Symbol Table

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

\[ \text{Symbol Table} \]
\[ \begin{array}{c|c}
\text{Names} & \text{Attributes} \\
\hline
\text{int x = 1;} & \text{double, local to p} \\
\text{char y = 'a';} & \text{character, global} \\
\text{void p(void) { } \text{double x=2.5; } \text{printf("%c\n", y);}} & \\
\text{void q(void) { } \text{int y = 42; } \text{printf("%d\n", x); } \text{p();}} & \\
\text{main()} { } \text{char x = 'b'; } \text{q();}} & \\
\end{array} \]

Static vs. Dynamic Scope

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

Static Scope

- The symbol table in p:
  - The bindings available in p

\[ \text{The symbol table in q:} \]
\[ \text{the bindings available in q} \]

- Static Scope
  - The symbol table in p:
    - The bindings available in p
  - Static Scope
    - The symbol table in q:
      - The bindings available in q
Static Scope

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

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

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

The symbol table in `main`:
- The bindings available in `main`
  - `x`: integer, local to `main`
  - `y`: character, global

Practice for Static Scope

```
int x,y;

void g(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);
}
```

Question 1:
Draw the symbol table at the given points in the program, using static scope?

Question 2:
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("%c\n",y);
}

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

The symbol table in `q`:
- The bindings available in `q`
  - `x`: integer, local to `main`
  - `y`: integer, 42, local to `q`

What if dynamic scope is used?

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

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

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

The symbol table in `p`:
- The bindings available in `p`
  - `x`: double, 2.5, local to `p`
  - `y`: integer, 1, global

What if dynamic scope is used?
Practice for Dynamic Scope

```c
int x, y;
void g(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);
}
```

Question 1:
Draw the symbol table at the given points in the program, using dynamic scope?

Point 1

Point 2

Point 3

Question 2:
What does the program print, using dynamic scope?

Overloading

• What is overloading?

• Why overloading?

• What can be overloaded?

Function/Method Overloading

• C: no overloading
• C++/Java/Ada: resolution by 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 directions.
  – C++: automatic conversions more flexible.
  – e.g., double sum(double a, int b) {…} double sum(int a, int b) {…} double sum(int a, double b) {…}

Overload Resolution

• Overload Resolution: select one entity.
• Name isn’t sufficient in resolution: need extra information (often data types)

Overload Resolution Example

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

```
int x;
double y;

C++ | Java | Ada
---|---|---
x = sum(3.4); | 1 | 1 | 1
y = sum(3.4); | 1 | 1 | 0
x = sum(3.4, 5.5); | 0 | 0 | 0
y = sum(3.4, 5.5); | 0 | 0 | 0
x = sum(3.4, 5.4); | 2 | 0 | 0
y = sum(3.5, 4.4); | 2 | 2 | 2
x = sum(3.5, 4.4); | 3 | 0 | 0
y = sum(3.5, 4.5); | 3 | 3 | 3
```

Environment

• Location: one specific attribute of names
• Environment: maintain bindings of names to locations
  – Static vs. dynamic
    – FORTRAN: completely static
    – LISP: completely dynamic
    – Algol-descendants (C, C++, Ada, Java): combination
      • global variables: static
      • local variables: dynamic
Stack-Based Allocations

- **Stack**
  - `static` (global) area
  - automatically-allocated spaces (local variables, procedures (chapter 8) under the control of runtime system

- **Heap**
  - manually-allocated spaces under the control of programmer

**Example**

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;
}

**Example**

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

- **C++**
  ```
  int *x;
  X = new int;
  delete x;
  ```

- **Java**
  ```
  Integer x = new Integer(2);
  ```
  //no delete
  //need 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

```
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:

```
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

```
Name
\[\text{Location}\]
\[\text{Value}\]
```

```
\text{x} = \text{y}
```

```
\text{x} = \text{y}
```

Assignment
Assignment by sharing

Java:
Student x = new Student("Amy");
Student y = new Student("John");
x.setAge(19);
y = y;
y.setAge(21);

Assignment by cloning

x = y

Aliases

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

After line 1:

Aliases

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

After line 2:

Aliases

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

After line 3:

Aliases

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

After line 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:

```
X -----> 1  
|    |    |
|    |    | ___
|    y    |
```

### 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", x);
(17) return 0;
(18)

Question 1:
Draw box-and-circle diagrams of the variables after line 11 and 15.

Question 2:
Which variables are aliases at each of those points?

Question 3:
What does the program print?

### Dangling References

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*/

### Dangling References

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

### Dangling References

```
int* dangle(void)
{
  int x;
  return &x;
}
```
```
y = dangle();
/* *y is a dangling reference */
```