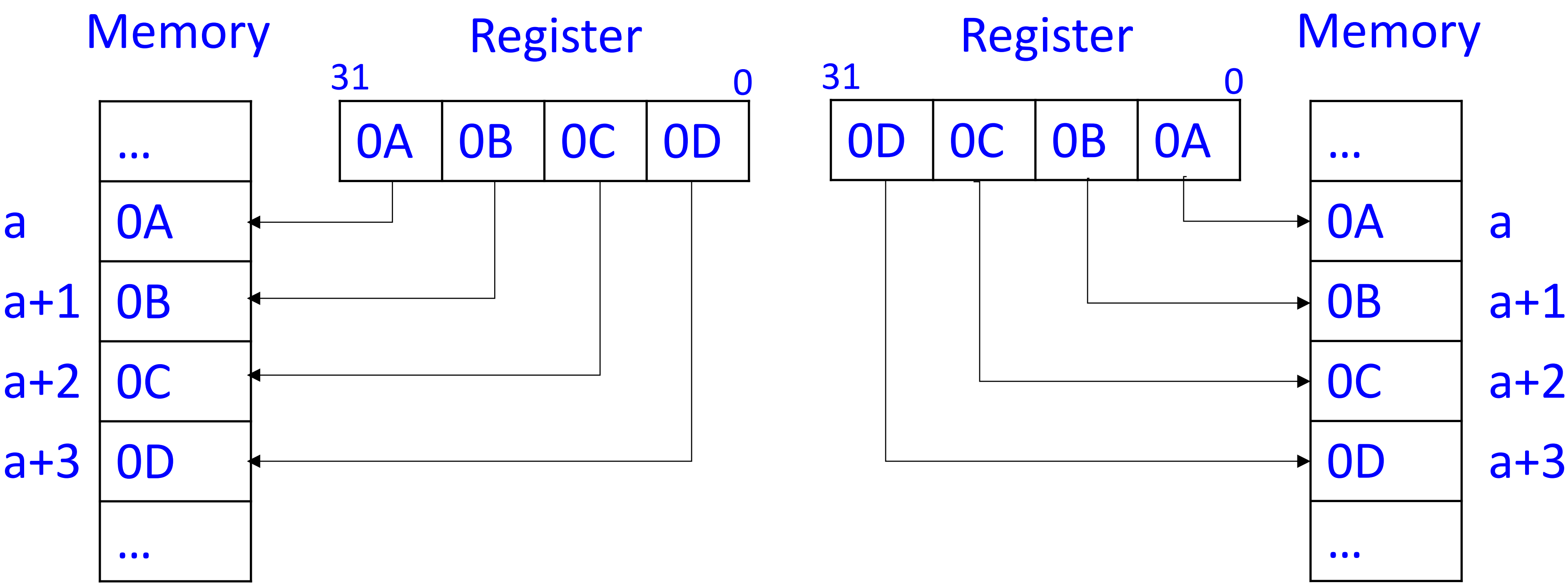
```
void foo() {
  char a;        // addr(a) = 0
  int b;         // addr(b) = 1
  int c;         // addr(c) = 5
  long long d; // addr(d) = 9
}
```

→

```
void foo() {
  char a;        // addr(a) = 0
  int b;         // addr(b) = 4
  int c;         // addr(c) = 8
  long long d; // addr(d) = 16
}
```

More about Storage Layout (cont.)

- Endianness[字节序]
  - Big endian: MSB (most significant byte) in lowest address
  - Little endian: LSB (least significant byte) in lowest address



Big-endian [大字节序]

Little-endian [小字节序]

https://www.geeksforgeeks.org/little-and-big-endian-mystery/