Names

- Names: identify language entities
  - variables, procedures, functions, constants, data types, ...
- Attributes: properties of names
- Examples of attributes:
  - A data type
    ```java
    int n = 5;
    (data type name: int)
    ```
  - A value
    ```java
    (value: 5)
    ```
  - A location
    ```java
    int* y;
    y = new int;
    ```
  - Function parameters, the type of return value
    ```java
    int f(int n) {...}
    ```

Binding

- Binding: associating attributes to names
  - declarations
  - assignments
  - declarations (prototype) and definition of a function

- The bindings can be explicit or implicit
  e.g. `int x;`
  - Explicit binding: the data type of `x`
  - Implicit binding: the location of `x` (static or dynamic, depending on where `x` is stored)

Binding Time

- Binding Time: the time when an attribute is bound to a name
  - **Static binding** (static attribute):
    occurs before execution
    - **Language definition/implementation time**: The range of the data type `int`
    - **translation time (parsing/semantic analysis)**: The data type of a variable
    - **link time**: The code of an external function
    - **load time**: The location of a global variable
  - **Dynamic binding** (dynamic attribute):
    occurs during execution
    - **Between the entry and exit points of a procedure or program**: the value of local variables
Where can declarations happen?

- Blocks ({}), begin-end, …
  - Algol descendants: C/C++, Java, Pascal, Ada, ...
e.g., C
  - Function body
  - Anywhere a statement can appear (compound statement)
- External/global
- Structured data type
- Class

C++ Example

```cpp
const int Maximum = 100;
struct FullName { string LastName, string FirstName; };

class Student {
    private:
        struct FullName name; int Age;
    public:
        void setValue(const int &i, const FullName &name) {
            int i;
            i = i + 1;
            name.LastName = lname;
            name.FirstName = FName;
        }
    }

void Student::setAge(const int a, string lName, string fName) {
    int i;
    Age = a;
    int j;
    name.LastName = lName;
    name.FirstName = fName;
}
```

Scope of Binding

- **Scope of Binding**: the region of the program text where the binding applies (is valid)

- **Block-structured language**
  
  *lexical scope (static scope)*: from the declaration to the end of the block containing the declaration
  
  *dynamic scope*: introduced later

Example

```cpp
int x;
void p(void) {
    char y;
    . .
    [ int i;
      . .
    ]
}

void q(void) {
    . .
    [ double x;
      . .
    ]

main() {
    int w[10];
    . .
}
```
Declaration before Use

```c
void p(void) {
    int x;
    
    char y;
    
}
```

Exception in OO languages: Scope of local declarations inside a class declaration includes the whole class

```java
public class {
    public int getValue() { return value; }
    
    int value;
}
```

Scope Hole

- **Scope Hole**: Declarations in nested blocks take precedence over the previous declarations. That is, the binding becomes invisible/hidden

```c
int x;

void p(void) {
    char x;
    x = 'a';
    
    main() {
        x = 2;
        
    }
}
```

Accessing Hidden Declarations

- **Scope resolution operator :: (C++)**

```c
int x;

void p(void) {
    char x;
    x = 'a';
    ::x=42;
    
} main() {
    x = 2;
    
}
```

Hide a Declaration

- **File 1:**

```c
extern int x;
int x;
```

- **File 2:**

```c
extern x;
static int x;
```
Symbol Table

- Symbol Table: maintains bindings
  - Can be viewed as functions that map names to their attributes

Static vs. Dynamic Scope

- Static scope (lexical scope):
  - The scope is maintained statically (during compilation)
  - It follows the layout of the source code
  - Used in most languages
- Dynamic scope:
  - The scope is maintained dynamically (during execution)
  - It follows the execution path
  - Few languages use it
    - The bindings cannot be determined statically
    - The result may depend on the user input
      - Lisp: considered a bug by its inventor
      - Perl: can choose lexical or dynamic scope

Static Scope

```c
int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%e\n",y);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}
```

The symbol table in p:
the bindings available in p

```c
int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%e\n",y);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}
```
### Static Scope

**Example 1:**

```c
int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%c\n",y);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}
```

**Symbol Table in `q`:**
- `integer, global`
- `integer, local to q`
- `character, global`

**Symbol Table in `main`:**
- `character, local to main`
- `integer, global`

### Static Scope

- The symbol table is built during compilation.
- The bindings are used in type-checking and in generating the machine code.
- Result:
  
- E.g., semantics of `q`

```c
void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}
```

- The symbol table in `q`:
- `integer, global`
- `integer, local to q`
- `character, global`

### Practice for Static Scope

**Example 2:**

```c
int x,y;

void q(void) {
    x = x + 1;
    y = x + 1;
}

void f(void) {
    int x;
    y = y + 1;
    x = y + 1;
    g();
}

main() {
    x = 1;
    y = 2;
    f();
    g();
    printf("x=%d,y=%d\n",x,y);
}
```

**Exercise:**

Draw the symbol table at the given points in the program using static scope.

**Question:**

What does the program print using static scope?
What if dynamic scope is used?

int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%d\n",x);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}

The symbol table in main:
the bindings available in main

character, 'b', local to main
integer, 1, global

What if dynamic scope is used?

The symbol table in q:
the bindings available in q

character, 'b', local to main
integer, 1, global

What if dynamic scope is used?

int x = 1;
char y = 'a';

void p(void) {
    double x=2.5;
    printf("%d\n",x);
}

void q(void) {
    int y = 42;
    printf("%d\n",x);
    p();
}

main() {
    char x = 'b';
    q();
}

The symbol table in p:
the bindings available in p

double, 2.5, local to p
character, 'b', local to main
integer, 1, global

What if dynamic scope is used?
Practice for Dynamic Scope

```c
int x, y;
void g(void) {
    x = x + 1;
}
Point 1
void f(void) {
    int x;
    y = y + 1;
    g();
}
Point 2
main() {
    x = 1;
    y = 2;
    f();
    printf("x=%d, y=%d\n", x, y);
}
Point 3
```

Exercise:
Draw the symbol table at the given points in the program using dynamic scope.

Question:
What does the program print using dynamic scope?

Overloading

- What is overloading?
  - Using the same name for more than one entities
- Why overloading?
  - Convenience
  - Optional function parameters
  - Polymorphism
    - e.g. we want + to work on both integers and reals
- What can be overloaded?

Overload Resolution

- Overload Resolution
  - select one entity among all qualified
- Name isn’t sufficient in resolution
  - need extra information (often data types)

Function/Method Overloading

- C: no overloading
- C++/Java/Ada: resolution by considering the number and types of parameters
  - Perfect if exact match exists
  - No perfect match: different conversion rules
    - Ada: automatic conversions not allowed
    - Java: conversions allowed in certain cases
    - C++: automatic conversions; more flexible
    - e.g.,
      - int sum(int a, int b) {...}
      - double sum(double a, double b) {...}
      - double sum(double a, int b) {...}
      - sum(1); sum(1.0); sum(1.0, 2.0); sum(1.0, 2.0);
### Overload Resolution Example

1. `int sum(int, int);`
2. `double sum(double, int);`
3. `double sum(double, double);`

```plaintext
int x;
double y;
```

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Java</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = sum(3,4);</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>y = sum(3,4);</code></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>x = sum(3,4,5);</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>y = sum(3,4,5);</code></td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td><code>x = sum(3,5,4);</code></td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>y = sum(3,5,4);</code></td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><code>x = sum(3,5,4,5);</code></td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>y = sum(3,5,4,5);</code></td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```

### Stack-Based Allocations

- **static(global) area**
- **stack**
  - automatically-allocated spaces
  - under the control of the runtime system
- **(unallocated)**
- **heap**
  - manually-allocated spaces
  - under the control of the programmer

### Environment

- **Location**: one specific attribute of names
- **Environment**: maintains bindings from names to locations
- **Static vs. dynamic**
  - **FORTRAN**: completely static
  - **LISP**: completely dynamic
  - **Algol-descendants (C, C++, Ada, Java)**: combination of:
    - global variables: static
    - local variables: dynamic

### Example

```plaintext
A: { int x;
    char y;
    
B: { double x;
      int a;
      }
  
C: { char y;
      int b;
      
D: { int x;
      double y;
      }
  }
```
A: { int x;
    char y;
}
B: { double x;
    int a;
    ...
}
C: { char y;
    int b;
    ...
}
D: { int x;
    double y;
    ...
}

Example

A: { int x;
    char y;
}
B: { double x;
    int a;
    ...
}
C: { char y;
    int b;
    ...
}
D: { int x;
    double y;
    ...
}
### Heap-Based Allocation

- **C**
  ```c
  int *x;
  x=(int *)malloc(sizeof(int));
  free(x);
  ```
- **C++**
  ```cpp
  int *x;
  x = new int;
  delete x;
  ```
- **Java**
  ```java
  Integer x = new Integer(2);
  // no deletion
  // needs garbage collection
  ```

### Scope vs. Lifetime

- **Lifetime beyond scope:**
  - alive in scope hole
  - alive outside scope
- **Scope beyond lifetime (unsafe)**

---

### Example: Alive in scope hole

A: ```
{ int x;
  char y;
}
```  
B: ```
{ double x;
  int  a;
}
```  
C: ```
{ char y;
  int  b;
}
```  
D: ```
{ int  x;
  double y;
}
```  

### Example: Alive outside scope

```c
int func(void) {
    static int counter = 0;
    counter += 1;
    return counter;
}
```  
main() ```
{  
    int i;
    int x;
    for (i=0; i<10; i++)
        x = func();
    printf("%d\n", x);
}  ```
Example: Scope beyond lifetime

Dangling pointer:

```c
int *x, *y, *z;

x=(int *) malloc(sizeof(int));
*x=2;
y=x;
free(x);

...  
printf("%d\n",*y);
```

Box-and-Circle Diagram for Variables

Assignment by Sharing

Java:

```java
Student x = new Student("Amy");  
Student y = new Student("John");  
x.setAge(19);  
x = y;  
y.setAge(21);
```

Assignment by Cloning

```java
x = y
```
(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 1:

![Diagram 1]

(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 2:

![Diagram 2]

(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 3:

![Diagram 3]

(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 4:

![Diagram 4]
### Aliases

(1) int *x, *y;
(2) x = (int *)malloc(sizeof(int));
(3) *x = 1;
(4) y = x;
(5) *y = 2;
(6) printf("%d\n",*x);

After line 5:

![Diagram of variables x and y with pointers]

### Practice for Aliases

(1) #include <stdio.h>
(2) main()
(3) int **x;
(4) int *y;
(5) int z;
(6) x = (int**)malloc(sizeof(int));
(7) y = (int*)malloc(sizeof(int));
(8) z = 1;
(9) *y = 2;
(10) **x = y;
(11) ***x = z;
(12) printf("%d\n",*y);
(13) *z = 3;
(14) printf("%d\n",*y);
(15) **x = 4;
(16) printf("%d\n",z);
(17) return 0;
(18)

Exercise:
Draw box-and-circle diagrams of the variables after line 11 and 15

Question 1:
Which variables are aliases at each of those points?

Question 2:
What does the program print?

### Dangling References

```c
int *x, *y;
...
x = (int *)malloc(sizeof(int));
...  
x = 2;
...
y = x;
free(x);
/* *y is now a dangling reference */
...
printf("%d\n",*y); /*illegal reference*/
```

```c
{int *x;
    { int y;
        y = 2;
        x = &y;
    }
    /* *x is now a dangling reference */
}
```
Dangling References

```c
int* dangle(void)
{
    int x;
    return &x;
}

...

y = dangle();
/* *y is a dangling reference */
```