Recitation 2

Jahnavi - jp5867@nyu.edu

Type Bindings:

A binding is an association between entity and attribute like variable and its type.

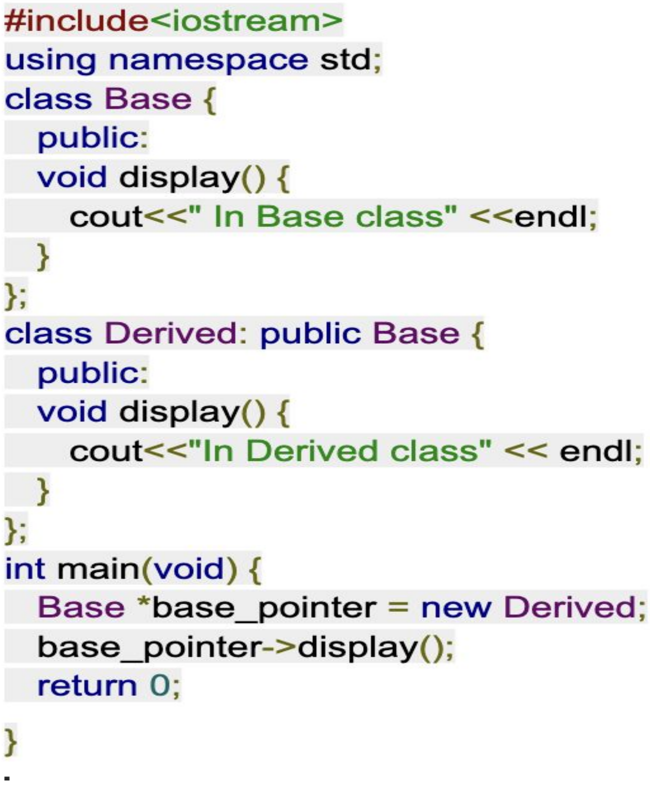
- A binding is static if it first occurs before run time and remains unchanged throughout program execution.

- A binding is dynamic if it first occurs during execution or can change during execution of the program

Early binding:

This is compile time polymorphism. Here it directly associates an address to the function call.

Output: In Base class

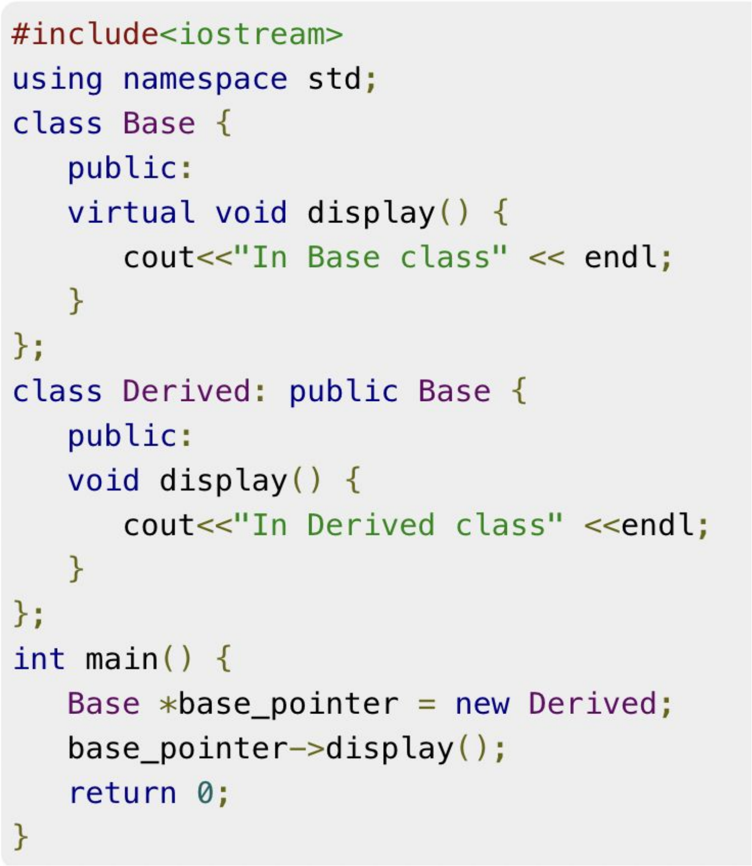


```
#include<iostream>
using namespace std;
class Base {
  public:
  void display() {
    cout<<" In Base class" <<endl;
  }
};
class Derived: public Base {
  public:
  void display() {
    cout<<"In Derived class" << endl;
  }
};
int main(void) {
  Base *base_pointer = new Derived;
  base_pointer->display();
  return 0;

}
.
```

Late binding:

This is runtime polymorphism. The compiler adds code that identifies the object type at runtime then matches the call with the right function definition. This is achieved by using virtual function.

Output: In Derived class



```cpp
#include<iostream>
using namespace std;
class Base {
   public:
   virtual void display() {
      cout<<"In Base class" << endl;
   }
};
class Derived: public Base {
   public:
   void display() {
      cout<<"In Derived class" <<endl;
   }
};
int main() {
   Base *base_pointer = new Derived;
   base_pointer->display();
   return 0;
}
```

Scoping:

Scoping: The rules that determine the visibility of a name in a program

1. Where to find the value that is attached to a name

2. Where that name is valid in a program

3. What to do if the name is defined twice

Scope: The portion of the program where a particular name is visible

Static Scoping:

- The bindings between names and objects can be determined at compile time by examining the program, without consideration of the flow of control at run time.
- To find the correct value corresponding to a name, you look at the closest binding based on the text of the program.
- All modern languages use Static Scoping.

Static Scoping:

```
program A;
var I:integer; K:char;
    procedure B;
    var K:real; L:integer;
        procedure C;
        var M:real;
        begin
        (*scope A+B+C*)
        end;
        procedure D;
        var N:real;
        begin
        (*scope A+B+D*)
        end;
    (*scope A+B*)
    end;
(*scope A*)
end;
```

Dynamic Scoping:

- In a language with dynamic scoping, the bindings between names and objects depend on the flow of control at run time, and in particular on the order in which subroutines are called.
- In simple terms: To find the correct value corresponding to a name, you look at the 'closest' binding on the program stack
- Very few languages implement dynamic scoping Lisp, postscript.
- Because the flow of control cannot in general be predicted in advance, the bindings between names and objects in a language with dynamic scoping cannot in general be determined by a compiler.
- The dynamic resolution can only be determined at run time (late binding)

Static Scoping Vs Dynamic Scoping: Ex - 1

```
int x;
int main()
 {
        x = 14;
        f();
        g();
}
void f() {
        int x = 13;
        h();
}
void g() {
        int x = 12;
        h();
}
void h() {
        printf("%d\n",x);
}
```

Static Scoping: 14 14
Dynamic Scoping: 13 12

Static Scoping Vs Dynamic Scoping: Ex - 2

```
int x;
int main()
 {
        x = 14;
        f();
        g();
}
void f() {
        x = 13;
        h();
}
void g() {
        x = 12;
        h();
}
void h() {
        printf("%d\n",x);
}
```

Static Scoping: 13 12
Dynamic Scoping: 13 12

Static Scoping Vs Dynamic Scoping: Ex - 3

```
const int b = 5;
int foo()
{
        int a = b + 5;
        return a;
}

int bar()
{
        int b = 2;
        return foo();
}
int main()
{
        foo();
        bar();
        return 0;
}
```

Static Scoping: foo and bar return 10 and 10 respectively

Dynamic Scoping: foo returns 10 bar returns 7

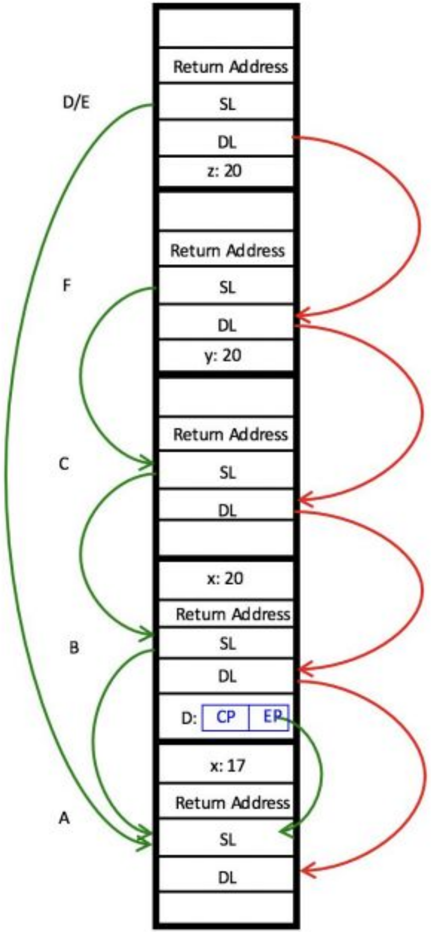
Shortcut evaluation:

```
If x/y == 5 then {
 ** What if y == 0 **
}
```

Lazy Evaluation:

```
If a && b then {
 // Evaluate b if and only if a is true
}
If a || b then {
// Evaluate b if and only if a is false
}
```

```
procedure A()
   x: integer := 17;

   procedure B(procedure D)
      x: integer := 20;

      procedure C()
        procedure F(y:integer)
        begin (* F *)
           D(y)
        end; (* F *)
      begin (* C *)
         F(x);
      end; (* C *)

   begin (* B *)
      C();
   end (* B *)

   procedure  E(z:integer)
   begin (* E *)
      print(z, x);
   end; (* E *)

begin (* A *)
  B(E);
end; (* A *)
```



D/E
Return Address
SL
DL
z: 20
F
Return Address
SL
DL
y: 20
C
Return Address
SL
DL
x: 20
B
Return Address
SL
DL
D:
CP
EP
x: 17
A
Return Address
SL
DL