Example: Build a Stack

1 int h, i;
2
3 void B (int w) {
4   int j, k;
5   i = 2 * w;
6
7   if (i > 0) {
8      int i = 5;
9      w = w + i;
10   }
11
12   w = w + i;
13 }
14
15 void A (int x, int y) {
16   float i, j;
17   B (h);
18   i = 3;
19 }
20
21 int main () {
22   int a, b;
23   h = 5;
24   a = 3;
25   b = 2;
26   A (a, b);
27   B (h);
28 }

The stack of symbol tables of Function B, A and main?
Example: References

Where are these three \( i \)s declaration?

```
1 int h, i;
2
3 void B (int w) {
4    int j, k;
5    i = 2 * w;
6
7    if (i > 0) {
8       int i = 5;
9       w = w + i;
10  }
11
12  w = w + i;
13 }
14
15 void A (int x, int y) {
16    float i, j;
17    B (h);
18    i = 3;
19 }
20
21 int main () {
22    int a, b;
23    h = 5;
24    a = 3;
25    b = 2;
26    A (a, b);
27    B (h);
28 }
```
Revisit the Example

```c
int h, i;
void B (int w) {
    int j, k;
i = 2 * w;
    if (i > 0) {
        int i = 5;
w = w + i;
    }
w = w + i;
}
void A (int x, int y) {
    float i, j;
    B (h);
i = 3;
}
int main () {
    int a, b;
h = 5;
a = 3;
b = 2;
A (a, b);
B (h);
}
```

- i in line 9 is a reference of the i declared in line 8
- the declaration of i in line 8 hides the declaration of i in line 1
Visibility

- A name is *visible* if it is not *hidden* or shadowed by another identifier with the same name declared in a nested scope.

- In some languages, the hidden identifiers can be *accessed via* using the *appropriate prefixes*.

```java
public class Cell {
    private int row;
    private int col;
    private boolean alive;

    public Cell (int row, int col, boolean alive) {
        this.row = row;
        this.col = col;
        this.alive = alive;
    }

    public void setPosition(int row, int col) {
        this.row = row;
        this.col = col;
    }

    public void setAlive(boolean alive) {
        this.alive = alive;
    }

    // ...
}
```

*this*: differentiates a class field from a local variable or parameter.
Overloading

- *Overloading* is the use of *same name* to *describe different functionalities*

- Overloading applies to *functions* and *operators*

- Using the *number or type of arguments* to distinguish among identical function names or operator
  
  - **Signature**: usually includes the *function name, number, type and order of parameters*

- *Overloading ≠ Overriding*
  
  - Overriding: redefine a method that has been defined in a parent class (*exactly the same signature*)
  
  - Overloading is resolved at *compile time*, overriding is resolved at *run time*
Lifetime

The **lifetime** of a variable is the **time interval** during which the **variable has been allocated a block of memory**

**Static allocation:**

- All variables have a lifetime that are the complete execution time (Fortran, Cobol)
- Dynamic memory management has to be handled by programmers

**Dynamic allocation:**

- **Lifetime is linked with the scope** of variables
- Methods to specify lifetime are different in different languages
Example: static in C

#include <stdio.h>

void foo () {
    int i = 10;
    static int si = 10;
    i += 10;
    si += 10;
    printf("i = %3d, si = %3d \n", i, si);
}

int main () {
    int i = 0;
    for (i = 0; i < 10; i++)
        foo();
}