Scope Rules

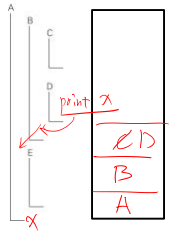
```
string x = "Global";

void function() {
        cout << x << endl;
        string x = "Function";
        cout << x << endl;
}

int main() {
        function();
        {
                string x = "Block 1";
                cout << x << endl;
        }
        cout << x << endl;
}
```


Static Scope Rules

- With STATIC (LEXICAL) SCOPE RULES, a scope is defined in terms of the lexical structure of the program
  - The determination of scopes can be made by compiler
  - All bindings for identifiers can be resolved by examining the program
  - Typically, we choose the most recent, active binding made at compile time
- Note that the bindings created in a subroutine are destroyed at subroutine exit

Static Scope Rules

- Access to non-local variables STATIC LINKS
  - You access a variable in a scope *k* levels out by following *k* static links and then using the known offset within the frame thus found



Static Scope Rules

```
string x;
void baz() {
    int a = 0;
    cout << x << endl;
    x = "Global in baz";
}
void bar() {
    string x = "Local in bar";
    baz();
    cout << x << endl;
}
void foo() {
    int a = 1;
    bar();
    cout << x << endl;
}
int main() {
    x = "Global in main";
    string x = "Local in main";
    cout << x << endl;
    foo();
}
```

x = Local in bar

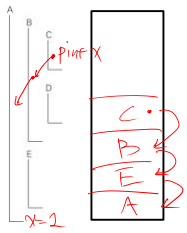
x = Global in ~~main~~ baz

Dynamic Scope Rules

- The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.
- With **dynamic scope rules**, bindings depend on the current state of program execution
  - They cannot always be resolved by examining the program because they are dependent on calling sequence
  - To resolve a reference, we use the most recent, active binding made at run time

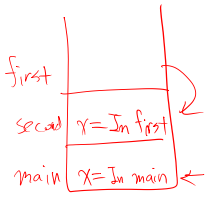
Dynamic Scope Rules

- Access to non-local variables DYNAMIC LINKS
  - You access a variable in a scope *k* levels out by following *k* dynamic links and then using the known offset within the frame thus found



Dynamic Scope Rules

```
#include <iostream>
using namespace std;

string x;
void first() {
    x = "In first";
}
void second() {
    string x;
    first();
}
int main() {
    x = "In main";
    second();
    cout << x << endl;
}
```



Dynamic Scope Rules

- If static scope rules are in effect, the program prints "In first"
- If dynamic scope rules are in effect, the program prints "In main"

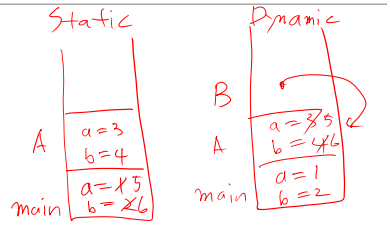
```
#include <iostream>
using namespace std;

string x;
void first() {
    x = "In first";
}
void second() {
    string x;
    first();
}
int main() {
    x = "In main";
    second();
    cout << x << endl;
}
```


Dynamic Scope Rules

- Dynamic scope rules require that we choose the most recent, active binding at run time
- Dynamic scope rules are usually encountered in interpreted languages.

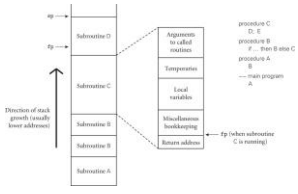
Scope Rules Wrap-up

```
1. procedure main
2.     a : integer := 1
3.     b : integer := 2

4.     procedure A
5.         a : integer := 3
6.         b : integer := 4
7.         B()
8.         print a, b

9.     procedure B
10.        print a, b
11.        a := 5
12.        b := 6

11.    -- body of main
12.    A()
13.    print a, b
```



Referencing Environments

- Where should I store the "actual values" of variables?



Referencing Environments

- Accessing variables with a stack
  - keep a stack (association list) of all active variables
    - When you need to find a variable, hunt down from top of stack
    - slow access but fast calls
- Accessing variables with a table
  - keep a central table with one slot for every variable name
    - Generally, a hash function or something to do lookup
    - slow calls but fast access

Function Resolution

- How to resolve function calls to appropriate functions?
  - name
  - return type
  - # parameters
  - parameter types
- Vary by programming language
  - In C, function signatures are name
  - In C++, function signatures are name + parameter types

```
int function(int x) {
    return x * 2;
}

double function(double x) {
    return x * x;
}

int function(int x, int y) {
    return x + y;
}

int main() {
    printf("%d\n", function(3));
}
```

Function Overloading

- Function overloading
  - Two different things with the same name in C++

```
struct complex {
    double real, imaginary;
};
enum base {dec, bin, oct, hex};

int i;
complex x;

void print_num(int n) { ...
void print_num(int n, base b) { ...
void print_num(complex c) { ...

print_num(i);       // uses the first function above
print_num(i, hex);  // uses the second function above
print_num(x);       // uses the third function above
```

Built-in Operators

- In C++, which are object-oriented, A + B maybe short for either operator+(A, B) or A.operator+(B). In the latter case, A is an instance of a class (module type) that defines an operator+ function.

```
class complex {
    double real, imag;
...
public:
    complex operator+(complex other) {
    return complex(real + other.real, imag + other.imag);
    }
...
};
...
complex A, B, C;
...
C = A + B; // uses user-defined operator+
```

Template

- A syntactic template that can be instantiated in more than one way at compile time

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
...
int k = GetMax<int>(i,j);
```

Conclusions

- Language features can be surprisingly subtle
- A language that is easy to understand leads to
  - a language that is easy to compile
  - more good compilers on more machines
  - better (faster) code
  - fewer compiler bugs