IT253

Functions

Survey results

- 45 responded
- ~50% are saying the pace is fast
- ~85% are happy with content and the delivery
- Free form comments:
  - More examples, go slower
  - Material to study
  - More than 2 hours difficult to sit – can be split into two sessions on Saturday?
  - Compilers should not part of this course
  - Not audible – fix audio issues

Survey results…2

- Free form comments…contd….
  - Call only last two digits during attendance ☺
  - LHC lecture halls are better
- Course correction
  - More examples will be given
  - Material to study (given)
  - Can we have two sessions on Saturday too? 1.5-2 hours * 3 per week?
  - LHC can be considered
  - Audio – will try and fix the issues

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Functional PLs, Logical PLs, Lambda Calculus, Event driven programming, Concurrency
29-36	Virtual Machines, Managed Languages, JIT, Case study

Procedural Abstraction

Can be overloaded (e.g., binary +)

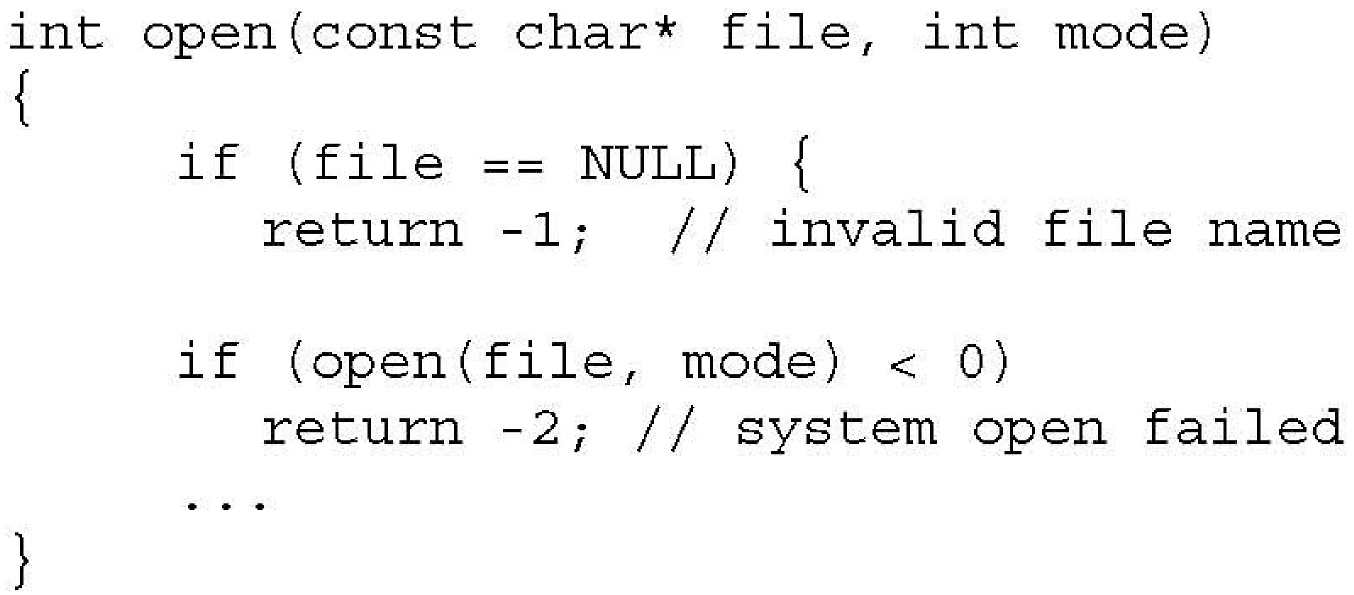
Can pass arguments into the scope

Contains local variable declarations and statements

- Procedure is a named parameterized scope
  - Allows programmer to focus on a function interface, ignoring the details of how it is computed
- Value-returning functions
  - Example: x = (b*b - sqrt(4*a*c))/2*a
- Non-value returning functions
  - Called “procedures” (Ada), “subroutines” (Fortran), “void functions/methods” (C, C++, Java)
  - Have a visible side effect: change the state of some data value not defined in the function definition
  - Example: strcpy(s1,s2)

System Calls

- OS procedures often return status codes
  - Not the result of computing some function, but an indicator of whether the procedure succeeded or failed to cause a certain side effect

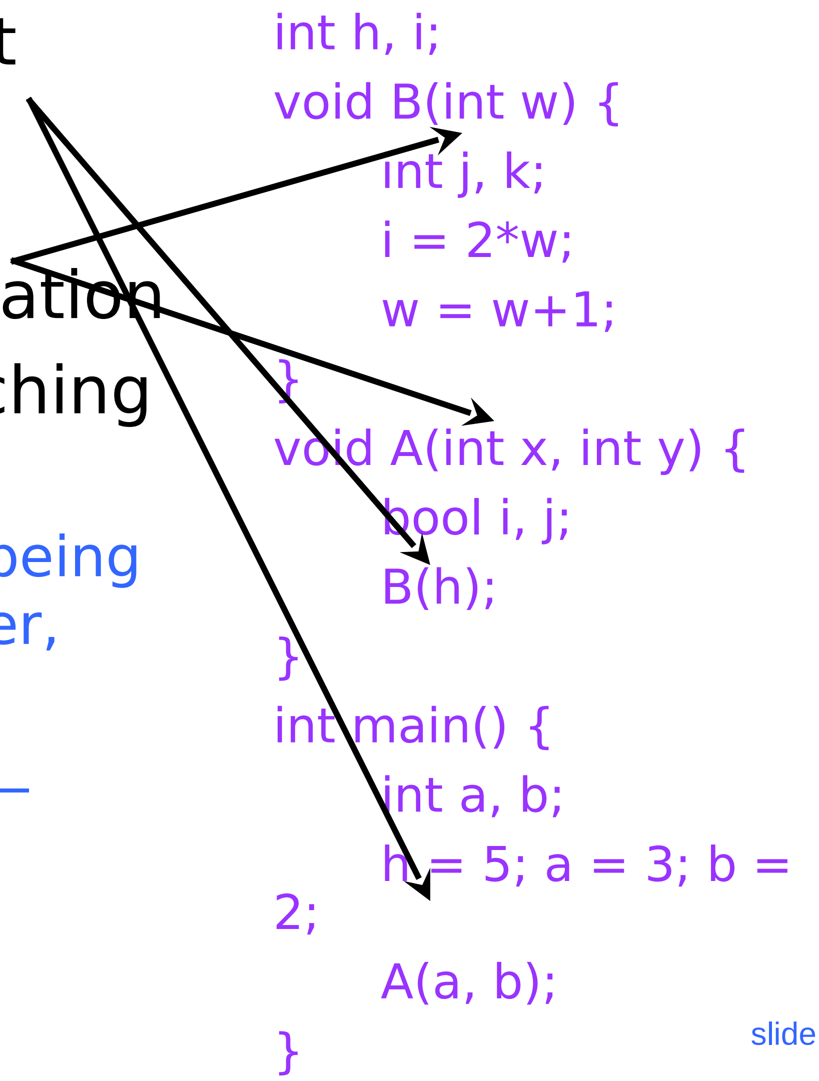


```
int open(const char* file, int mode)
{
     if (file == NULL) {
       return -1;  // invalid file name

     if (open(file, mode) < 0)
       return -2; // system open failed
     ...
}
```

Arguments and Parameters

- Argument: expression that appears in a function call
- Parameter: identifier that appears in function declaration
- Parameter-argument matching by number and position
  - Exception: Perl. Instead of being declared in a function header, parameters are available as elements of special array @_



```
int h, i;
void B(int w) {
          int j, k;
          i = 2*w;
          w = w+1;
}
void A(int x, int y) {
          bool i, j;
          B(h);
}
int main() {
          int a, b;
          h = 5; a = 3; b =
2;
          A(a, b);
}
```

Parameter Passing Mechanisms

- By value
- By reference
- By value-result
- By result
- By name

Pass by Value

- Caller passes r-value of the argument to function
  - Compute the value of the argument at the time of the call and assign that value to the parameter
  - Reduces “aliasing”
    - Aliasing: two names refer to the same memory location
- Function cannot change value of caller’s variable

```
int increment (int age)
    { age = age + 1; return age; }
main () {
    int myAge=45;
    int newAge increment(myAge);
}
```

Pass by Value

- Caller passes r-value of the argument to function
  - Compute the value of the argument at the time of the call and assign that value to the parameter
  - Reduces “aliasing”
    - Aliasing: two names refer to the same memory location
- Function cannot change value of caller’s variable
- All arguments in C and Java are passed by value
  - To allow caller’s variables to be modified, pointers can be passed as arguments
    - Example: void swap(int *a, int *b) { … }

Is there a contradiction here?

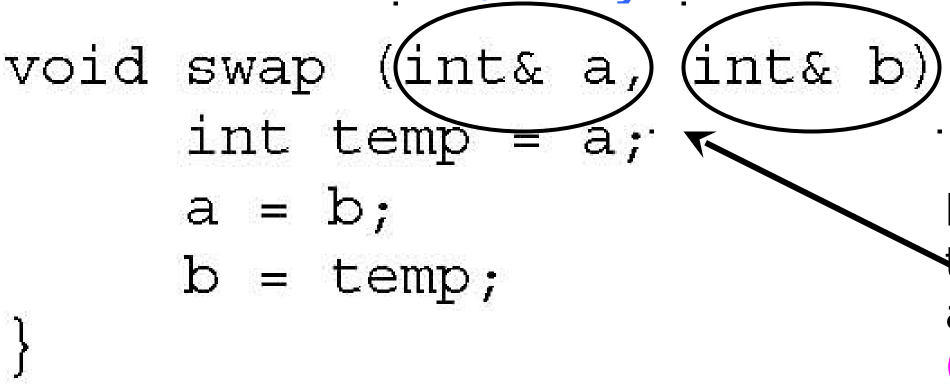
Pass by Reference

- Caller passes l-value of the argument to function
  - Compute the address of the argument and assign that address to the parameter
  - Increases aliasing (why?)
- Function can modify caller’s variable via the address it received as argument

```
int h, i;
void B(int* w) {
        int j, k;
        i = 2*(*w);
        *w = *w+1;
}
void A(int* x, int* y) {
        bool i, j;
        B(&h);
}
int main() {
        int a, b;
        h = 5; a = 3; b =
2;
        A(&a, &b);
}
```

Pass by Reference in C++

- Special “reference type” indicates that l-value is passed as argument
  - Recall that in C, only r-values can be arguments



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

l-values for C++ reference types are completely determined at compile-time

(why is this important?)

- & operator is overloaded in C++
  - When applied to a variable, gives its l-value
  - When applied to type name in parameter list, means pass the argument by reference

Two Ways To Pass By Reference

C or C++

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

int x=3, y=4;
swap(&x, &y);
```

C++ only

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

int x=3, y=4;
swap(x, y);
```

Which one is better? Why?

Pass by Value-Result

- Pass by value at the time of the call and/or copy the result back to the argument at the end of the call (copy-in-copy-out)
  - Example: “in out” parameters in Ada
- Reference and value-result are the same, except when aliasing occurs
  - Same variable is passed for two different parameters
  - Same variable is both passed and globally referenced from the called function

Pass by Name

- Textually substitute the argument for every instance of its corresponding parameter in the function body
  - Originated with Algol 60 but dropped by Algol’s successors -- Pascal, Ada, Modula
- Example of late binding
  - Evaluation of the argument is delayed until its occurrence in the function body is actually executed
  - Associated with lazy evaluation in functional languages (e.g., Haskell)

An example

- Computes $\sum_{i=1}^{100} \frac{1}{i}$ in Algol 60

passed by name

```
begin
   integer i;
   real procedure sum (i, lo, hi,
term);
          value lo, hi;
          integer i, lo, hi;
          real term;
   begin
          real temp;
          temp := 0;
          for i := lo step 1 until hi do
               temp := temp + term;
          sum := temp
   end;
   print (sum (i, 1, 100, 1/i))
end
```

becomes 1/i when sum is executed

Macro

- Textual substitution

```
#define swap(a,b) temp=a; a=b; b=temp;
…
int x=3, y=4;
int temp;
swap(x,y);
```

Textually expands to

```
temp=x; x=y; y=temp;
```

- Looks like a function definition, but …
  - Does not obey the lexical scope rules (i.e., visibility of variable declarations)
  - No type information for arguments or result

Problems with Macro Expansion

```
#define swap(a,b) temp=a; a=b; b=temp;
…
if (x<y)
   swap(x,y);
```

Textually expands to

```
if (x<y)
   temp=x;
x=y;
y=temp;
```

Why not `#define swap(a,b) { int temp=a; a=b; b=temp; }`?

Instead

```
#define swap(a,b) do {
                          int temp=a; a=b; b=temp;
                        } while(false);
```

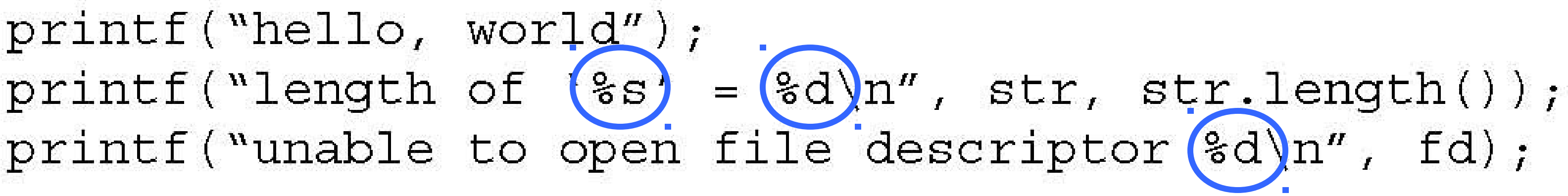
Fixes type of swapped variables

Variable Arguments

- In C, can define a function with a variable number of arguments
  - Example: void printf(const char* format, …)

Part of syntax!

- Examples of usage:



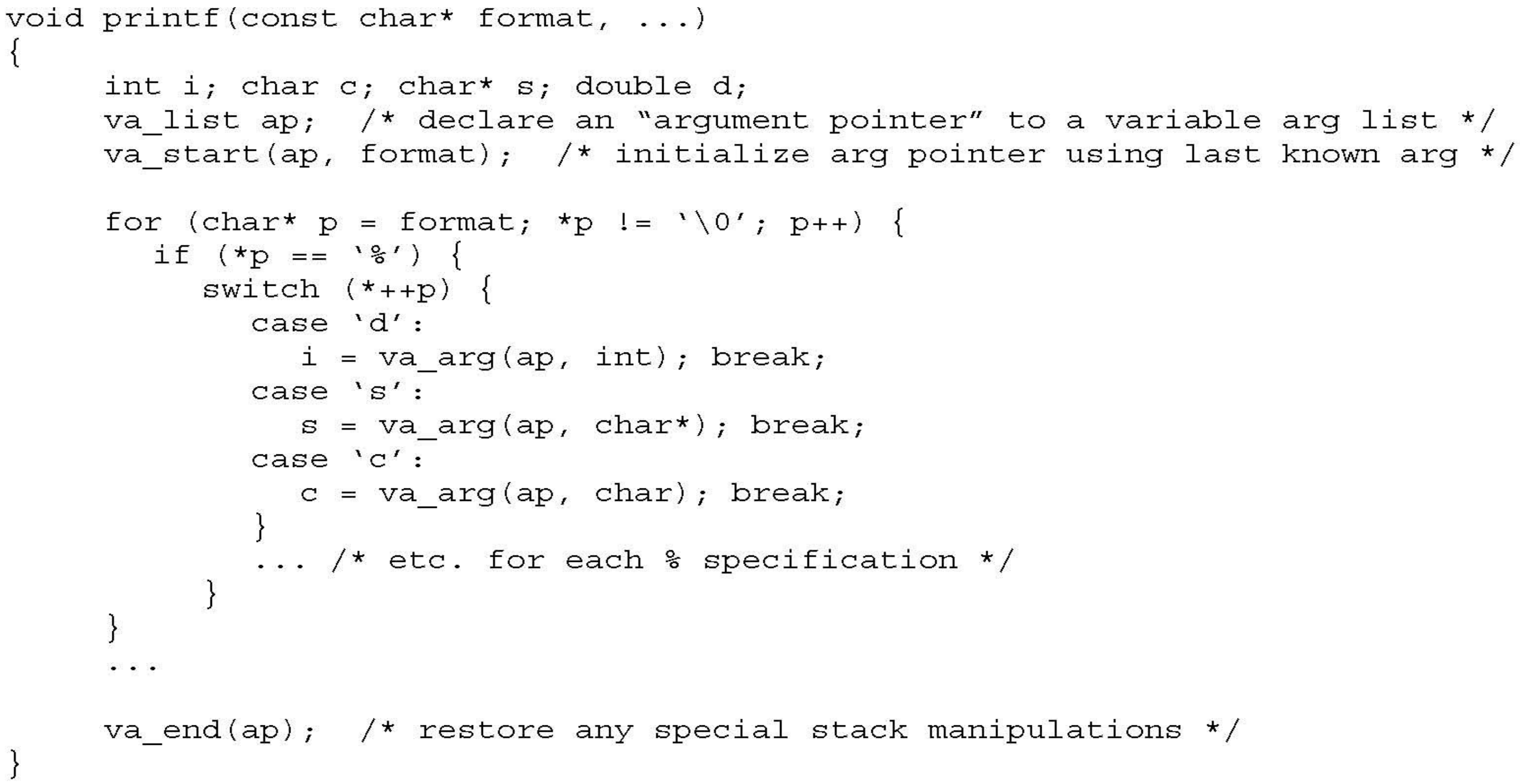
```
printf(“hello, world”);
printf(“length of ‘%s’ = %d\n”, str, str.length());
printf(“unable to open file descriptor %d\n”, fd);
```

Format specification encoded by special %-encoded characters

- %d,%i,%o,%u,%x,%X – integer argument
- %s – string argument
- %p – pointer argument (void *)
- Several others (see C Reference Manual!)

Implementation of Variable Args

- Special functions va_start, va_arg, va_end compute arguments at run-time



```
void printf(const char* format, ...)
{
      int i; char c; char* s; double d;
      va_list ap;  /* declare an “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;
               }
               ... /* etc. for each % specification */
            }
      }
      ...

      va_end(ap);  /* restore any special stack manipulations */
}
```

Implementation of Variable Args

- Special functions va_start, va_arg, va_end compute arguments at run-time

```
#include <stdarg.h> #include <stdio.h>
double average ( int num, ... )
{
    va_list arguments;
    double sum = 0;
    va_start ( arguments, num );
    for ( int x = 0; x < num; x++ )
         { sum += va_arg ( arguments, double ); }
    va_end ( arguments );
    return sum / num;
}
int main() {
    printf( "%f\n", average ( 3, 12.2, 22.3, 4.5 ) );

    printf( "%f\n", average ( 5, 3.3, 2.2, 1.1, 5.5, 3.3 ) );
}
```