Pre Processor Directives, C Standard Library (string.h, math.h), searching and sorting

C - Course

Contents

- Pre Processor Directives
- C Standard Library
- Functions in string.h
- Functions in math.h
- Searching Algorithms
- Sorting Algorithms

Pre Processor Directives

- #include
- #define
- #if
- #ifdef
- #ifndef
- #else
- #elif
- #endif

- Used for Including Files, Conditional Compilation and Macro Definition

Including Files

```
#include <stdio.h>
int main (void)
{
    printf("Hello, world!\n");
    return 0;
}
```

Conditional Compilation

- The #if, #ifdef, #ifndef, #else, #elif and #endif directives can be used for conditional compilation.

- **Example 1**

```
#define __WINDOWS__
#ifdef __WINDOWS__
   #include <windows.h>
#else
   #include <unistd.h>
#endif
```

Conditional Compilation

Example 2

```
#define __DEBUG__
#ifdef __ DEBUG __
   printf("trace message");
#endif
```

Macro Definition and Expansion

Object Like:

#define <identifier> <replacement token list>

Example:

#define PI 3.14159

Function Like:

#define <identifier>(<parameter list>) <replacement token list>

Example:

#define RADTODEG(x) ((x) * 57.29578)

Macro Definition and Expansion

Function Like: Be careful!

Example:
#define MAC1(x) (x * 57.29578)
will expand MAC1(a + b)
to (a + b * 57.29578)

#define MIN(a,b) ((a)>(b)?(b):(a))
What happens when called as
MIN(++firstnum,secondnum) ?
firstnum will be incremented twice

Multi File Programs

- Why?
  - As the file grows, compilation time tends to grow, and for each little change, the whole program has to be re-compiled.
  - It is very hard, if not impossible, that several people will work on the same project together in this manner.
  - Managing your code becomes harder. Backing out erroneous changes becomes nearly impossible.
- Solution
  - split the source code into multiple files, each containing a set of closely-related functions

Multi File Programs

- Option 1
  - Say Program broken up into main.c A.c and B.c
  - If we define a function (or a variable) in one file, and try to access them from a second file, declare them as external symbols in that second file. This is done using the C "extern" keyword.
  - Compile as:
    gcc main.c A.c B.c -o prog
- Option 2
  - Use header files to define variables and function prototypes
  - Use #ifndef _*headerfile name* #define _ *headerfile name* and #endif to encapsulate the code in each Header file
  - Compile only the modified files as:

    gcc -c main.cc

    gcc -c A.c

    gcc -c B.c

    And then link as gcc main.o A.o B.o -o prog

Multi File Programs

- Which is better Option 1 or Option 2?

C Standard Library

- The **C standard library** (aka **libc**) is a standardized collection of **header** files and **library** routines, which are used to implement common operations, such as input/output and string handling etc.
- C does not have built in keywords for these tasks, so nearly all C programs rely on the standard library.

libc

- <assert.h>
- <ctype.h>
- <errno.h>
- <float.h>
- <limits.h>
- <locale.h>
- <math.h>
- <setjmp.h>
- <signal.h>
- <stdarg.h>
- <stddef.h>
- <stdio.h>
- <stdlib.h>
- <string.h>
- <time.h>

libc

- <**assert.h**> : Contains the assert macro, helpful in detecting logical errors and other types of bug in debugging versions
- <**ctype.h**> : to classify characters by their types or to convert between upper and lower case
- <**errno.h**> : For testing error codes
- <**float.h**> : Contains macros that expand to various limits and parameters of the standard floating-point types

libc

- <**limits.h**> : constants specifying the implementation-specific properties of the integer types
- <**locale.h**> : to set and select locale
- <**math.h**> : common math functions
- <**setjmp.h**> : macros setjmp and longjmp
- <**signal.h**> : various exceptional conditions
- <**stdarg.h**> : to allows functions to accept an variable number of arguments

libc

- <**stddef.h**> : some useful types and macros
- <**stdio.h**> : input/output functionaliteis
- <**stdlib.h**>: conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
- <**string.h**> : string manipulation and memory handling
- <**time.h**> : time/date formats and manipulation

string.h (memory handling)

- void* **memcpy** (void* dest, const void* src, size_t num)
- void* **memmove** (void* dest, const void* src, size_t num) /*works even when the objects overlap*/
- int **memcmp**(const void* buffer1, const void* buffer2, size_t num)
- void* **memchr** (const void* buffer, int c, size_t num)
- void* **memset** (void* buffer, int c, size_t num)

string.h (string manipulation)

- char* **strcpy** (char* *dest*, const char* *src*)
  copy src to dest including ‘\0’

- char* **strncpy** (char* *dest*, const char* *src*, sizet_t *num*)
  pad with ‘\0’s if src has fewer than num chars

- char* **strcat** (char* *dest*, const char* *src*)

- char* **strncat** (char* *dest*, const char* *src*, sizet_t *num*)

  concatenate at most num chars, terminate dest with ‘\0’ and return dest

string.h (string manipulation)

- int **strcmp** (const char* *string1*, const char* *string2*)
  returns <0 if string1<string2, 0 if string1==string2, or >0 if string1>string2

- int **strncmp** (const char* *string1*, const char* *string2*, sizet_t *num*)

- char* **strchr** (const char* *string*, int *c*)
  return pointer to first occurrence of c in string or NULL if not present.

- char* **strrchr** (const char* *string*, int *c*)
  last occurrence of *c* in *string*

string.h (string manipulation)

- size_t **strspn** (const char* *string1*, const char* *string2*)
  return length of prefix of string1 consisting of chars in string2

- size_t **strcspn** (const char* *string1*, const char* *string2*)
  return length of prefix of string1 consisting of chars not in string2

- char* **strpbrk** (const char* *string1*, const char* *string2*)
  return pointer to first occurrence in string1 of any character of string2, or NULL if none is present.

string.h (string manipulation)

- char* **strstr** (const char* *string1*, const char* *string2*)
  return pointer to first occurrence of string2 in string1, or NULL if not present

- size_t **strlen** (const char* *string*)

string.h (string manipulation)

- char* **strerror** (int errnum)

  Returns a pointer to a string with the error message corresponding to the *errnum* error number. Subsequent calls to this function will overwrite its content. This funcion can be called with the global variable, errno, declared in errno.h to get the last error produced by a call to a C library function.

string.h (string manipulation)

- char* **strtok** (const char* *string*, const char* *delimiters*)
  If *string* is not NULL, the function scans *string* for the first occurrence of any character included in *delimiters*. If a member of *delimiters* is found, the function overwrites the delimiter in *string* by a null-character and returns a pointer to the token, i.e. the part of the scanned string previous to the member of *delimiters*. After a first call to strtok, the function may be called with NULL as the string parameter, and it will continue from where the last call to strtok found a member of *delimiters*. Delimiters may vary from one call to another.

math.h

- **HUGE_VAL**
  symbolic constant for a positive double expression
- double **sin**(double x)
- double **cos**(double x)
- double **tan**(double x)
- double **asin**(double x)
- double **acos**(double x)
- double **atan**(double x)
  arc tangent of x in the range [-π/2,+π/2] radians

math.h

- double **atan2**(double y, double x)
  arc tangent of y/x in the range [-π,+π] radians
- double **sinh**(double x)
- double **cosh**(double x)
- double **tanh**(double x)
- double **exp**(double x)
- double **log**(double x)
- double **log10**(double x)

math.h

- double **pow**(double x, double y)
- double **sqrt**(double x)
- double **ceil**(double x)
- double **floor**(double x)
- double **fabs**(double x)
- double **ldexp(double x, int n)**
  $x.2^n$

math.h

- double **frexp**(double x, int *exp)
  splits x into a normalized fraction in the interval [1/2, 1), which is returned, and a power of 2, which is stored in *exp. If x is zero, both parts of the result are zero.

- double **modf**(double x, double *iptr)
  splits x into integral and fractional parts, each with the same sign as x. Integral part is stored in *iptr, and the fractional part is returned.

- double **fmod**(double x, double y)
  floating-point remainder of x/y.

math.h

- **Domain Error (EDOM) :** If an argument is outside the domain over which the function is defined

- **Range Error (ERANGE)** : If the result cannot be represented as a double.

  If the result overflows, function returns HUGE_VAL with the right sign, and errno is set to ERANGE

  If the result underflows, function returns zero, whether errno is set to ERANGE is implementation defined

Searching

- Linear Search
  - Simplest but Costly
- Binary Search
  - Efficient but assumes the input to be sorted

Binary Search

```
int BinarySearch(int x, int v[], int n)
{
    int low, high, mid;
    low = 0;
    high = n – 1;
    while ( low    <= high) {
        mid = (low + high) / 2;
        if (x < v[mid])
            high = mid – 1;
        else if (x > v[mid])
            low = mid + 1;
        else
            return mid; /*found match*/
    }
    return –1; /*No match*/
}
```

Binary Search

Available in libc

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base,
    size_t nmemb,size_t size, int (*compar)(const
    void *, const void *));
```

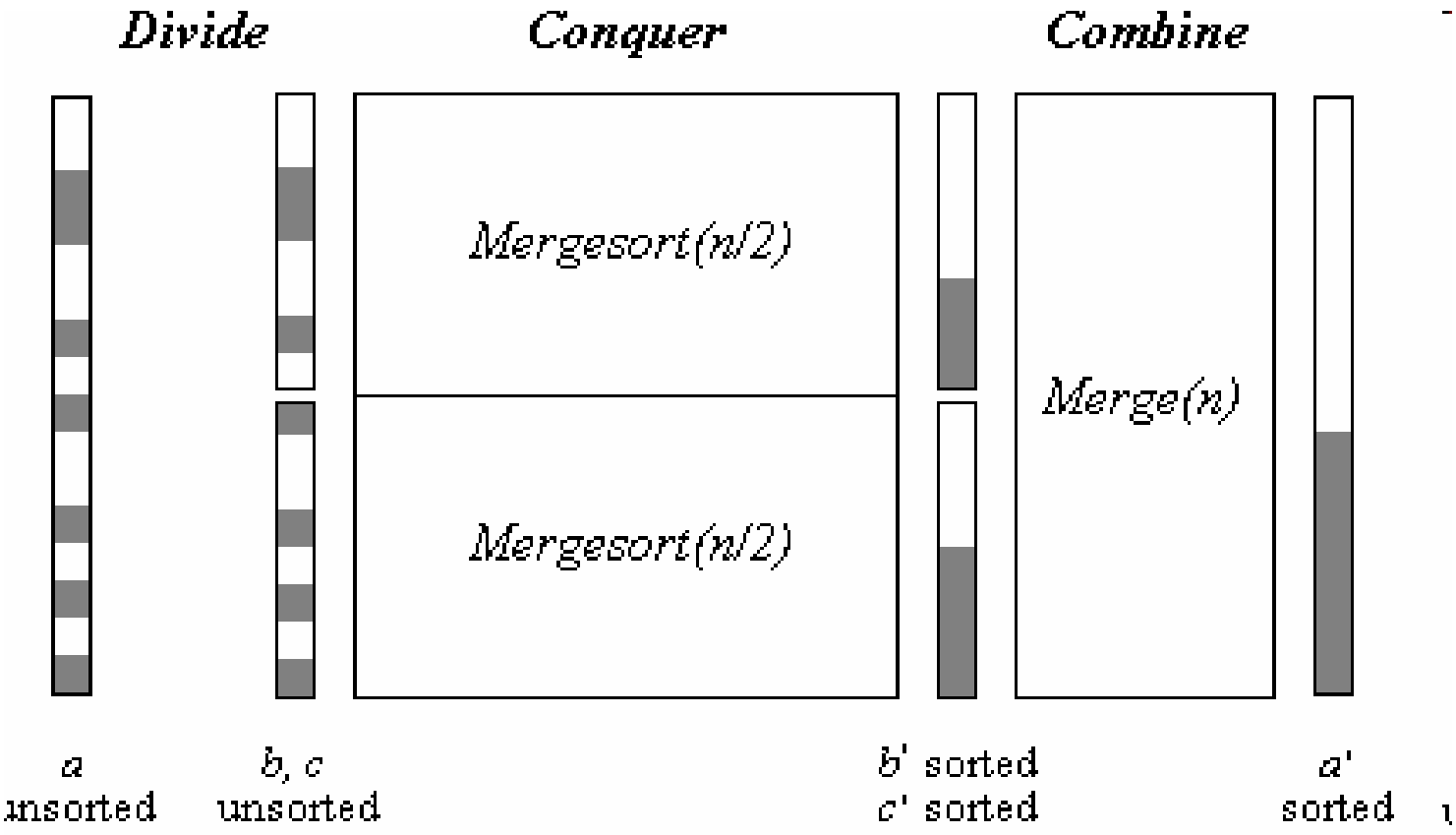
Sorting

- **Insertion Sort**
- **Merge Sort**
- **Quick Sort**

Insertion Sort

```
void insertionSort(int numbers[], int array_size)
{
    int i, j, index;
    for (i=1; i < array_size; i++) {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j-1] > index)) {
            numbers[j] = numbers[j-1];
            j = j - 1;
        }
    numbers[j] = index;
    }
}
```

Merge Sort


Divide
Conquer
Combine
Mergesort(n/2)
Mergesort(n/2)
Merge(n)
a
unsorted
b, c
unsorted
b' sorted
c' sorted
a'
sorted

Merge Sort

```
void mergesort(int lo, int hi)
{
    if (lo<hi) {
        int m=(lo+hi)/2;
        mergesort(lo, m);
        mergesort(m+1, hi);
        merge(lo, m, hi);
    }
 }
```

Quick Sort

Algorithm

- Pick an element, called a *pivot*, from the list.
- Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively sort the list of lesser elements and the list of greater elements in sequence.

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
    int(*compar)(const void *, const void *));
```

Thank You