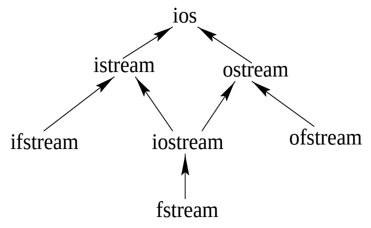
CSC 181F Lecture Notes

C++ I/O

- C++ consider I/O as *streams* of bytes.
  C++ file I/O is based on three stream classes: *istream*, *ostream*, and *iostream* classes.



- The system defines the objects *cin* (standard input), *cout* (standard output), *cerr* (standard error) in `<iostream.h>` .
  ifstrem, *ofstream*, *fstream* classes are the file version of the stream classes and defined in `<fstream.h>`.

Ostream Methods

ostream & put(**char** c);

ostream & write(**const signed char** *p, **int** m);

- `put` method displays the character c and returns the reference to the invoked object.
- `write` method displays the first m characters of the string p and returns the reference to the invoked object.

Example:

```
cout.put('A');
cout.put('A').put('B');
```


Formatting with cout

- Output format is specified either using **manipulators** or using setf() method in `ios` class to set format flag members
- Manipulator is a function that modifies a stream.
- The following manipulators can be used by including `iostream.h`:

endl	write newline and flush stream
ends	write null terminator in string
flush	flush output stream
dec	write integers in decimal
oct	write integers in octal
hex	write integers in hexadecimal

- ostream class overloads the insertion operator << such that the following two statements are equivalent: flush(cout); cout << flush;

Some Format Flags[a]

left	left justify
right	right justify
showpoint	print trailing zeros in floating-point numbers
scientific	use scientific notation for floating-point numbers
fixed	use fixed notation for floating-point numbers

[a] See C++ reference manual for more format flags

ios Class Methods to Manipulate Format Flags[a]

setf(**long**)	set specified flags
setf(ios::left, ios::adjustfield)	set left justification
setf(ios::scientific, ios::floatfield)	set scientific notation
unsetf(**long**)	clear specified flags
width(**int**)	change the field width

Examples:

```
cout.setf( ios::scientific, ios::floatfield );
cout << x << endl;
```

[a] See C++ reference manual for more format flags

Input with cin

- Input stream state may be determined with the following `ios` member functions:

good()	returns true if the stream can be used
eof()	returns true if the end of file has been reached
fail()	returns true if an input operation failed to read the expected characters or an output operation failed to write the expected characters
bad()	returns true if the stream may be corrupted.

- **if** or **while** test such as

  while (cin >> in)

  is true if the stream state is good, *i.e.,* cin.good() is true.

Example:

```
while ( cin >> in ) {
    ...
}
if ( cin.eof() ) {
...
```


istream Class Methods (Single Character Input)

istream & get(**char** & c);

int get(**void**);

- Both methods provide singe character input that doesn't skip over white space.
- With the first form, the input character is stored in c. The function return value at end-of-file is 0.

  cin.get(c);
- The second form returns the code for input character as type **int** value. The function return value at end-of-file is EOF.

  c = cin.get();

  cin.get(c).get();

istream Class Methods (String Input)

istream & get(**char** *a, **int** m, **char** c = '\n');

istream & getline(**char** *a, **int** m, **char** c = '\n');

istream & ignore(**int** m = 1, **char** c = EOF);

- With get and getline methods, characters are read from the stream into the array `a` until the character `c` is encountered or until `m-1` characters have been read into `c`, whichever happens first.
- `get` leaves the newline character in the input stream.
- `getline` extracts and discards the newline character from the input stream.
- `ignore` discards the next `m` characters or up through the character `c`, whichever comes first.

Example:

```
cin.getline(str, 80);
cin.ignore( 80 );
```


File I/O

- Basic file output operations
  - Create an ofstream object
  - Associate that object with a file

  Example:

  ```
  ofstream myFile("testFile");
  myFile << "Dull Data";
  myFile.close();
  ```
- Basic file input operations
  - Create an ifstream object
  - Associate that object with a file
- Include the header file `<fstream.h>` for all file I/O operations.

File Modes

- File mode describes how a file is to be used.
- When an object is associated with a file, the mode can be specified as a second argument. Or the mode can be specified using open() method.

  ```
  ifstream myFile1("testFile1", mode1);
  ofstream myFile2();
  myFile2.open("testFile2", mode2);
  ```
- File modes

ios::out	open file for writing
ios::in	open file for reading
ios::app	append to end-of-file
ios::trunc	truncate file if it exists

Unformatted File I/O

ofstream & write(**char** *s, **int** n);

ifstream & read(**char** *s, **int** n);

- `write` method writes `n` characters from the array `s` to the output file as raw bytes
- `read` method reads `n` characters (stored as raw bytes) into the array `s`.

Example:

```
int A[100];
ifstream inFile("dataFile");
inFile( (char *) A, sizeof(A) );
```

Writing Efficient Programs

- **Correctness is more important than efficiency!**
- Readability and maintainability are *very* important for large programs.
- **A more efficient Algorithm is the first choice for improving efficiency.**
  Reducing the order of the algorithm (e.g. from N^2 to $Nlog(N)$)
  helps more than any code tuning. After that, program tuning helps.
- Use the 10/90 Rule for optimizing programs

  Typically 10% of the source code in a program is responsible
  for 90% of the program's execution time

  This 10% of the program is the only part worth optimizing.

Timing Approximations

- Most arithmetic, comparison and logical operators take unit time.
  Exponentiation (and often division) are usually slower.
  Floating point arithmetic is usually slower than integer arithmetic.
- Assignments of scalars take unit time. Assignments of larger data structures (e.g. strings, records, arrays) take time proportional to the size of the data item.
  Array subscripting and pointer dereferencing take 2 time units.
- **if** statements take time that is the sum of the time for the controlling expression and the time for the part of the **if** that is executed.
  Loops take time which is proportional to the time to execute the body of the loop multiplied by the number of loop iterations.
- Calls to functions and procedures take about 10 time units plus 4 time units for each parameter. Calls to builtin functions take hundreds to thousands of time units

Program Improvement Strategies

- Space for Time Tradeoffs
  - Add extra data structures, accelerators for common cases
  - Store precomputed results, const table lookup for easy cases
  - Cache recent results
  - Use lazy evaluation
- Time for Space Tradeoffs
  - Pack data
  - Interpreters
  - Recompute large results
  - Expand procedure & function calls inline

Program Improvement

- Move loop independent code out of loops
- Combine tests. Eliminate duplicate tests, e.g search with end marker
- Unroll and/or fuse loops
- Use algebra, e.g. sqrt(x) $>$ 0 vs. x $>$ 0
- Reorder tests, put most decisive test first.
- Replace functions/expressions with table lookup.
- Eliminate useless expressions and assignments.
- *In general try to reorganize the program so that it computes the same result but does less work to do so*

Example
Matrix Multiply

Assume C_{nm}, A_{np}, B_{pm} then

$$C_{ij} = \sum_{i\equiv 1}^{k} A_{ik} \cdot B_{kj}$$

```
/* CSC 180F */
for ( I = 0 ; I < N ; I+ + )
    for ( J = 0 ; J < M ; J+ + ) {
        C[I][J] = 0.0 ;
        for ( K = 0 ; K < P ; K+ + )
            C[I][J] = C[I][J]) + A[I][K] * B[K][J] ;
    }
```


Example
Matrix Multiply

Assume C_{nm}, A_{np}, B_{pm} then

$$C_{ij} = \sum_{i\equiv 1}^{k} A_{ik} \cdot B_{kj}$$

```
/* CSC 181F */
for ( I = 0 ; I < N ; I+ + )
    for ( J = 0 ; J < M ; J+ + ) {
        double sum = 0.0 ;
        for ( K = 0 ; K < P ; K+ + )
            sum += A[I][K] * B[K][J] ;
        C[I][J] = sum ;
    }
```


Example
Least Squares Coefficients

```
/* CSC 180F */
void leastSquares(float X[] , float Y[] , , int N , float * A , float * B) {
    float C[ 6 ] ;
    int I , J ;
    for ( I = 0 ; I < 5 ; I += 1 )
        C[ I ] = 0.0 ;
    /* Calculate LSQ Coeffients C[0] .. C[5] */
    for ( I = 0 ; I < N ; I += 1 )
        for( J = 0 ; J < 5 ; J+ + )
            switch ( J ) {
            case 0 : C[ J ] = C[ J ] + X[ I ] ; break ;
            case 1 : C[ J ] = N ; break ;
            case 2 : C[ J ] = C[ J ] + Y[ I ] ; break ;
            case 3 : C[ J ] = C[ J ] + pow(X[ I ] , 2 ) ; break ;
            case 4 : C[ J ] = C[ J ] + X[ I ] ; break ;
            case 5 : C[ J ] = C[ J ] + X[ I ]*Y[ I ] ; break ;
            } ;
    /* Rest of least squares calculation here */
}
```


Example
Least Squares Coefficients

```
/* CSC 180F */
void leastSquares(float X[] , float Y[] , int N , float * A , float * B) {
    float C[ 6 ] ;
    int I , J ;
    for ( I = 0 ; I < 5 ; I += 1 )
        C[ I ] = 0.0 ;
    /* Calculate LSQ Coeffients C[0] .. C[5] */
    for ( I = 0 ; I < N ; I += 1 ) {
        C[ 0 ] = C[ 0 ] + X[ I ] ;
        C[ 1 ] = N ;
        C[ 2 ] = C[ 2 ] + Y[ I ] ;
        C[ 3 ] = C[ 3 ] + pow(X[ I ] , 2 ) ;
        C[ 4 ] = C[ 4 ] + X[ I ] ;
        C[ 5 ] = C[ 5 ] + X[ I ]*Y[ I ] ;
    } ;
    /* Rest of least squares calculation here */
}
```

Example
Least Squares Coefficients

```
/* CSC 181F */
void leastSquares(float X[] , float Y[] , int N , float * A , float * B) {
    float C0 = 0.0 , C1 = N , C2 = 0.0 , C3 = 0.0 , C4 = 0.0 , C5 = 0.0
    /* Calculate LSQ Coeffients c0, c2..c5 */
    for ( I = 0 ; I < N ; I += 1 ) {
        register float XI = X[ I ] ;
        register float YI = Y[ I ] ;
        C1 += XI ;
        C3 += YI ;
        C4 += XI * XI ;
        C6 += XI * YI ;
    }
    C5 = C1 ;
    /* Rest of least squares calculation here */
}
```


Example - Table Search

- Search array X containing N real numbers for value T
  If T = X(I) result is I otherwise result is -1
- **#define** N ..
  float X[N] ;
  float T ;
- Performance measures
  - Execution Time in uSec
  - Time for 1,000,000 searches with
    - N = 100, 1000, 10000
- Evolving example to show effect of code improvements.

L1 — Naive Linear Search

```
J = 0 ;
while ( J < N && X[ J ] != T )
    J++ ;
if ( J < N )
    return J ;
else
    return -1 ;
```

	Time	N=100	N=1,000	N=10,000
L1	$3.65 \cdot N$	6 mins	1 hrs	10 hrs

L2
Improved Linear Search — Stop on End Sentinel

```
/* Extend array to N+1 elements */
float X[ N+1 ] ;
X[ N ] = T ;
J = 0 ;
while ( X[ J ] != T )
    J++ ;
if ( J < N )
    return J ;
else
    return -1 ;
```

	Time	N=100	N=1,000	N=10,000
L1	$3.65 \cdot N$	6 mins	1 hrs	10 hrs
L2	$2.05 \cdot N$	3.5 mins	33 mins	5.5 hrs

B1

Binary Search — Written as in Basic

```
float binSearch( float T , float L , float U , FLOAT X[] ) {
    if (L > U )
        return -1 ;
    else
        if ( T == X[floor((L+U)/2.0]) )
            return floor((L+U)/2.0) ;
        else if (T < X(floor((L+U)/2.0)))
                return binSearch(T,L,(L+U)/2.0-1.0, X) ;
            else
                return binSearch(T,(L+U)/2.0+1.0,U,X) ;
}
```

	Time	N=100	N=1,000	N=10,000
L2	$2.05 \cdot N$	3.5 mins	33 mins	5.5 hrs
B1	$256 \cdot \log_2 N$	28 mins	42 mins	57 mins

B2

Basic-like — Eliminate redundant calculations

```
float binSearch( float T , float L , float U , FLOAT X[] ) {
    if ( L > U )
        return -1 ;
    else
        const float M = floor((L+U)/2.0) ;
        if ( T == X[ M ] )
            return M ;
        else if ( T < X[ M ] )
                return binSearch(T, L, M - 1.0, X) ;
            else
                return binSearch(T, M + 1.0, U, X) ;
}
```

	Time	N=100	N=1,000	N=10,000
B1	$256 \cdot \log_2 N$	28 mins	42 mins	57 mins
B2	$101 \cdot \log_2 N$	11 mins	16.5 mins	22 mins

B3

Improved Binary Search — Use integer indices as in C

```
int binSearch( float T , int L , int U , FLOAT X[] ) {
    if ( L > U )
        return -1 ;
    else
        const int M = (L+U) / 2 ;
        if ( T = X[ M ] )
            return M ;
        else if ( T < X[ M ] )
                return binSearch(T, L, M - 1, X) ;
            else
                return binSearch(T, M + 1, U, X) ;
}
```

	Time	N=100	N=1,000	N=10,000
B2	$101 \cdot \log_2 N$	11 mins	16.5 mins	22 mins
B3	$23 \cdot \log_2 N$	2.5 mins	3.8 mins	5 mins

B4

Binary Search — Use interation instead of recursion

```
int binSearch( float T , int L , int U , FLOAT X[] ) {
    int M ;
    while ( 1 ) {
        if ( L > U )
            return -1 ;
        M = (L + U) / 2 ;
        if ( T = X[ M ] )
            return M ;
        else if ( T < X[ M ] )
                U = M - 1 ;
            else
                L = M + 1 ;
    }
}
```

	Time	N=100	N=1,000	N=10,000
B3	$23 \cdot \log_2 N$	2.5 mins	3.8 mins	5 mins
B4	$16 \cdot \log_2 N$	1.7 mins	2.7 mins	3.5 mins

B5

Binary Search — Iterative, Reorder tests

```
int binSearch( float T , int L , int U , FLOAT X[] ) {
    int M ;
    while ( 1 ) {
        if lL > lU
            return -1 ;
        M = (L + U) / 2 ;
        if ( T < X[ M ] )
            U = M - 1 ;
        else if ( T > X[ M ] )
                L = M + 1 ;
            else
                return M ;
    }
}
```

	Time	N=100	N=1,000	N=10,000
B4	$16 \cdot \log_2 N$	1.7 mins	2.7 mins	3.5 mins
B5	$15 \cdot \log_2 N$	1.6 mins	2.5 mins	3.3 mins

H1

Hard Core Binary Search, N = 1000

```
int binSearch( float T , int L , int U , FLOAT X[] ) {
    int K = 0 ;
    if ( X[ 512 ] <= T )        K = 488 ;        /* 488 = 999 - 512 + 1 */
    if ( X[ K +256 ] <= T )     K += 256 ;
    if ( X[ K +128 ] <= T )     K += 128 ;
    if ( X[ K +64 ] <= T )      K += 64 ;
    if ( X[ K +32 ] <= T )      K += 32 ;
    if ( X[ K +16 ] <= T )      K += 16 ;
    if ( X[ K +8 ] <= T )       K += 8 ;
    if ( X[ K +4 ] <= T )       K += 4 ;
    if ( X[ K +2 ] <= T )       K += 2 ;
    if ( X[ K +1 ] <= T )       K += 1 ;
    return ( X[ K ] == T   ?  K  :  -1 ) ;
}
```

	Time	N=100	N=1,000	N=10,000
B5	$15 \cdot \log_2 N$	1.6 mins	2.5 mins	3.3 mins
H1	$5 \cdot \log_2 N$	30 secs	50 secs	1.1 mins
ASM	$0.9 \cdot \log_2 N$	6 secs	9 secs	12 secs

Perfomance Summary

Execution Time and Time for 1,000,000 Searches

	Time	N=100	N=1,000	N=10,000
L1	$3.65 \cdot N$	6 min	1 hr	10 hr
L2	$2.05 \cdot N$	3.5 min	33 min	5.5 hr
B1	$256 \cdot \log_2 N$	28 min	42 min	57 min
B2	$101 \cdot \log_2 N$	11 min	16.5 min	22 min
B3	$23 \cdot \log_2 N$	2.5 min	3.8 min	5 min
B4	$16 \cdot \log_2 N$	1.7 min	2.7 min	3.5 min
B5	$15 \cdot \log_2 N$	1.6 min	2.5 min	3.3 min
H1	$5 \cdot \log_2 N$	30 sec	50 sec	1.1 min
ASM	$0.9 \cdot \log_2 N$	6 sec	9 sec	12 sec

C Optimization Example

```
float  A[ M ][ P ], B[ P ][ N ], C[ M ][ N ] ;
int  i, j, k ;
/* Naive Algorithm  */
for( i = 0 ; i < M ; i++ )
   for( j = 0 ; j < N ; j++ ) {
        C[i][j] = 0 ;
        for( k = 0 ; k < P ; k++ )
            C[i][j] = C[i][j] + A[i][k] * B[k][j] ;
   }
```

cc	cc -O	gcc	gcc -O2
100 instructions	25 instructions	94 instructions	18 instructions
74 inner loop	20 inner loop	90 inner loop	12 inner loop

C Optimization Example

```
/*  C Hacking Algorithm (Don't Do This at Home)*/
float  A[ M ][ P ], B[ P ][ N ], C[ M ][ N ] ;
register float  *ap, *bp, *dp, *aStart, *aEnd ;
register float  *bEnd, *bStart, T , *aStop   ;
for( aStart = &A[0][0] , aEnd = &A[M][0] , dp =&D[0][0] ;
            aStart < aEnd  ;  aStart += P  ) {
    aStop = aStart + P ;
    for( bStart = &B[0][0] , bEnd = &B[0][N] ;
           bStart < bEnd  ;  bStart++ ) {
        for( ap = aStart, bp = bStart , T = 0.0 ,
              ; ap < aStop ; bp += N )
              T += *ap++ * *bp ;
        *dp++ = T ;
    }
}
```

cc	cc -O	gcc	gcc -O2
48 instructions	16 instructions	25 instructions	14 instructions
26 inner loop	12 inner loop	18 inner loop	8 inner loop