



Saint Vincent College

Department of Computing
and Information Science



Names

Fr. Boniface, OSB
Lecture 15

(some slides modified from Tucker/Noonan, copyright McGraw-Hill, 2006)

“If they ask me, ‘What is his name?’ what am I to tell them?” (Ex. 3:13)

Example - Static Scope

```

1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }

```

```

14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }

```

```

8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }

```

Line	Reference	Declaration
4	i	
10	h	
11	i	
16	h	
18	h	

Note: *forward references* allowed in Java, not C

Example - Static Scope



```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
```

```
14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
```

```
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
```

Note: *forward references*
allowed in Java, not C

Line	Reference	Declaration
4	i	1
10	h	1
11	i	9
16	h	1
18	h	1

Dynamic Scope



- Bindings established at runtime
- Symbol table *built at compile time*, but *managed at runtime*
- Examples: (early) Lisp, APL, Snobol, Perl, C macros
- Symbol table pushed/popped at runtime when scope entered/exited

Example - dynamic scope

```

1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
    
```

```

14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
    
```

```

8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
    
```

Line	Reference	Declaration
4	i	
10	h	
11	i	
16	h	
18	h	

Example - dynamic scope



```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
```

```
14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
```

```
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
```

Line	Reference	Declaration
4	i	9
10	h	1
11	i	9
16	h	1
18	h	1

- A name is *visible* in its referencing environment
- A name declared in a nested (inner) scope *hides* or *overshadows* a name in an outer scope
- Some languages allow references to names in an outer scope:

```
public Noun(Article art, String noun) {  
    this.art = art;  
    this.noun = noun;  
}
```

- A name is *visible* in its referencing environment
- A name declared in a nested (inner) scope *hides* or *overshadows* a name in an outer scope
- Some languages allow references to names in an outer scope:

```
public Noun(Article art, String noun) {  
    this.art = art;  
    this.noun = noun;  
}
```

Overloading



Modula: library functions

- Read() for characters
- ReadReal() for floating point
- ReadInt() for integers
- ReadString() for strings

Java: overloaded methods

```
public class PrintStream extends
FilterOutputStream {
...
public void print(boolean b);
public void print(char c);
public void print(int i);
public void print(long l);
public void print(float f);
public void print(double d);
public void print(char[ ] s);
```

C++ allows operator overloading
This severely complicates parsing



- Lifetime - (run)time during which a variable is allocated in memory
- Earliest languages - static allocation *only*
- Algol - connected variable (de)allocation w/ scope
- Now, general rule is *scope = lifetime*
- Exceptions
 - ▶ *static* means static allocation (retains value)
 - ▶ “globals” in C, pascal are always statically allocated
 - ▶ dynamic allocation (malloc, new)