a = 1;
b = 2;
c = a + b;

a = 1;
b = 2;
c = 3;
Alias Analysis Motivation

```c
int a = 1;
int b = 2;
int x = 4;
int c = a + b;
```

If `x` must equal `&a`, `c = 6`.
If `x` must equal `&b`, `c = 5`.
If not (`x` may equal `&a` or `x` may equal `&b`), `c = 3`. 

```c
int a = 1;
int b = 2;
int x = 4;
int c = ???;
```
If \( x \) must equal \&a, \( c = 6 \).
If \( x \) must equal \&b, \( c = 5 \).
If not (\( x \) may equal \&a or \( x \) may equal \&b), \( c = 3 \).
1. Call by reference

```c
int f(var x, var y) {
    x = 1;
    y = 2;
    return x + y;
}
```

```c
int z = 1;
z = f(z, z);
```
1. Call by reference

```c
int f(var x, var y) {
    x = 1;
    y = 2;
    return x + y;
}

int z = 1;
z = f(z, z);
```
Sources of Aliases

2. Address-of operator

```c
int a;
int* x = &a;
```

3. Dynamic memory allocation

```c
int* x = malloc(sizeof(int));
```

4. Array expressions

```c
a[x] = 1;
a[y] = 2;
c = a[x]; // is c = 1 or 2?
```
Address Taken

\[
\begin{align*}
  &a = 1; \\
  &b = 2; \\
  &*x = 4; \\
  &c = a + b;
\end{align*}
\]

IF \&a, \&b never occur in the program, THEN x can never equal \&a or \&b. (except for evil pointer arithmetic)

Local variables:
in Java, address cannot be taken
in C, quite rare to take their address
IF the language enforces declared types, THEN $x$ can never equal $&a$ or $&b$. 

double* x;
int a = 1;
int b = 2;
*x = 4;
c = a + b;
Alias pairs

$$(\alpha, \beta)$$ if the expressions $\alpha$ and $\beta$ must/may point to the same location

Points-to sets

$$o \in pt(p)$$ if $p$ must/may point to $o$
Flow-sensitive vs. Flow-insensitive

**Flow-sensitive**
- Compute separate analysis information at each program point.
- Consider order in which statements execute.
- Allow strong updates.

**Flow-insensitive**
- Compute single result for whole program (which holds at every program point).
- Ignore statement execution order.
- Only weak updates allowed.
Subset-based vs. Equality-based

Assignment statement: \( x = y \)

Subset-based aka Propagation-based aka Andersen

\[ pt(y) \subseteq pt(x) \]

\( x \) points to everything that \( y \) points to. \( O(n^3) \) in theory.

Equality-based aka Unification-based aka Steensgaard

\[ pt(y) = pt(x) \]

\( x \) and \( y \) point to the same set of objects. \( O(n^{\alpha(n)}) \) but less precise.
Equality-based points-to analysis

Let $S = \{s_1, s_2, s_3, \ldots\}$ be the set of all possible targets of pointers. We partition it into disjoint subsets, and model each pointer to point to one of the disjoint subsets.

**Example**

\[
\begin{align*}
p &= &\&x; \\
q &= &\&y; \\
p &= q; \\
r &= &\&z;
\end{align*}
\]

![Diagram](image1.png)

**Example**

\[
\begin{align*}
a &= &\&b; \\
b &= &\&c; \\
a &= &\&d; \\
d &= &\&e;
\end{align*}
\]

![Diagram](image2.png)
<table>
<thead>
<tr>
<th></th>
<th>Field Sensitive</th>
<th>Field Insensitive</th>
<th>Field Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstraction</strong></td>
<td>o.f</td>
<td>o.*</td>
<td>*.f</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>slowest</td>
<td>medium</td>
<td>fastest single iteration</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>most precise</td>
<td>very imprecise for OO-style code</td>
<td>medium</td>
</tr>
<tr>
<td><strong>Soundness</strong></td>
<td></td>
<td></td>
<td>only if field types enforced</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>Java, C</td>
<td>C</td>
<td>Java</td>
</tr>
</tbody>
</table>