CS240: Programming in C

Lecture 12: Function pointers. Variable arguments. Time.

Example

```
#include <stdio.h>

typedef void (*ptrFun)(int, char*);

void print_error(int n, char* str){
  printf("Error (%d): %s\n", n, str);
}


void print_message(int n, char *str) {
  printf("Message(%d): %s\n", n, str);
}


void PassFunctionPointer(ptrFun g, int
    n, char *s) {
  (*g)(n, s);
}
```

```
int main() {
  ptrFun f = NULL;


  f = &print_error;
  (*f)(10, "Call print_error");

  f = &print_message;
  (*f)(11, "Call print_message");

  PassFunctionPointer(&print_error,
   12, "Passing print_error");
  PassFunctionPointer(&print_message,
   13, "Passing print_message");

  return 0;
}
```

A qsort example

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

- Can you spot the function pointer?
- **`Base`** specifies the array, **`nmemb`** is the number of items, **`size`** is the size of each item and **`compar`** the function that does the comparison.

Void pointers

- Pointers that point to a value that has no type (undetermined dereference properties).
- They cannot be directly dereference-ed they have to be cast-ed to some other pointer type that points to a concrete data type before dereferencing it.
- Using void* in qsort allows it to be used on any type of data.

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

```
#include <stdlib.h>
#include <stdio.h>
int my_sort_int(const void *i1,
                const void *i2){

   int item1 = *(int*)i1;
   int item2 = *(int*)i2;

   if(item1 < item2){
        return -1;
   }
   else if(item1 == item2){
        return 0;
   }
   else{
        return 1;
   }
}
```

```
int main(){
   int a[10];
   int i;

   for(i = 0; i < 10; i++){
        a[i] = 10 - i;
   }

   qsort(a, 10, sizeof(int), my_sort_int);

   for(i = 0; i < 10; i++) {
       printf("%d\n”, a[i]);
   }

   return 0;
}
```

Reminder: command-line parameters

```
int main(int argc, char *argv[])
```

- **`argc`**: the number of the command-line arguments the program was called with.
- **`argv`**: pointer to an array of character strings that contain the arguments. By convention, argv[0] is the name of the program so argc is at least 1.

Example

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  while(--argc > 0) {
    printf("%s%s", *++argv, (argc > 1) ? " ": "");
  }
  printf("\n");

  return 0;
}
```

Varying-length Argument List

```
#include <stdarg.h>
void va_start(va_list ap, arg_name);
```

- initializes processing of a varying-length argument list.
- **`arg_name`**, is the name of the parameter to the calling function after which the varying part of the parameter list begins (the parameter immediately before the ,...). The results of the **`va_start`** macro are unpredictable if the argument values are not appropriate.

```
void va_end(va_list ap);
```

- ends varying-length argument list processing

Varying-length Argument List

```
#include <stdarg.h>
(arg_type) va_arg(va_list ap, arg_type);
```

- returns the value of the next argument in a varying-length argument list.
- **`va_list`** must be initialized by a previous use of the **`va_start`** macro, and a corresponding **`va_end`** should be called after finishing processing the arguments.
- **`arg_type`** is the type of the argument that is expected.
- the returned results are unpredictable if the argument values are not appropriate.
- no way to test whether a particular argument is the last one in the list. Attempting to access arguments after the last one in the list produces unpredictable results.

Example

```
void concat (int count, ...) {
    va_list ap;
    char *target, *source;
    if (count <=1)
            return;

    va_start(ap, count);
    target = va_arg(ap, char *);
    target += strlen(target);
    while (--count > 0) {
            source = va_arg(ap, char*);
            while (*source)
                *target++ = *source++;
    }
    *target = '0';
    va_end(ap);
    return;
}
```

```
#include <stdarg.h>
#include <string.h>
#include <stdio.h>

void concat (int count, ...);

int main() {
    char str[20] = "abcd";
    concat(4, str, "efgh", "ijkl", "mnop");
    printf("The concatenated “
             “string = %s\n",str);
     return 0;
}
```

Date and Time

- The time functions fall into three main categories:
  - Functions for measuring elapsed CPU time.
  - Functions for measuring absolute clock or calendar time.
  - Functions for setting alarms and timers.

CPU Time

- Useful when optimizing a program or measure its efficiency
- CPU time is different from actual wall clock time because it does not include any time spent waiting for I/O or when some other process is running.
- CPU time is represented by the data type clock_t, and is given as a number of clock ticks relative to an arbitrary base time marking the beginning of a single program invocation.

CPU Time – Basic Information

```
#include <time.h>
clock_t clock(void);
```

- Returns the CPU time used so far as a `clock_t` which is in units of clock ticks.
- To get the number of seconds used, divide by `CLOCKS_PER_SEC` (the number of clock ticks per second measured by the clock function.).
- The base time is arbitrary but does not change within a single process. If the processor time is not available or cannot be represented, clock returns the value `(clock_t)(-1)`.

Elapsed Time - Example

```
#include <time.h>

int main() {
  clock_t start, end;
  double elapsed;

  start = clock();

  /* Do the work. */

  end = clock();
  elapsed = ((double)(end-start))/CLOCKS_PER_SEC;
  printf(“elapsed: %f\n”, elapsed);

  return 0;
}
```

CPU Time – Detailed Information

```
#include <sys/times.h>
clock_t times(struct tms *buffer);
struct tms {
  clock_t tms_utime;  /* user time */
  clock_t tms_stime;  /* system time */
  clock_t tms_cutime; /* user time of children */
  clock_t tms_cstime; /* system time of children */
 };
```

- Stores the CPU time information for the calling process in buffer. The returned value is the same as the value of clock. On failure, returns **`(clock_t)(-1)`**.
- All of the times are given in clock ticks. These are absolute values; in a newly created process, they are all zero.

Calendar Time – Basic Information

```
#include <time.h>
time_t time(time_t *result);
```

- **`time_t`**: data type used to represent calendar time. As an absolute time value, it represents the number of seconds elapsed since 00:00:00 on January 1, 1970.
- On different systems **`time_t`** can be long, int or double type.
- The time function returns the current time as a value of type **`time_t`**. If the argument result is not a null pointer, the time value is also stored in *result. If the calendar time is not available, the value **`(time_t) (-1)`** is returned.

Calendar Time – Basic Information

```
#include <time.h>
double difftime(time_t time1, time_t
  time0);
```

- The **`difftime`** function returns the number of seconds elapsed between time **`time1`** and time **`time0`**, as a value of type **`double`**.

High Resolution Calendar Time

```
#include <sys/time.h>

struct timeval {
  long tv_sec;    /* seconds */
  long tv_usec;   /* microseconds */
};

 struct timezone {
  int tz_minuteswest; /* minutes West of GMT */
  int tz_dsttime;     /* type of dst correction
   */
};
```

gettimeofday

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct
   timezone *tz);
```

- Returns the current date and time in the **`struct timeval`** structure indicated by **`tv`**. Information about the time zone is returned in the structure pointed at **`tz`**. If the **`tz`** argument is a null pointer, time zone information is ignored.
- The return value is 0 on success and -1 on failure.

See man/info pages for a detailed description.

settimeofday

```
#include <sys/time.h>
int settimeofday(const struct timeval *tv, const
   struct timezone *tz);
```

- Sets the current date and time according to the arguments. As for **`gettimeofday`**, time zone information is ignored if **`tz`** is a null pointer.
- You must be a privileged user in order to use **`settimeofday`**.
- The return value is 0 on success and -1 on failure.

See man/info pages for a detailed description.

Conversions

- Conversion from string of characters

```
#include <stdlib.h>
int atoi(const char *nptr);
long atol(const char *nptr);
double atof(const char *nptr);
```

- Conversion to string of characters

```
#include <stdio.h>
int sprintf(char *str, const char
  *format, ...);
```

atoi

```
#include <stdlib.h>
int atoi(const char *nptr);
```

- Converts the initial portion of the string pointed to by nptr to int.
- Returns the converted value.
- Example:

```
char buf[10]=“5”;
int v = atoi(buf);
```

sprintf

```
#include <stdio.h>
int sprintf(char *str, const char
  *format, ...);
```

- Returns the number of characters printed (not including the trailing `\0' used to end output to strings).
- Examples:

```
char buf2[100], buf2[100];
int v = 10;
double d = 4.56;

sprintf(buf1, “d”, v);
sprintf(buf2, “%4.2f”, d);
```

Readings and exercises for this lecture

Read the recommended man pages.

Code all the examples in the lecture.

Write a my_sort_string function using function pointers and qsort.