Subprograms

Programming Languages
William Killian
Millersville University
Outline

- Process Abstraction
- Activation Records
- Executing Subprograms
- Nested Subprograms
- Referencing Environments
Process Abstraction

Overarching Goal

*Generalize program execution to the degree that we do with data structures (Abstract Data Types)*

1. Every subprogram has exactly **one** entry point
2. A *calling subprogram* (caller) is suspended during the execution of the *called subprogram* (callee)
3. Control will **always return** to the caller when the callee terminates
Process Abstraction

1. Every subprogram has exactly one entry point

- Typically called **main**
  - Java: `public static void main(String[])`
  - C/C++: `int main() or int main(int, char**)`
  - C#: `static (void/int) Main([string[]])`
  - D: `(void/int) main([string[]])`
  - Go: `func main()` (as part of main package)

- Implicitly the execution environment in others
  - F#: `[<EntryPoint>]` annotation
  - Python: can check `__name__ == "__main__"`
Process Abstraction

2. A *calling subprogram* (caller) is suspended during the execution of the *called subprogram* (callee)

```c
main () {
    int x;
    x = add (3, 4);
    print (x);
}
```

Begin execution of main
Process Abstraction

2. A *calling subprogram* (caller) is suspended during the execution of the *called subprogram* (callee)

```c
main () {
    int x;
    x = add (3, 4);
    print (x);
}
```

*main* is the *caller*
*add* is the *callee*

*Prepare to call add – still within main*
Process Abstraction

2. A **calling subprogram** (caller) is suspended during the execution of the **called subprogram** (callee)

```cpp
main () {
    int x;
    x = add(3, 4);
    print(x);
}

add (int x, int y) {
    int res;
    res = x + y;
    return res;
}
```

Call `add`. “Pause” execution of `main`.
2. A **calling subprogram** (caller) is suspended during the execution of the **called subprogram** (callee)

```c
main() {
  int x;
  x = add(3, 4);
  print(x);
}

add(int x, int y) {
  int res;
  res = x + y;
  return res;
}
```

*Continue execution of add*
Process Abstraction

2. A calling subprogram (caller) is suspended during the execution of the called subprogram (callee).

```c
main() {
    int x;
    x = add(3, 4);
    print(x);
}
```

```c
add(int x, int y) {
    int res;
    res = x + y;
    return res;
}
```

End execution of add. “Return” back to caller.
Process Abstraction

3. Control will **always return** to the caller when the callee terminates

```c
main() {
  int x;
  x = add(3, 4);
  print(x);
}

add(int x, int y) {
  int res;
  res = x + y;
  return res;
}
```

*End execution of add. “Return” back to caller*
Process Abstraction

3. Control will **always return** to the caller when the callee terminates

```c
main () {
    int x;
    x = add (3, 4);
    print (x);
}
```

*main is the caller*

*add is the callee*

*Continue execution of main*
Process Abstraction

*What do we need to keep track of in order to guarantee the process execution abstraction?*
Process Abstraction

What do we need to keep track of in order to guarantee the process execution abstraction?

• Where we will (eventually) return to
• Parameters
• Local variables

New Concept: Activation Record
Activation Record

Return Address
• Where we will (eventually) return to

Parameters
• Data explicitly passed to the callee from the caller

Local Variables
• Data created within the callee
• Includes all blocks (statically determined)
Activation Record

Return Address
• Where we will (eventually) return to [code]

Dynamic Link
• *Which Activation Record to return to* [data]

Parameters
• Data explicitly passed to the callee from the caller

Local Variables
• Data created within the callee
• Includes all blocks (statically determined)
Activation Record: Caller Semantics

Return Address
save where we are in the code

Dynamic Link
save the current activation record

Parameters
pass all parameters to the callee

Local Variables

Transfer control to callee
Activation Record: Callee Semantics

Return Address

Dynamic Link

Parameters
  read values during execution

Local Variables
  make space for all local variables

Begin execution of code
Activation Record: Return Semantics

**Return Address**
- Restore prior code location

**Dynamic Link**
- Restore prior activation record

**Parameters**
- Update reference parameters during execution

**Local Variables**
- Clear the created space

*Transfer control to caller once done*
Executing a Subprogram
Executing a Program

Two Main Parts

• Code
  • The actual program being executed
  • What the programmer wrote (but the computer understands via compilation / interpreting)

• Data
  • The information created/modified/destroyed during the lifetime of the program execution
  • Also known as !code
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
Executing a Program

```c
int a(int x) {
    int y = x;
    int y = y * y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z = z + y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}

Activation Record:
- Parameters
- Return Address (code)
- Dynamic Link (data)
- Locals
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```

Activation Record:
- Parameters
- Return Address (code)
- Dynamic Link (data)
- Locals
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```

Activation Record:
- **Parameters**
- Return Address *(code)*
- Dynamic Link *(data)*
- **Locals**
Executing a Program

```
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}
int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}
int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```

Activation Record:
- Parameters
- Return Address (code)
- Dynamic Link (data)
- Locals
```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}
int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}
int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
### Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
Executing a Program

```c
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
```
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
int a(int x) {
    int y = x;
    y *= y;
    return y;
}

int b(int x, int y) {
    int z = a(x);
    z += y;
    return z;
}

int main() {
    int i = a(4);
    int j = b(i, 5);
    return j;
}
Executing a Subprogram

Recursive Example
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Recursion?

```c
int fact(int x) {
    int res = x;
    if (res <= 1) {
        return 1;
    }
    res *= fact(x - 1);
    return res;
}

int main() {
    int s = fact(2);
    printf("%d\n", s);
}
```
Nested Subprograms

- A subprogram is considered **nested** when it is defined within another subprogram

```python
function f () {
    function g() {
        return 1;
    }
    return g() + g();
}
```
Nested Subprograms: Issues

• Mathematically pure subprograms (functions) introduce no issues

• **Non-local references** must be resolved (e.g. \( x \))

```c
function f () {
    int x = 4;
    function g() {
        return x * 2;
    }
    return g() + g();
}
```
Nested Subprograms

Locating a Non-Local Reference

1. Find the correct activation record instance
   *The most “recent” activation record containing a reference with a specific name*

2. Determine the correct variable offset within the selected activation record instance.
   *This is easily determined by the local variable ordering*
Nested Subprograms

Two Approaches to Non-Local Reference Resolution

1. **Static Scoping**

   *Create and manage a static chain. The static chain is a list of our ancestors.*

2. **Dynamic Scoping**
   
   • Deep Access
     
     *Scan the dynamic link chain*
   
   • Shallow Access
     
     *Maintain a central “stack” of names to references*
Static Scoping

At a function call the activation record needs to include an additional piece of information: static link.

The static link will refer to the most recent activation record instance of the static parent.

This means we need to scan the dynamic chain!
Static Scoping Problems

• Non-local references are slow when the nesting depth is large
• Cost of non-local references is difficult/impossible to determine
• Simple code changes can alter/change the nesting depth (which changes the performance!)
function main() {
    var x;
    function bigsub() {
        var a, b, c;
        function sub1() {
            var a, d;
            a = b + c; // PLACE 1
        }
        function sub2(x) {
            var b, e;
            function sub3() {
                var c, e;
                sub1();
                e = b + a; // PLACE 2
            }
            sub3();
            a = d + e; // PLACE 3
        }
        sub2(7);
    }
    bigsub();
}
function main(){
    var x;
    function bigsub() {
        var a, b, c;
        function sub1() {
            var a, d;
            a = b + c; // PLACE 1
        }
        function sub2(x) {
            var b, e;
            function sub3() {
                var c, e;
                sub1();
                e = b + a; // PLACE 2
            }
            sub3();
            a = d + e; // PLACE 3
        }
        sub2(7);
    }
    bigsub();
}
Dynamic Scoping

Deep Access
Scan the dynamic link chain
Every activation record instance tracks names

Shallow Access
Central location for all names
One stack required for each name

```c
void sub3() {
    int x, z;
    x = u + v;
    ...
}

void sub2() {
    int w, x;
    ...
}

void sub1() {
    int v, w;
    ...
}

void main() {
    int v, u;
    ...
}
```
function main(){
    var x;
    function bigsub() {
        var a, b, c;
        function sub1() {
            var a, d;
            a = b + c; // PLACE 1
        }
        function sub2(x) {
            var b, e;
            function sub3() {
                var c, e;
                sub1();
                e = b + a; // PLACE 2
            }
            sub3();
            a = d + e; // PLACE 3
        }
        sub2(7);
    }
    bigsub();
}