```
printf("abc");
printf("%s", "abc");
printf("%s %d", A, 10);
```

Varying # of prameters

```
int    sum(int *X, int r, int c){
   int total=0;
   int i,j;
   for(i=0; i<r; i++)
      for(j=0; j<c; j++)
         total+=*(X+ (i*c)+j );
                      int
}
```

Type Checking Parameters

Includes 2 aspects

- Ensuring that the number of actual parameters is equal to the number of formal parameters. Given **void f(int a, int b){…}**, **f(x)** is invalid
- Actual parameters are compatable with corresponding formal parameters. Given **void f(int a){…}**, **f("abc")** is invalid.

Type Checking Parameters In Languages

- Early languages such as original Fortran 77 did not require type checking of parameters
- Most later languages like C++, Java require it
- In relatively recent languages Perl, Javascript, PHP do not

Type Checking Parameters in Languages

- Original C, neither number of parameters nor their types were checked
- In C89, formal parameters can be defined in 2 ways
  - Without type checking

    double sin(x){double x; …}

    double value;

    int count;

    …

    value = sin(count);// legal but logical error
  - With type checking

    double sin(double x){…}

    value=sin(count); //legal and count is coerced to double

Type checking parameters

- In C99 and C++, type checking parameters can be avoided by replacing parameters with …
- An example

void printf(char* format_string, …){…}

Generic code

```
void printf(const char * format, ...){
    int i; char c; char * s; double d; // locations to store values to be printed
    va_list ap; // "argument pointer" to a variable arg list
    va_start(ap, format); // initialize arg pointer using last known arg
    for (char *p = format; *p != '\0'; p++) {
        if (*p == '%') {
                    switch (* ++ p) {
                        case 'd': i = va_arg(ap, int); break;
                        case 's': s = va_arg(ap, char *); break;
                        case 'c': c = va_arg(ap, char); break;
                        ...
                    }// switch
                    // print value of i, or s, or c, ...
                    ....
        }
        else
        //print *p
        ...
    }//for
    va_end(ap); //restore any special stack manipulations
} // printf
```

'c'

printf("%c%d%i",

%s

%d

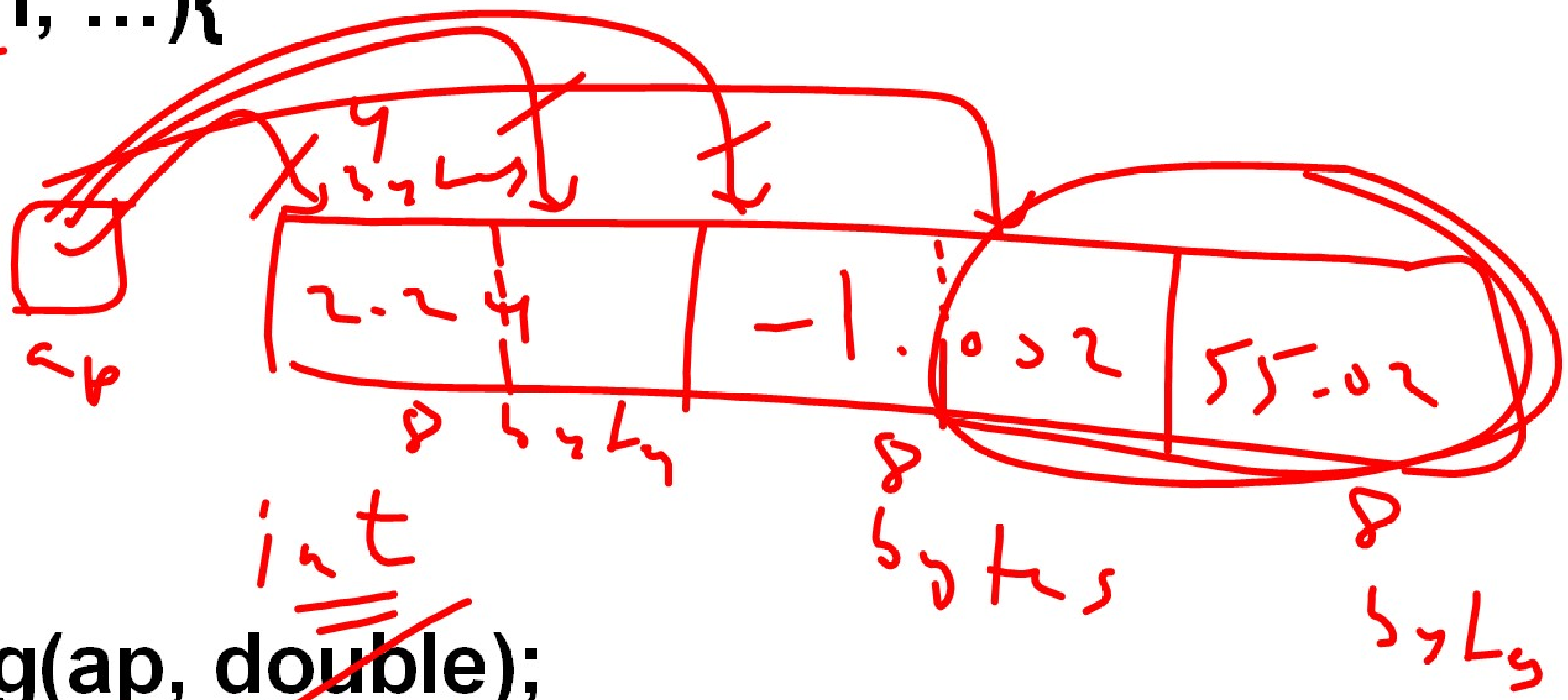
%c

%f

...

C++ Varying Number of Parameters Example

```
double average(int i, ...){
    double total=0;
    int j;
    va_list ap;
    va_start(ap,i);
    for(j=1; j<=i; j++)
        total+=va_arg(ap, double);
    va_end(ap);
    return total/i;
}
...
cout<<average(3, 2.24, -1.002, 55.02);
cout<<average(2, 1.5, -4.43);
```



ap

"abcdefg"

Varying number of arguments in Ruby

```
def showargs(*args)
  printf("%d arguments:\n", args.size)

  for i in 0..args.size do # a..b is a to b-1
    printf("#%d: %s\n", i, args[i])
  end
end
```

```
>> showargs(1, "two", 3.0)
3 arguments:
#0: 1
#1: two
#2: 3.0
```

args

>> showargs(1,1,1,1,1,1) ↵

Varying number of parameters in C#

```
public static void DisplayItems(params int[ ] X) {
   for(int i=0; i<X.length; i++)
        console.write(X[i] + "\t");
}

...

DisplayItems(1, 2, 3, 4);
DisplayItems(44, 12);
int [ ] A = new int[5] {100, 200, 300, 400, 500};
DisplayItems(A);
```

Multi Dimensional Arrays as Parameters

m x n

- A C/C++ function cannot receive a parameter matrix with variable number of columns

void f(int X[ ][ ], int r, int c){…} // invalid

- A possible solution

void f(float * X, int r, int c){…}

Inside f, **X[i][j]=10** is replaced with

***(X+(i*c)+j)=10**

Multi Dimensional Arrays as Parameters in Java and C#

- Arrays and matrices are objects
- Each array has a length attribute that is set when the array is created. **length** in Java and **Length** in C#
- Arrays are one dimensional but elements can be arrays
- **X.length** returns number of rows in matrix X
- **X[0].length** returns number of elements in array row 0 in matrix X

Java example

```
float sumMatrix(float mat[ ][ ]){
   float sum=0;
   for(int i=0; i<mat.length; i++)
        for(int j=0; j<mat[~~row~~i].length; j++)
              sum+=mat[i][j];

   return sum;
}
```

Generic Subprograms

- A subprogram takes parameters of different types on different actications
- Lisp example

  (define (square X) (* X X))

  > (square 3)

  9

  > (square 1.5)

  2.25
- Another example is C++ template

C++ template

```
template <class T> //T is a type parameter
T max(T x, T y){
  if (x>y) return x; else return y;
}
```

Preprocessor

The preprocessor instantiates the template for every function call statement in the program creating an ordinary function. **max(10,20)** creates a function obtained by replacing T with int in the template.

C++ template

```
template <class T>
void sort(T list[ ], int length){
    int i, j;
    T temp;
    for(i=0; j<=length-1; j++)
          for(j=i+1; j<=length-1; j++)
                if(list[i]>list[j]){
                      temp=list[i];
                      list[i]=list[j];
                      list[j]=temp;
                } //if
}
....
float B[100]={…};
sort(B,100);
char C[20]={…};
sort(C, 20);
```

Functions as Parameters

- Often it is needed to have a function receiving another function as a parameter. Some languages such as Ada do not allow parameters to be functions
- Pascal allows function parameters

```
procedure integrate(function f(x:real): real; L,U: real; var result: real);
   var
     temp:real;
     …
begin
   …
   temp:=f(L);
   …
end;
```

C/C++

- Allow pointers to functions as parameters
- The function prototype is provided to be able to do type checking

```
void f(int l, int (* g) (int, int)){
   int j, k, l;
   j=(*g)(k, l);
  …
}
```

Referencing environment for executing a passed function

In languages that allow nested subprograms such as Javascript there three choices:

- *Shallow binding*: the environment of the call statement of the passed function
- *Deep binding*: the environment of the definition of the passed subprogram
- *Ad hoc binding*: the environment of the call statement that passed the function as an actual parameter

Referencing environment for executing a passed function

void main(){ …f(k);… }

void f(g){… g(); …}

void k( ){ …}

How to determine referencing environment of K?

deep binding: K definition (static scope)

shallow binding: environment of **g()** statement inside f

ad hoc: environment of **f(k)** statement inside main

Example

```
//Javascript syntax
function sub1( ){ //environment of of sub2 [deep binding]
    var x;
    function sub2( ){
        alert(x); //Create a dialog box with value of x
    }
    function sub3( ){
        var x;
        x=3;
        sub4(sub2); // call statement that passed sub2 [ ad hoc]
    };
    function sub4(subx){
        var x;
        x=4;
        subx( ); // sub2 call statement [shallow binding]
    }
    x=1;
    sub3();
};
```

Output

- Shallow binding: 4
- Deep binding: 1
- Ad hoc: 3

Overloaded Subprograms

- An overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing envirnment
- Every version must have a different protocol (prototype)
- C does not allow overloading
- C++, Java, Ada, and C# have overloaded constructors

Overloaded Subprograms

- C++, Java, and C# allow mixed mode operations and hence the return type cannot be used to distinguish an overloaded function

  //the following overloading is invalid

  int f(int a){…}

  double f(int b){…}

  int x;

  x+f(…);

  // f(…) might return int or double and the return type cannot be used to distinguish which f

- Ada does not allow mixed mode operations and hence the overloading above is valid

  x+f(…)

  // f(…) is considered int and the first f will be called

Overloading in Ruby

Ruby does not allow overloading. A new definition replaces the old one

```
def wrap(s)
    wrap(s, “()”)
end
def wrap(s, wrapper)
    wrapper[0,1] + s + wrapper[1,1]
end
```

```
> wrap(“x”)
ArgumentError: wrong number of arguments (1 for 2)
>wrap(“testing”,”[ ]”)
“[testing]”
```

Overloading in Ruby

An alternative for failed overloading attempt:

```
def wrap(s, wrapper = “()”)
  wrapper[0,1] + s + wrapper[1,1]
end
> wrap(“x”, “<>”)
“<x>”
> wrap(“x”)
“(x)”
```