

CS 460

Programming Languages

Fall 2008

Dr. Watts

(Week 2)



Thoughts on Computer Programming Languages



- The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence. (Edsger Dijkstra)
- Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice. (Sun FORTRAN Reference Manual)
- Cobol has almost no fervent enthusiasts. As a programming tool, it has roughly the sex appeal of a wrench. (Charles Petzold)
- C++ is the only current language making COBOL look good. (Bertrand Meyer)
- C++ has its place in the history of programming languages. Just as Caligula has his place in the history of the Roman Empire. (Robert Firth)
- Arguing that Java is better than C++ is like arguing that grasshoppers taste better than tree bark. (Thant Tessiman)
- Java is, in many ways, C++-. (Michael Feldman)
- If Java had true garbage collection, most programs would delete themselves upon execution. (Robert Sewell)
- It is practically impossible to teach good programming style to students that have had prior exposure to BASIC; as potential programmers they are mentally mutilated beyond hope of regeneration. (Edsger Dijkstra)
- In my egotistical opinion, most people's C programs should be indented six feet downward and covered with dirt. (Blair P. Houghton)
- C++ is history repeated as tragedy. Java is history repeated as farce.
- (Scott McKay)
- Unix and C are the ultimate computer viruses. (Richard P Gabriel)



Subprograms – why?

- Why do we create subprograms?
 - Reusability
 - Modularity
 - Readability
 - Writeability
 - Software Engineering
- Why are we studying subprograms (now)?
 - Recursion – advanced sorting
 - Scheme – a functional language



Subprogram components

- Subprogram definition
 - Describes interface and actions of the subprogram
- Subprogram call
 - Explicit request that subprogram be executed
 - Complies to defined interface
- Subprogram header
 - Indicates type of subprogram
 - Indicates name of subprogram
 - Optionally, specifies list of parameters

Subprogram Header Examples



- FORTRAN
 - SUBROUTINE ADDER (parameters)
- Pascal
 - Procedure adder (parameters);
 - Function adder (parameters) : returnType;
- C++
 - returnType adder (parameters);
 - returnType adder (parameters) { }
- Python
 - Def adder (parameters) :

Alternative Subprogram Definitions



- Function overloading
- Within #ifdef, #else, and #endif
- Nested subroutines
- Within Python if statement



Other subprogram issues

- Prototypes vs definitions
 - Why prototypes?
 - C, C++, Pascal
- Subprogram vs method
 - Method associated with object of a class
- Formal vs actual parameters
 - Formal – in function definition
 - Actual – in function call
- Parameter profile
 - Number, order and types of formal parameters



More subprogram issues

- Procedure vs function
 - Procedure – no return value
 - Function returns a value
- Nested subprograms
 - Scoping issues
- Local vs global variables

Function names and return types



- Function names
- Return types
 - Scalar
 - Composite

Parameters



- Formal parameters in subprogram description
- Actual parameters in subprogram call
- Positional parameters
- Keyword parameters
- Variable number of parameters
- Default parameters

Parameter Passing Types



- In mode, out mode, inout mode
- Pass-by-value
 - In mode – value is passed
- Pass-by-result
 - Out mode – value is “returned”
- Pass-by-value-result
 - In mode then out mode – value passed then “returned”
- Pass-by-reference
 - Inout mode – address is passed
- Pass-by-name
 - Inout mode – actual name is substituted

Type checking parameters



- Compile time checking
- Type coalescing

Passing arrays



- Passing base address
- Passing copy of array
- Passing multidimensional arrays

Parameter passing design considerations



- Efficiency
- One-way or two-way data transfer



C++ vs C

- C++ or C : pass by value

- What will this do?

- `void Swap (int a, int b)`

```
{  
    int t = a;  
    a = b;  
    b = t;  
}
```

```
int x, y;  
Swap (x, y);
```



C++ vs C

- C++ : pass by reference

- May “implicitly” pass and use address if two-way transfer is needed

- `void Swap (int &a, int &b)`

```
{  
    int t = a;  
    a = b;  
    b = t;  
}
```

```
int x, y;  
Swap (x, y);
```



C++ vs C

- C : only pass by value
 - Must explicitly pass and use address if two-way transfer is needed

```
void Swap (int * a, int * b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
int x, y;
Swap (&x, &y);
```



Passing subprograms as parameters

- C++ Example 1

```
int function1 (float param)
{
    return int (param);
}

int function2 (float param)
{
    return int (param + 0.5);
}

int function3 (int f (float), float p)
{
    return f(p) * f(p);
}

int main ()
{
    cout << function3 (function1, 3.5);
    cout << function3 (function2, 3.5);
    return 0;
}
```

Passing subprograms as parameters



- C++ Example 2

```
int function1 (float param);
int function2 (float param);
int function3 (int f (float), float p);

int main {}
{
    int (* array []) (float) = {function1, function2};
    for (int i = 0; i < 2; i++)
        cout << function3 (array[i], 5 + 4 * i / 3.0);
    return 0;
}
```

Passing subprograms as parameters



- Issues

- Passed function must have identical parameters
 - Type and number
- Passed functions must have identical return types
- When would you ever use this?
 - Calling functions based on a code
 - Calling “operators” based on op-codes

Overloading subprograms



- C++ Example

```
void Swap (int & a, int & b)
{
    int t = a; a = b; b = t;
}
void Swap (float & a, float & b)
{
    float t = a; a = b; b = t;
}

int main ()
{
    int x, y;
    Swap (x, y);
    float p, q;
    Swap (p, q);
    return 0;
}
```

Overloading subprograms



- Issues

- Complexity
- Have to add new function for each new type used in application
- Not available in all languages



Generic subprograms

- C++ Example

```
template <class T>
T min (T arg1, T arg2)
{
    if (arg1 < arg2)
        return arg1;
    else
        return arg2;
}

int main ()
{
    int i1, i2;      float f1, f2;
    char * c1, * c2; string s1, s2;

    cout << min (i2, i2);
    cout << min (f1, f2);
    cout << min (c1, c2);
    cout << min (*c1, *c2);
    cout << min (s1, s2);
}
```



Generic subprogams

- Issues
 - Compile time errors
 - Multiple instantiations

Side effects of subprograms



- Modification of non-local variables
- Global variables may be modified
- Sometimes “unexpected”

Return types



- Scalar only?
- Composite types?
- Pointers?

Overloaded operators



- C++ example

```
class CExample
{
public:
    CExample operator + (const CExample & E);
    friend CExample operator / (const CExample & E1
                               const CExample & E2);
    friend ostream & operator << (ostream & O,
                                   const CExample & E);
private:
    int variable;
};

int main ()
{
    CExample obj;
    cout << obj;
    obj = obj + obj;
    obj = obj / obj;
}
```

Member vs friend implementation



```
CExample operator + (const CExample & E)
{
    CExample sum;
    sum.variable = variable + E.variable;
    return sum;
}
```

```
CExample operator / (const CExample & E1 const CExample & E2)
{
    CExample diff;
    diff.variable = E1.variable / E2.variable;
    return diff;
}
```

Overloaded operators



- Issues
 - Complexity
 - Only works with classes (OOP)
 - Not available in all languages

Co-routines



- Symmetric unit control model
- Maintain status between activations
- Multiple entry points
 - Execution can begin at entry points other than beginning
- Resume
 - Starts execution at point after where it left
- Quasi-concurrency
 - Only 1 co-routine executing at a time

Other C Programming Hints



- Opening a file

```
FILE * filePointer;  
filePointer = fopen (argv[1], "r");
```

- Closing a file

```
fclose (filePointer);
```

- Reading from a file

```
int count;  
fscanf (filePointer, "%d", &count);
```

- Writing to standard output (console)

```
printf ("%d", count);
```

Other C Programming Hints



- Allocating space

```
int * list = (int *) malloc (count * sizeof (int));
```

- Freeing allocated space

```
free ((void *) list);
```

C Programming Resources



- [C Programming Language \(2nd Edition\) \(Prentice Hall Software\)](#)
 - by Brian W. Kernighan and Dennis M. Ritchie
 - April 1, 1988
- [C: A Reference Manual \(5th Edition\)](#)
 - by Samuel P. Harbison and Guy L. Steele
 - Mar 3, 2002

Scheme



- Dialect of LISP
- MIT – mid-1970s
- Interpreted
- Expressions evaluated by the function EVAL
- Only primitives:
 - + (add), - (subtract), * (multiply), / (divide)
 - +, * may have zero or more operators
 - (+ 4 6) returns 10; (+) returns 0
 - (- 7 5) returns 0
 - (* 3 2 4) returns 24; (*) returns 1
 - (/ 6 8) returns $\frac{3}{4}$ or 0.75 (implementation dependent)

Semantics of call and return



- Basic subprogram call actions
 - Save the execution state of the current program unit
 - Pass the parameters
 - Pass the return address to the callee
 - Transfer control to the callee

Semantics of call and return



- Basic subprogram return actions
 - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
 - If the subprogram is a function, the return value is copied to a place accessible to the caller
 - The execution status of the caller is restored
 - Control is passed back to the caller

Subprogram call and return storage requirements



- Status information about the caller
- Parameters
- Return address
- Return value for functions

- Activation record – stack frames

Stack-dynamic local variables



- Multiple versions of local variable space
- Support for recursion
- Compiler must generate code to cause the implicit allocation and deallocation of local variables
- Dynamic link – pointer to the activation record at the top of the call stack (EP)

Non-recursive example



```
char a (char b)
{
    cout << "in a, b = " << b << endl;
    return b;
}

int c (int d)
{
    cout << "in c, d = " << d << endl;
    cout << "in c, a = " << a ('a' + d) << endl;
    return d;
}

float e (float f)
{
    cout << "in e, f = " << f << endl;
    cout << "in e, c = " << c (int(f)) << endl;
    return f;
}

int main ()
{
    cout << "Calling a('x') from main\n";
    a ('x');
    cout << "Calling c(12) from main\n";
    c (12);
    cout << "Calling e(4.3) from main\n";
    e (4.3);
    return 0;
}
```

Non recursive example output



```
Calling a('x') from main
in a, b = x
Calling c(12) from main
in c, d = 12
in a, b = m
in c, a = m
Calling e(4.3) from main
in e, f = 4.3
in c, d = 4
in a, b = e
in c, a = e
in e, c = 4
```



Recursive example

```
int a (int b)
{
    if (b < 10)
    {
        cout << "in a, b = " << b << endl;
        a (b+1);
        cout << "in a, b = " << b << endl;
        a (b * 2);
        cout << "in a, b = " << b << endl;
    }
    return b;
}

int main ()
{
    cout << "Calling a(8) from main\n";
    a (8);
    cout << "Calling a(4) from main\n";
    a (4);
    return 0;
}
```



Recursive example output

```
Calling a(8) from main
in a, b = 8
in a, b = 9
in a, b = 9
in a, b = 9
in a, b = 8
in a, b = 8
in a, b = 8
```

```
Calling a(4) from main
in a, b = 4
in a, b = 5
in a, b = 6
in a, b = 7
in a, b = 8
in a, b = 9
in a, b = 9
in a, b = 9
in a, b = 8
in a, b = 8
in a, b = 7
in a, b = 7
in a, b = 6
in a, b = 6
in a, b = 5
in a, b = 5
in a, b = 4
in a, b = 8
in a, b = 9
in a, b = 9
in a, b = 9
in a, b = 8
in a, b = 8
in a, b = 8
in a, b = 4
```

Nested subprograms example (C)



```
int main ()
{
    int a = 1, b = 2, c = 3;

    void nsp1 (int a)
    {
        void nsp2 (int b)
        {
            printf ("nsp2 : a = %d; b = %d; c = %d\n", a, b, c);
        }

        printf ("nsp1 : a = %d; b = %d; c = %d\n", a, b, c);
        nsp2 (6);
    }

    void nsp3 (int c)
    {
        printf ("nsp3 : a = %d; b = %d; c = %d\n", a, b, c);
        nsp1 (7);
    }

    printf ("main : a = %d; b = %d; c = %d\n", a, b, c);
    nsp1 (4);
    nsp3 (5);
    return 0;
}
```

Nested subprograms example output



```
main : a = 1; b = 2; c = 3
nsp1 : a = 4; b = 2; c = 3
nsp2 : a = 4; b = 6; c = 3
nsp3 : a = 1; b = 2; c = 5
nsp1 : a = 7; b = 2; c = 3
nsp2 : a = 7; b = 6; c = 3
```

Blocks example (C++)



```
int main (int argc, char * argv [1])
{
    int value = argc < 2 ? 5 : atoi (argv[1]);
    int total = 0;
    for (int i = 0; i < value; i++)
    {
        int subtot = 0;
        for (int j = 0; j < i; j++)
            subtot += j;
        total += subtot;
    }
    cout << "Total = " << total << endl;
    return 0;
}
```

Blocks example output



```
[tiawatts@cwolf server ~/cs460f08]$ a.out
Total = 10
[tiawatts@cwolf server ~/cs460f08]$ a.out 10
Total = 120
[tiawatts@cwolf server ~/cs460f08]$ a.out 30
Total = 4060
[tiawatts@cwolf server ~/cs460f08]$ a.out 101
Total = 166650
```

Subprogram issues summary



- Definition and call
- Alternative definitions
- Parameter passing modes
- Return values
- Call stack and activation records
- Scoping