Procedures vs. Functions

- **Function:**
  - no side effect
  - return a value
  - Function call: expression

- **Procedure:**
  - side effect, executed for it
  - no return value
  - Procedure call: statement

No clear distinction made in most languages
- C/C++: void
- Ada/FORTRAN/Pascal: procedure/function

### Procedures Call

- **Caller:**
  - int f(int y)
  - f(a);
  
- **Callee:**
  - int f(int y){
    - int x;
    - if (y==0) return 0;
    - x=y+1;
    - return x;
  }

  - Control transferred from caller to callee, at procedure call
  - Transferred back to caller when execution reaches the end of body
  - Can return early

### Environment

- **Environment:** binding from names to their attributes

  - **static(global) area**
  - stack
  - heap
  - automatically-allocated spaces
  - manually-allocated spaces
  - under the control of runtime system
  - for dynamic binding

### Activation Record for Nested Blocks

- **Activation record:** memory allocated for the local objects of a block
  - Entering a block: activation record allocated
  - Exit from inner block to surrounding block: activation record released
- int x; //global
  - 
  ```
  int x,y;
  x = y*10;
  int i;
  i = x/2;
  ```
  -
Activation Record for Nested Blocks

```c
int x; //global
{
    int x,y;
    x = y*10;
    {
        int i;
        i = x/2;
    }
}
```

X: Nonlocal variable in the surrounding activation record

Activation Record for Procedures

```c
int x; //global
void B(void) {
    int i;
    i = x/2;
}
void A(void) {
    int x,y;
    x = y*10;
    B();
}
main() {
    A();
    return 0;
}
```

X: global variable in defining environment
Y: local variable in calling environment
I: local variable in called environment

Need to retain information in calling environment

Procedure Call

- Caller:
  ```c
  f(i);
  ```
  
- Callee:
  ```c
  int f(int a){
  ...
  ...
  return a;
  }
  ```

Parameter Passing Mechanisms:
- When and how to evaluate parameters
- How actual parameter values are passed to formal parameters
- How formal parameter values are passed back to actual parameters
Parameter Passing Mechanisms

- Pass/Call by Value
- Pass/Call by Reference
- Pass/Call by Value-Result
- Pass/Call by Name

Example

- What is the result?
  void swap(int a, int b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
  }
  main(){
   int i=1, j=2;
   swap(i,j);
   printf("i=%d, j=%d\n", i, j);
  }
  It depends...

Pass by Value

- Caller:
  f(i); ...a...;

- Callee:
  int f(int a){
   ...a...;
  }

- Most common one
- Replace formal parameters by the values of actual parameters
- Actual parameters: No change
- Formal parameters: Local variables (C, C++, Java, Pascal)

Example: Pass By Value

void swap(int a, int b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
}

main(){
   int i=1, j=2;
   swap(i,j);
   printf("i=%d, j=%d\n", i, j);
}

Are these Pass-by-Value?

- C:
  void f(int *p) { *p = 0; }
  void f(int a[]) { a[0]=0; }

- Java:
  void f(Vector v) { v.removeAll(); }

Yes!

Pass-by-Value: Pointers

- C:
  void f(int *p) { *p = 0; }
  void f(int a[]) { a[0]=0; }
  main() {
   int *q;
   q = (int *) malloc(sizeof(int));
   *q = 1;
   f(q);
   printf("q\n", q);
  }

- Java:
  Vector v = new Vector();
  v.add(new Object());
  void f(Vector v) {
   v.remove(0);
   f(v);
  }

- It depends...
Pass-by-Value: Pointers

• C:
  void f(int *p) { p = (int *) malloc(sizeof(int)); *p = 0; }
  main() {
    int *q;
    q = (int *) malloc(sizeof(int));
    *q = 1;
    f(q);
    printf("%d\n", q[0]);
  }

• What happens here?

Pass-by-Value: Arrays

• C:
  void f(int p[]) { p[0] = 0; }
  main() {
    int q[10];
    q[0]=1;
    f(q);
    printf("%d\n", q[0]);
  }

• What happens here?

Pass-by-Value: Java Objects

• Java:
  void f(Vector v) { v.removeAll(); }
  main() {
    Vector vec;
    vec.addElement(new Integer(1));
    f(vec);
    System.out.println(vec.size());
  }

• What happens here?

Pass by Reference

• Caller:
  int f(int a){
    f(a);
  }

• Callee:

• Formal parameters become alias of actual parameters
• Actual parameters: changed by changes to formal parameters
• Examples:
  • Fortran: the only parameter passing mechanism
  • C++ (reference type, &) / Pascal (var)
Example: Pass By Reference

C++ syntax. Not valid in C

```cpp
void swap(int &a, int &b) {
    int temp;
    temp = a;
    b = temp;
    a = b;
}
```

```cpp
main() {
    int i=1, j=2;
    swap(i,j);
    printf("i=%d, j=%d\n", i, j);
}
```

Pass-by-Reference: How to minic it in C?

- C:
  ```cpp
  void f(int *p) { *p = 0; }
  main() {
      int q;
      q = 1;
      f(&q);
      printf("\n", q);
  }
  ```

- It is really pass-by-value. Why?

It is really pass-by-value

- C:
  ```cpp
  void f(int *p) { *p = 0; }
  main() {
      int q;
      q = 1;
      f(&q);
      printf("\n", q);
  }
  ```

Pass-by-Reference: C++ Constant Reference

- C++:
  ```cpp
  void f(const int & p) {
      int a = p;
      p = 0;
  }
  main() {
      int q;
      q = 1;
      f(q);
      printf("\n", q);
  }
  ```

- What happens here?

Pass-by-Reference: C++ Reference-to-Pointer

- C++:
  ```cpp
  void f(int * &p) { *p = 0; }
  main() {
      int *q;
      int a[10];
      a[0]=1;
      q=a;
      f(q);
      printf("\n", q[0], a[0]);
  }
  ```

- What happens here?

Pass-by-Reference: C++ Reference-to-Pointer

- C++:
  ```cpp
  void f(int * &p) { p = new int; *p = 0; }
  main() {
      int *q;
      int a[10];
      a[0]=1;
      q=a;
      f(q);
      printf("\n", q[0], a[0]);
  }
  ```

- What happens here?
Pass-by-Reference:

C++ Reference-to-Array

- C++:
  ```cpp
  void f(int (&p)[10]) {
    p[0]=0;
  }
  main() {
    int *q;
    q = a;
    f(a);
    printf("%d, %d\n", q[0], a[0]);
  }
  ```

- What happens here?

Pass by Value-Result

- Caller:
  ```cpp
  int f(int a){
  ....
  }
  ```

- Callee:
  ```cpp
  ....
  ```

- Combination of Pass-by-Value and Pass-by-Reference (Pass-by-Reference without aliasing)
- Replace formal parameters by the values of actual parameters
- Value of formal parameters are copied back to actual parameters

Example: Pass By Value-Result

```cpp
void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
main(){
  int i=1, j=2;
  swap(i,j);
  printf("i=%d, j=%d\n", i, j);
}
```
Pass-by-Name: Side Effects

```c
int p[3] = {1, 2, 3};
int i;

void swap(int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
main(){
    i = 1;
    swap(i, a[i]);
    printf("%d, %d\n", i, a[i]);
}
```

• What happens here?

Some Variants

- **Pass by Name**
  - Evaluated at every use, in the calling environment
- **Pass by Need**
  - Evaluated once, memorized for future use
- **Pass by Text (Macro)**
  - Evaluated using the called environment.

• All belong to Non-strict evaluation (lazy evaluation)

Comparisons

- **Call by Value**
  - Efficient. No additional level of indirection.
  - Less flexible and less efficient without pointer.
  - (array, struct, union as parameters)
- **Call by Reference**
  - Require one additional level of indirection (explicit dereferencing)
  - If a parameter is not variable (e.g., constant), a memory space must be allocated for it, in order to get a reference.
  - Easiest to implement.
- **Call by Value-Result**
  - You may not want to change actual parameter values when facing exceptions.
- **Call by Name**
  - Lazy evaluation
  - Difficult to implement