CSC 181F Lecture Notes

Reading Assignment

`K.N. King` `Chapter 14`

`Section 26.1`

C Preprocessor

- Processes program text *before* compiler
- Implements
  - Source Include mechanism
  - Conditional compilation
  - Macro Definition and Use

Source inclusion

- Used to decompose large programs into manageable pieces
- Most common usage is:
  - `foo.h` - interface file
  - `foo.c` - implementation
  - `foo.o` - compiled implementation
- Interface file contains:
  - function headers
  - shared definitions
    - structs, constants,types, macros
  - *Avoid shared variables*
- Programs that use foo include `foo.h` to get interface and get linked with `foo.o` to get the implementation

Example Interface File

```
/* File house.h - Interface to house.c */

/* Hide real representation inside house.c */

#define HOUSE_HEIGHT   2.0
#define HOUSE_WIDTH    3.0
#define ATTIC_HEIGHT   0.7

. . .
extern void DrawHouse(double x , double y) ;
extern void DrawOutline(double x , double y) ;
extern void DrawWindows(double x , double y) ;
extern void DrawDoor(double x , double y) ;
extern void DrawBox(double x , double y , double width , double height) ;
extern void DrawTriangle(double x , double y , double base , double height) ;
extern void DrawCenteredCircle(double x , double y , double r) ;
. . .
```


#include

#include "fileName"

#include < systemFile >

- *fileName* is the name of the file to include.
  Compiler searches through a standard list of directories looking for it.
  Usually start with current directory.
- The second form specifies a system interface file.
  The compilers search starts in the directory `/usr/include`
- **Good Technique:** Use only the first form of include. The compiler will still find system files, but you can customize include files if you have to.

Macros

- A *macro* is a piece of *text* that is used repeatedly in a program.
- Use macros for:
  - defining literal constants
  - defining short pieces of ugly code
  - hiding details from the user
- Macros affect the **source program text just before** the program is compiled.

Macro Definition

#define *macroName* text

macroName is the name of the macro

Good Technique: use UPPERCASE NAMES for macros to make them stand out a program that uses them

The text (if any) on the same line following the macro name becomes the definition of the macro.

Use the backslash character (\ ) at the end of a line to extend long definitions.

Examples:

```
#define  A_SIZE  ( 15 )
#define PROMPT ("Hi:" )
```

Simple Macro Examples

```
#define BUFSIZE   ( 1024 )
#define NULL  ( (void * ) 0 )

#define TRUE   ( 1 )
#define FALSE   ( 0 )

#define TWO_PI  (2*3.14159)

#define RESET { i = j = k = 0  ;  }
#define POLYX ( x * (x * (3.14 * x + 2.78) + 1.35) - 2.16 )

#define CLEARA { int  i ;  for ( i=0 ; i < N ;  ) \
          A[i++] = 0 ;  }
```


Macro Parameters

- Macros can be defined with simple **text** parameters that are substituted as directed in the body of the macro
- Parameters are identifiers listed in parentheses after the macro name
- Use this form for declarations or code fragments with substitutable parts
- **Parameter substitution is text substitution**

Examples Macro with Parameters

#define macroName(macroParmList) text

macroParmList is a list of identifiers separated by commas

Examples:

```
#define  SCALE(x)   ((x)*10)
#define  MIN(x,y)   ( (x) < (y) ? (x) : (y) )
#define  POLY(x)  ( (x)*((x)*(3.14*(x)+2.78)+1.35)-2.16 )
#define  FILLARRAY( arr , val , n)  \
      { int i ;  for (i=0 ; i < (n) ; )   arr[i++] = val; }
```

Examples of use:

```
j = SCALE( i + 1 ) ;
if ( MIN( i ,  j ) == 0 ) ...
y = POLY( y+7.0 )  ;
FILLARRAY( myData ,  100 ,  3.14 ) ;
```


Macro Use

- For macros without parameters, just write name of macro in program, C preprocessor substitutes macro body for name
- For macros with parameters, follow macro name with list of macro parameters enclosed in parentheses. The **text** of each parameter is substituted into the macro body
- Examples:

```
char  buf1[ BUFSIZE ] ;
int  allDone =  FALSE ;
float  circumference = TWO_PI * r ;
i = MAX( 1 , i  ) ;
SWAP( A[k] , A[k+1] );
STOPHERE  ;
```

HOW TO Use Macros

- Use macros for program parameterization and as an abstraction tool.
  Don't use it to make programs unreadable or obscure.
  Make body of macro a complete expression or statement
- Macros with expression parameters should be designed to work with **any** valid expression passed as an argument.
- **Good Technique:** Wrap parameter names in parentheses to avoid unexpected expansions and to force error message for invalid parameters
- **Good Technique:** Macros that look like statements should behave like statements. Wrap body in { and }. Do not put `;` at end.
- The truely paranoid will use each macro parameter exactly once.

More Macro Examples

```
#define DATARRAY(name,size)  float name[size]
#define MAX( a, b )   ((a) > (b) ? (a) : (b) )
#define SWAP( x, y )  { int t =(x); x = (y); y = t }
#define CRASH_AND_BURN( why )   \
    { fprintf(stderr, "Bailing out of program due to error %s\n", why ); \
      abort(); \
    }
#define MALLOC( size, type , pointer ) \
    { void * mTemp ; \
        mTemp = malloc( (size) ) ; \
        assert ( mTemp ) ; \
        pointer = ( type ) mTemp ; \
    }
```

Note use of ( ) around expressions and { } around statements.
mTemp is used in MALLOC to achieve the *one-touch* property.

Conditional Compilation

- Used to selectively include or exclude parts of a program
- Used for:
  Optional code (e.g. debugging)
  Machine customization
  Operating system customization
- Select based on:
  defined and undefined macro names
  compiler flags `-Dname , -Dname=value , -Uname`
  compile time expressions, can use most C operators
- Conditionals may be nested

Preprocessor Constant Expressions

- The conditional expression that the preprocessor can evaluate are made up of
  - Integer constants defined using **#define** .
  - Integer constants defined using the compiler option `-Dname=constant`
  - use of the *defined(identifier)* function which has the value one if the identifier has been defined in the preprocessor at that point.
  - Almost all C arithmetic and logical operators. Logical expressions can be used to define complicated conditions.
  - Symbols predefined by the compiler, e.g. `__i486__` , `__GNUC__`
    Generally these symbols identify the hardware, the compiler and the operating system.
    Use the command `gcc -dM -E` to see your definitions.

#if head

#if constant-expression
#ifdef identifier
#ifndef identifier

The three forms of if head listed above are used at the start of a preprocessor conditional statement.

the first form is true if the constant (i.e. compile time) expression evaluates to non-zero.
This expression can include use of the `defined`(identifier) predicate

Second form is true if identifier has been defined using **#define** or by the compiler `-D` option.

Third form is true if identifier has **not** been defined using **#define** or it has been undefined using the compiler `-U` option.

Preprocessor Conditional Statement

if-head
  text
#elif constant-expression
  text
#else
  text
#endif

text is any program text. It may be arbitrarily long. It may contain nested conditionals.

If *if-head* evaluates to true, text is included in the program

The elif part may be repeated as many times as required. The else part is optional. If it appears, then the text following the **#else** is included in the program if none of the preceding if or elifs have evaluated to true.

#if examples

```
#if X == 3
    int data3[ 100 ] ;
#else
    int data2[ 50 ], data1[ 200 ] ;
#endif
#ifdef DEBUG
    fprintf(stderr, "Made it as far as Checkpoint Charlie\n");
#endif
#ifndef ONCE
    #define ONCE
    ...
#endif
#if defined(UNIX) || defined(_unix_)
    #include < stdio.h>
#elif defined(VMS)
    #include < VMSstdio.h>
#else
    #include "D:/SYS/INCLUDE/STDIO.H"
#endif
```


HOW TO USE #if *et.al.*

- KISS
- Use conditional compilation sparingly to customize your program for different environments.
- Use indentation and paragraphing to indicate matching **#if** , **#else** and **#endif**.
- Complicated **#if** structures are a symptom of bad program design.
- As with any use of conditionals, make sure all cases are covered and each logical expression does what you expect.
- KISS

Functions with a Variable Number of Arguments

- There is a mechanism in C that allows you to write your own functions that take a variable number of arguments like printf and scanf.
- The include file `stdarg.h` defines three macros

  void va_start(va_list ap , *parmN*) **;**

  type-name va_arg(va_list ap, *type-name*) **;**

  void va_end(va_list ap) **;**

  that can be used to access variable length argument lists in a safe and portable way.
- A function that takes a variable number of arguments must have at least one named argument. An *ellipsis* `( . . . )` is used to tell the compiler that the function takes an arbitrary number of arguments. Example

  int doList(int first , . . .) **;**

HOW TO Use Variable Length Argument Lists

- Declare a variable of type `va_list` to hold an index into the variable length argument list while it is being processed. Example

  `va_list ap ;`
- The variable argument list comes after the last named parameter. Call `va_start` to initialize the argument pointer *ap* to the first variable argument

  `va_start( ap, first ) ;`
- The function `va_arg` is used to fetch a varialble argument *and* advance to the next argument in the list.

  `argVal = va_arg( ap ,` *type-name* `) ;`

  where *type-name* is the type of the variable (argVal) being assigned to.
- Call the function `va_end` to clean up after argument processing.

  `va_end( ap ) ;`
- You can cycle through the argument list more than once by calling `va_start` to reinitialize the argument pointer. You can use more than one argment pointer on the same list.

Which Machine for You?

Problem Size	Machine A	Machine B
10	3.0 microseconds	200 milliseconds
100	3.0 milliseconds	2.0 seconds
1,000	3.0 seconds	20 seconds
10,000	49 minutes	3.2 minutes
100,000	35 days	32 minutes
1,000,000	95 years	5.4 hours

Come to Lecture and Find Out

Evaluating Program Performance[a]
How Fast Does it Run?

- Cost of executing a program can be measured in terms of the amount of time and space the program uses.
- Often there is a tradeoff between space and time
- It is very useful to be able to estimate the time and/or space used by a program in terms of the size of the input it must process

[a] Adapted from: F. Fich & D. Horton, CSC148F Lecture Notes, 1993

Measuring Time

- Run program on various inputs and measure how long it takes
- Usually unsatisfactory, can depend on things external to the program like other users on the computer
- Alternative, count number of (certain) operations the program performs
- Examples: assignment, arithmetic, comparisons, array access, pointer access, branching
- We want to characterize the running time of a program as a function of the size of its input.

Running Time Functions

	Approximate Value of T(n) for n =				
T(n)	10	100	1000	10,000	100,000
log_e(n)	3	6	9	13	16
log_e^2(n)	9	36	81	169	256
$\sqrt{n}$	3	10	31	100	316
n	10	100	1000	10,000	100,000
$nlog_e(n)$	30	600	9000	$13 \cdot 10^4$	$16 \cdot 10^5$
n^2	100	10,000	10^6	10^8	10^{10}
n^3	1000	10^6	10^9	10^{12}	10^{15}
2^n	1024	10^{30}	10^{300}	$10^{3,000}$	$10^{30,000}$

Growth Rate of various functions

Big O Notation

- f(n) and g(n) be functions defined on the non-negative integers such that f(n) >= 0 and g(n) >= 0 for all n
- A function f(n) is *order g(n)* if there exists positive constants c and B such that
  f(n) <= c · g(n) for all n >= B
- Written as "f(n) is O(g(n))"
- Constant factors and lower order terms are ignored
- Examples:

  $a_2n^2 + a_1n + a_0$ is O(n^2)
  $a_3n^3 + nlog_e(n)$ is O(n^3)

O Algebra

if f(n) is O (g(n)) and g(n) is O (h(n))
  f(n) is O (h(n))

if f_1(n) is O (g_1(n)) and f_2(n) is O (g_2(n))
  f_1(n) + f_2(n) is O (g_1(n) + g_2(n))

if f_1(n) is O (g_1(n)) and f_2(n) is O (g_2(n))
  f_1(n) · f_2(n) is O (g_1(n) · g_2(n))

f(n) + g(n) is O (max(f(n), g(n)))

Analyzing Running Time

- Code without loops or procedure calls takes O (1) time
- If the body of a **for** loop is executed O (f(n)) times and each iteration takes O (g(n)) time, then the entire loop takes

  O (1 + f(n) · g(n)) time
- if the true and false parts of an **if** statement take O (f(n)) and O (g(n)) time respectively then the if statement takes

  O (max(f(n),g(n))) time

Examples

```
for ( sum = 0 , I = 0 ; I < N / 2 ; I++ )     O(n)
    for ( J = 0 ; J < I ; J++ )               O(n)
        sum += 1 ;                            O(1)
---------------------------------------------------
for ( sum = 0 , I = 0 ; I <= N / 2 ; I++ )    O(n)
    for ( J =0 ; J < N * N ; J++ )            O(n^2)
        sum += 1 ;                            O(1)
---------------------------------------------------
if ( I <= 10 )                        O(1)
    smallData = 1 ;                   O(1)
else
    for ( J = 0 ; J < N ; J++ )       O(n)
        data[ J ] = J ;               O(1)
```


Tables and Searching

- Table - a collection of key, object pairs
- Keys must uniquely identify objects
- Typical operations on tables:

  insert - 10 %

  delete - 5 %

  search - 85 %
- Implement table as an array of structures or as a dynamically allocated list structure

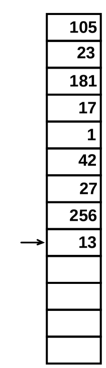
Linear Table Management

- Implement Table as array and pointer

  Store elements in insertion order

  Linear search for key value
- Performance

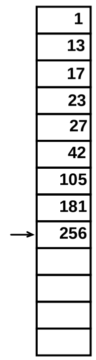
Operation	Method	Time
insert	at end	1
delete	remove & compress	N
lookup	linear search	N



Sorted Table Management

- Implement Table as array and pointer
  Store elements *sorted in order* by key
  Binary search[a] for key value
- Performance

Operation	Method	Time
insert	in order	$N + log_2(N)$
delete	remove & compress	$N + log_2(N)$
lookup	binary search	$log_2(N)$

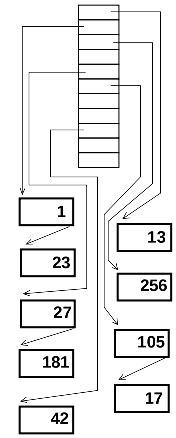


[a] (See Slides 193 & 194)

Hash Table Management[a]

- Implement Table as array and pointer
  Store elements indexed by hash(key)
  Use hash(key) and chaining for search
- Performance

Operation	Method	Time
insert	by hashing	1 normally
		N worst case
delete	by delinking	1 normally
		N worst case
lookup	hash & chain	$1 + N/tableSize$
		N worst case



[a] Hash Tables will be discussed in detail in CSC190S and CSC191S.

Hash Functions

- Maps table key into table index.
  Usually a many-to-one mapping
- Should be easy to compute
- Should spread keys as uniformly as possible across all table entries
- Not worth huge effort to improve hash function
- *Perfect* hash functions are possible in restricted cases
- **Good Technique:** Compute some function of the key and use the modulus operator (%) to reduce the value of the function to the range
  0 **..** TABLE_SIZE - 1

Hash Function Example

- For words use:
  - First letter
  - First letter + length
  - First letter & last letter
- Combine letters using arithmetic or bit operations
  Reduce modulo table size
- Example:

Word	first	first + length	first + last	first ˆ last
TAHIR	84	89	166	6
TERSIGNI	84	92	157	29
VANDERBY	86	94	175	15
VARODAYAN	86	95	164	24

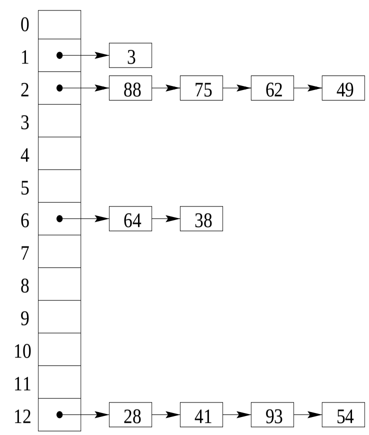
Hash Function Example

Suppose we were to store employee records, where each employee has a number n.

Given a table of size 13, use the the following function (*hash function*) to store the records:

$h(n) = (2n - 5)\%13.$

n	$h(n)$
88	2
75	2
64	6
28	12
41	12
38	6
62	2
3	1
49	2
93	12
54	12



Data Structures in C

- Sets
- Sparse Arrays

Data structures is one of the major topics in CSC190S and CSC191S.

The following slides are a small preview.

Sets in C

- A set is a collection of arbitrary elements
- Primary issue is set membership
- Set operations include union ($\cup$), intersection ($\cap$), subset relation ($\subset$), membership relation (ε), member creation
- Sets can be represented in C in a number of ways depending on:
  - type and homogeneity of elements
  - bounded or unbounded size
  - relative frequency of operations

Word Sets in C

- Small sets with an integer base type can be represented in C using the 32 bits in an integer to indicate the presence or absence of a particular element
- Use C bit operations on integers (& , | , etc.) to implement set operations
- For a set with members in the range low .. high
  If low is non-zero, subtract it from all elements so all sets are represented internally as

  set of 0 .. high - low
- Number the bits in an integer from 0 (rightmost) to 31 (leftmost). Associate each bit with a particular element of the set.
- This representation of Sets is used in many implementations of Turing and Pascal.

Word Set Operations[a]

<table>
<tr><td>$\{\ \}$</td><td>0</td></tr>
<tr><td>$\{$ all $\}$</td><td>0xFFFFFFFF</td></tr>
<tr><td>$\{A\}$</td><td>(1 << (A - low))</td></tr>
<tr><td>$A \varepsilon B$</td><td>((A $<$ low) | | (A $>$ high) ? FALSE :</td></tr>
<tr><td></td><td>(1 << (A - low)) & B)</td></tr>
<tr><td>$B \cup C$</td><td>(B | C)</td></tr>
<tr><td>$B \cap C$</td><td>(B & C)</td></tr>
<tr><td>$B \subset C$</td><td>((B | C) == C) && ((B & C) != C)</td></tr>
<tr><td>$B \subseteq C$</td><td>((B | C) == C)</td></tr>
<tr><td>$B \setminus C$</td><td>(B ^ C)</td></tr>
</table>

[a] For a set of *low* .. *high* elements

Larger Word Sets

- For a set with elements *low* .. *high*
  where *high* - *low* is greater than 32, use an array of integers for the set
  typedef unsigned Set[(*high*-*low*+31) >> 5] **;**
- Use division and modulus to select word in array and bit within word
  For an arbitrary element A
  array index is (A - *low*) >> 5
  (usually faster than (A - *low*) / 32)
  word index is (A - *low*) & 0x1F
  (faster than (A - *low*) % 32)
- Use macros to parameterize access

```
#define getElement( SetVar, Elem ) \
    ( SetVar[((Elem)-Setlow)>> 5]  & (1<< (((Elem)-SetLow)&0x1F)) )
```


Sets of Arbitrary Elements

- To build sets of arbitrary (non-integer) elements (e.g. reals, *Things*) you need to actually store elements in the set.
- If membership/non-membership is the dominant operation, could use a hash table to represent a set. Could also use a hash table to store *Things* efficiently and store hash table indices in the sets.
- If maximum size of set is known, could represent them as arrays of *Things*
- If maximum size of set is not known, represent sets as linked list of elements.
- For a *Thing* to be an element of a set we need only a few operations on *Things* :
  - The ability to copy *Things* from one place to another
  - The ability to store *Things* in data structures
  - An equality relation (e.g. ==) that will determine if two *Things* are the same
  - Possibly an ordering relation (e.g. <=) on *Things*

Set Operations on Arbitrary Sets

- Assume a set of *Things*
- Assume an ordering relation on *Things*
- Store sets as ordered lists of *Things*
- Operations:

$\{A\}$	Add A to list in order
$A \varepsilon B$	Search list B for A
$B \cup C$	merge lists B and C, deleting one of each duplicate
$B \cap C$	merge lists B and C keeping only one of each duplicate
$B \subseteq C$	Use $(B \cap C) == B$

Sparse Arrays

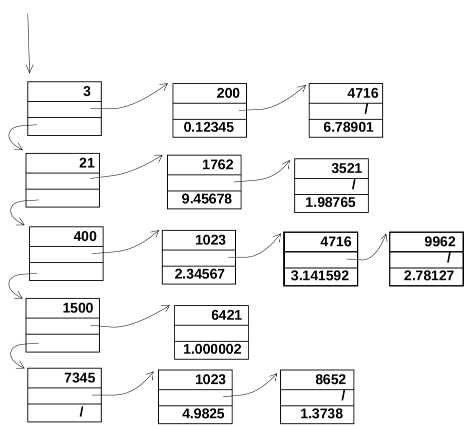
- An array is *sparse* if more than 99% of it's elements have the values zero
- Sparse arrays arise naturally in the solution of many numerical problems in science and engineering.
- Sparse arrays are also often very large (e.g. 10,000 X 10,000) so storing them in a space efficient fashion is an important issue.
- Operations on sparse arrays include access to individual elements and standard matrix operations such as matrix multiplication
- Most numerical algorithms that work on matrices can be adapted to work on sparse arrays

Storage for Sparse Arrays

- A hash table (hash on row/column index) could be used, but an unstructured hash table isn't convenient for operations like matrix multiplication. Most practical schemes are equivalent to some form of structured hash table
- Most representation schemes use some form of single or double indexing. Sparse array is stored in some form of linked list data structure
- Array elements are typically *self identifying*, i.e. the row and column index are stored with the element
- Two illustrative techniques: single and double indexing are described in the following slides. Linked lists are used for both row and columns, but a vector of pointers could be used if the size of the sparse array isn't too large
- Single link is similar to a hash table on the row index
  Double link is similar to overlaid row and column hash tables

Sparse Array - Single Link

```
typedef struct elem *
    colPtr ;
struct elem {
    int colNo ;
    colPtr nextCol ;
    double Val
} ;
typedef struct rows *
    rowPtr ;
struct rows {
    int rowNo ;
    colPtr column ;
    rowPtr nextRow
} ;
typedef rowPtr Sparse ;
```



Sparse Array - Double Link

```
typedef struct elem * elPtr ;
struct elem {
    int colNo , rowNo ;
    elPtr nextCol , nextRow ;
    double Val
} ;
typedef struct index * indPtr ;
struct index {
    int rowcolNo ;
    elPtr rowcol ;
    indPtr nextIndex
} ;
struct Sparse {
    indPtr rows ;
    indPtr cols ;
} ;
```