# CISC 3142
# Programming Paradigms in C++

## Ch43 – The C Standard Library

### The Standard Library

(*Stroustrup – The C++ Programming Language, 4th Ed*)

Brooklyn College, Fall 2020

# Introduction

- The standard library for the C language is incorporated into the C++ standard library with very minor modification
- Therefore, suppose you want to use C's standard I/O system, you can use <stdio.h> or <cstdio> (the latter being the C++'s counterpart of stdio.h, but defined in the std namespace)
- The C standard library is still in wide use – especially for low-level programming (also *the* choice for Linux systems programming)
- For learning C, the "Kernighan and Ritchie" book is universally recognized as an excellent C textbook (also an exemplar of great technical writing)

# Files

- The I/O system with C is via file operations declared in <stdio.h>
- Open files are accessed through a file pointer FILE*
- Three standard I/O streams are available without the need of opening
  - stdin (keyboard), stdout (screen), and stderr (screen)
- File Open and Close

  **f=fopen(s,m)** Open a file stream for a file named **s** with the mode **m, f** is the **FILE**∗ for the opened file if successful or **nullptr** otherwise

  **x=fclose(f)** Close file stream **f**; return **0** if successful
- A *mode* is a C-style string specifying how a file is to be opened/used

| | |
|---|---|
| **"r"** | Reading |
| **"w"** | Writing (discard previous contents) |
| **"a"** | Append (add at end) |
| **"r+"** | Reading and writing |
| **"w+"** | Reading and writing (discard previous contents) |
| **"b"** | Binary; use together with one or more other modes |

# The printf() Family

- The formatted output function, printf(), is widely used and imitated in other languages

  **n=printf(fmt,args)** Print the format string **fmt** to **stdout**, inserting the arguments **args** as appropriate, **n** is # of characters written

  **n=fprintf(f,fmt,args)** Print the format string **fmt** to file **f**, inserting the arguments **args** as appropriate

  **n=sprintf(s,fmt,args)** Print the format string **fmt** to the C-style string **s**, inserting the arguments **args** as appropriate

- Example (use reference for details of the % format specifiers)

  ```
  int x = 5;
  double y = 3.1415926536;
  const char∗ p = "Pedersen";
  printf("x is '%d', y is '%5.2f', and s is '%s'\n", x, y, p); // 5.2f (total width 5 – including the dot, decimal place 2)
  ```

- Output

  ```
  x is '5', y is ' 3.14' and s is 'Pedersen'
  ```

- The input counterpart of printf() is scanf() (description skipped here)
- Additionally the stdio library also has character read/write functions
  - getc(st), getchar() (=getc(stdin)), ungetc(c, st)
  - putc(c, st), putchar(c) (=putc(c, stdout)),

# C-Style Strings

- A C-style string is a zero-terminated array of char, with functions declared in <cstring> (<string.h>), as well as in <cstdlib>, operating on char*

  **x=strlen(s)** Count the characters (excluding the terminating **0**)

  **p=strcpy(s,s2)** Copy **s2** into **s**; [**s**:**s+n**) and [**s2**:**s2+n**) may not overlap; **p=s**;
  the terminating **0** is copied, may overflow

  **p=strcat(s,s2)** Copy **s2** onto the end of **s**; **p=s**; the terminating **0** is copied

  **x=strcmp(s, s2)** Compare lexicographically: if **s<s2**, then **x** is negative;
  if **s==s2**, then **x==0**; if **s>s2**, then **x** is positive

  **p=strncpy(s,s2,n)** **strcpy** of max **n** characters; may fail to copy terminating **0**

  **p=strncat(s,s2,n)** **strcat** of max **n** characters; may fail to copy terminating **0**

  **x=strncmp(s,s2,n)** **strcmp** of max **n** characters

  **p=strchr(s,c)** **p** points to the first **c** in **s**

  **p=strrchr(s,c)** **p** points to the last **c** in **s**

  **p=strstr(s,s2)** **p** points to the first character of **s** that starts a substring equal to **s2**

  **p=strpbrk(s,s2)** **p** points to the first character of **s** also found in **s2** (similar to find_first_of())

  **p=strtok(s, d)** extract tokens from a string - see sidebar for a typical use example

**Usage of strtok()**

Note that original str is modified

```
char str_orig[] =
        "apple banana cherry";
char *str, *token;
for (str = str_orig; ; str = NULL) {
    token = strtok(str, " ");
    if (token == NULL)
        break;
    printf("%s\n", token);
}
```

# C-Style String Numeric Conversions

- These functions are declared in <cstdlib>

  **x=atof(s) x** is a **double** represented by **s**

  **x=atoi(s) x** is an **int** represented by **s**

  **x=atol(s) x** is a **long** represented by **s**

  **x=atoll(s) x** is a **long long** represented by **s**

  **x=strtod(s,p) x** is a **double** represented by **s,** output **p** points to first char of **s** not used for the conversion (same for all following functions)

  **x=strtof(s,p) x** is a **float** represented by **s**

  **x=strtold(s,p) x** is a **long double** represented by **s**;

  **x=strtol(s,p,b) x** is a **long** represented by **s, b** is the base of the input number [2:36]

  **x=strtoll(s,p,b) x** is a **long long** represented by **s**

  **x=strtoul(s,p,b) x** is an **unsigned long** represented by **s**

  **x=strtoull(s,p,b) x** is an **unsigned long long** represented by **s**

# Memory Manipulation

- These functions are either from <cstring> or <cstdlib>, operating on void* (raw memory w/o type)

  q=memcpy(p,p2,n) Copy **n** bytes from **p2** to **p** (like **strcpy**); [**p**:**p+n**) and [**p2**:**p2+n**) may not overlap; **q=p**

  q=memmove(p,p2,n) Copy **n** bytes from **p2** to **p**; **q=p** (source and destination can overlap)

  x=memcmp(p,p2,n) Compare **n** bytes from **p2** to the equivalent **n** bytes from **p**;

  **x<0** means **<**, **x==0** means **==**, **0<x** means **>**

  q=memchr(p,c,n) Find **c** (as an **unsigned char**) in [**p**:**p+n**); **q** points to that element; **q=0** if **c** is not found

  q=memset(p,c,n) Copy **c** (converted to an **unsigned char**) into each of [**p**:**p+n**); **q=p**

  p=calloc(n,s) **p** points to **n∗s** bytes initialized to **0** on free store; **p=nullptr** if the bytes could not be allocated

  p=malloc(n) **p** points to **n** uninitialized bytes on free store; **p=nullptr** if the **s** bytes could not be allocated

  q=realloc(p,n) **q** points to **n** bytes on free store; **p** must be a pointer returned by **malloc()** or **calloc()**, or **nullptr**; if possible, reuse the space pointed to by **p**; if not, copy all bytes in the area pointed to by **p** to a new area; **q=nullptr** if **s** bytes could not be allocated

  free(p) Deallocate the memory pointed to by **p**; **p** must be **nullptr** or a pointer returned by **malloc()**, **calloc()**, or **realloc()**

# Date and Time

- Types and functions declared in <ctime>

  clock_t An arithmetic type for holding short time intervals (maybe just a few minutes)

  time_t An arithmetic type for holding long time intervals (maybe centuries)

  tm A **struct** for holding the time of a date (since year 1900)

- The struct tm is defined as follows

```
struct tm {
        int tm_sec;          // second of minute [0:61]; 60 and 61 represent leap seconds (discrepancy between
                                atomic clocks and observed solar time)
        int tm_min;          // minute of hour [0:59]
        int tm_hour;         // hour of day [0:23]
        int tm_mday;         // day of month [1:31]
        int tm_mon;          // month of year [0:11]; 0 means January (note: not [1:12])
        int tm_year;         // year since 1900; 0 means year 1900, and 115 means 2015
        int tm_wday;         // days since Sunday [0:6]; 0 means Sunday
        int tm_yday;         // days since January 1 [0:365]; 0 means January 1
        int tm_isdst;        // hours of daylight savings time (1: DST, 0: Non-DST)
};
```

# Date and Time (cont’)

- Date and Time Functions

  **t=clock()** **t** is the number of clock ticks (1 tick = 1 micro-sec) since the start of the program; **t** is a **clock_t**

  **t=time(pt)** **t** is the current calendar time (# of seconds since 00:00 hours, 1/1/1970 UTC); **pt** is a **time_t** or **nullptr**; **t** is a **time_t**; if **pt!=nullptr**, ∗**pt=t**;

  **d=difftime(t2,t1)** **d** is a **double** representing **t2−t1** in seconds

  **ptm=localtime(pt)** If **pt==nullptr**, **ptm=nullptr**; otherwise **ptm** points to **tm** for the local time in ∗**pt**

  **ptm=gmtime(pt)** If **pt==nullptr**, **ptm=nullptr**; otherwise **ptm** points to the **tm** for Greenwich Mean Time (GMT) in ∗**pt**

  **t=mktime(ptm)** **time_t** for ∗**ptm**, or **time_t(−1)** if a calendar time can’t be represented

  **p=asctime(ptm)** **p** is a C-style string representation for ∗**ptm**

  **p=ctime(t)** **p=asctime(localtime(t))**

  **n=strftime(p,max,fmt,ptm)** Copy ∗**ptm** into [**p**:**p+n+1**) controlled by the format string **fmt**; characters beyond [**p**:**p+max**) are discarded; **n==0** in case of errors; **p[n]=0**; refer to book for complete **fmt** listing

# Date and Time - Examples

- Using clock() to time a function

```
int main(int argc, char∗ argv[]) {
    int n = atoi(argv[1]); // first cmd-line argument
    clock_t t1 = clock();
    if (t1 == clock_t(−1)) { // clock_t(-1) means "clock() didn’t work"
        cerr << "sorry, no clock\n";
        exit(1);
    }
    for (int i = 0; i<n; i++)
        do_something(); // timing loop
    clock_t t2 = clock();
    if (t2 == clock_t(−1)) {
        cerr << "sorry, clock overflow\n";
        exit(2);
    }
    cout << "do_something() " << n << " times took "
            << double(t2−t1)/CLOCKS_PER_SEC << " seconds"
            << " (measurement granularity: " << CLOCKS_PER_SEC
            << " of a second)\n";
}
```

- Customizing time format with strftime()

```
void almost_C() {
    const int max = 80;
    char str[max];
    time_t t = time(nullptr); // current time since epoch in sec
    tm∗ pt = localtime(&t); // convert it to struct tm
    strftime(str, max,"%D, %H:%M (%I:%M%p)\n", pt);
    printf(str);
}
```

- The output is something like:

```
06/28/12, 15:38 (03:38PM) // %I means hour in 12-hour clock
```

# Etc.

- Etc. functions from <cstdlib>

  abort() Terminate the program “abnormally”

  exit(n) Terminate the program with value **n**; **n==0** means successful termination

  system(s) Execute the string as a command (system-dependent)

  qsort(b,n,s,cmp) Sort the array starting at **b** with **n** elements of size **s** using the comparison function **cmp** as: **int(*cmp)(const void* p, const void* q);**

  bsearch(k,b,n,s,cmp) Search for **k** in the sorted array starting at **b** with **n** elements of size **s** using the comparison function **cmp**

  d=rand() **d** is a pseudo-random number in the range [**0**:**RAND_MAX**]

  srand(d) Start a sequence of pseudo-random numbers using **d** as the seed

# C vs C++

| | C | C++ |
|---|---|---|
| Developed by | Dennis Ritchie (1941-2011) in 1973 | Bjarne Stroustrup (1950- ) in 1979 |
| Programming paradigms | Procedural only | Multiparadigm (OOP, generic, meta-, functional) |
| Compatibility | C is a subset of C++ | C++ is a superset of C |
| Source code file extensions | .c (header files: .h) | .cpp, .cc (header files: .h, .hpp) |
| Standard I/O operations | scanf()/printf() | cin/cout |
| Strings | char* pointing to sequence of characters terminated with 0 (null) | std::string is size aware, and supports much more features |
| Data type other than built-ins | struct only (can't have functions as members) | class (support for full-blown OOP) |
| Memory management | malloc(), calloc(), and free() | new/delete operators |
| Supported features | Has no support for: reference, function/operator overloading, namespace, exception handling, templates, etc. | Has support for all listed on left |