简介

- C语言常用函数

mlockall

- 简介：
  - mlockall() 是一个函数，它用于将进程的内存锁定在物理内存中，防止被换出到磁盘交换空间
- 原型：

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

int mlockall(int flags);
```

- 参数：
  - flags：指定锁定内存的选项。可以是以下值的按位或
    - MCL_CURRENT：锁定当前进程的地址空间中的所有内存。
    - MCL_FUTURE：锁定当前进程在将来动态分配的内存
- 返回值：
  - 成功时，返回0；
  - 失败时，返回-1，并设置 errno 表示错误原因
- 详解：
  - mlockall() 函数用于在内存敏感的应用程序中确保关键数据或代码保持在物理内存中，以提高访问速度和可靠性
  - 调用 mlockall() 后，指定选项的内存将被锁定，不会被交换到磁盘上的交换空间
  - 锁定内存可能需要特权权限，通常只有具有足够权限的进程才能成功调用 mlockall() 函数
  - 当不再需要锁定内存时，可以使用 munlockall() 函数来解锁内存
- 注：
  - 需要注意的是，使用 mlockall() 函数需要谨慎，因为过度锁定内存可能导致系统资源不足，并影响整体性能。仅在确实需要将关键数据保留在物理内存中并且有足够的系统资源时才应使用该函数

---

popen

- 简介：
  - popen()是一个C标准库函数，用于在一个新进程中执行shell命令，并创建一个管道来与该进程通信
- 原型：

```
FILE* popen(const char* command, const char* mode);
```

- 参数：
  - command：要执行的shell命令。
  - mode：操作模式，可以是"r"（读取模式）或"w"（写入模式）
- 返回值：
  - 如果成功，返回一个指向FILE结构的指针，用于后续与新进程之间的读写操作
  - 如果失败，返回NULL
- 详解：
  - 函数功能：
    - popen()函数通过创建一个新的进程来执行指定的shell命令，并返回一个指向FILE结构的指针。该FILE结构可以用于后续的读取或写入操作。新进程的标准输入或标准输出将通过管道与调用进程进行通信
  - 操作模式
    - "r"：读取模式，用于从新进程的标准输出读取数据。
    - "w"：写入模式，用于向新进程的标准输入写入数据
  - 注意事项
    - 使用popen()函数执行shell命令时，请确保命令参数是可信的，以防止命令注入等安全问题。
    - 在使用完popen()函数后，应该使用pclose()函数关闭由popen()创建的管道
- 示例：

```
/*下面是一个简单的示例，演示如何使用popen()函数执行shell命令并读取输出：*/
#include <iostream>
#include <cstdio>

int main() {
    const char* command = "ls -l";
    const char* mode = "r";

    FILE* pipe = popen(command, mode);
    if (pipe) {
        char buffer[128];
        while (!feof(pipe) && fgets(buffer, sizeof(buffer), pipe) !=
nullptr) {
            std::cout << buffer;
        }
        pclose(pipe);
    } else {
        std::cout << "Command execution failed." << std::endl;
    }
```

```
    return 0;
}
```

- 注：
  - 上述示例执行ls -l命令，并将命令的输出打印到控制台

__sync_fetch_and_add

- 简介：
  - __sync_fetch_and_add() 是一个GCC内置函数，用于实现原子的自增操作。它可以用于多线程环境下对共享变量进行原子操作，确保操作的原子性和可见性
- 原型：

```
T __sync_fetch_and_add(T* ptr, T value);
```

- 参数：
  - ptr：指向共享变量的指针。
  - value：要添加到共享变量的值
- 返回值：
  - 返回执行自增操作前的共享变量的值
- 详解：
  - 函数功能：
    - __sync_fetch_and_add() 函数用于对给定的共享变量进行原子的自增操作。它将指定的值（value）添加到共享变量（通过指针 ptr 引用）中，并返回执行自增操作前的共享变量的值。
  - 原子操作的特性确保了多个线程对同一共享变量进行自增操作时的正确性。该操作在执行期间不会被中断，以确保其他线程不会在此期间修改共享变量的值
- 示例：

```
/*下面是一个简单的示例，演示了如何使用 __sync_fetch_and_add() 函数对共享变量进行原子的自增操作*/
#include <iostream>

int main() {
    int sharedVariable = 0;

    for (int i = 0; i < 10; i++) {
        int previousValue = __sync_fetch_and_add(&sharedVariable, 1);
        std::cout << "Previous value: " << previousValue << ", New value: " << sharedVariable << std::endl;
    }
```

```
    return 0;
}
```

- 注：
  - 上述示例使用 __sync_fetch_and_add() 函数对共享变量 sharedVariable 进行了10次原子的自增操作。每次自增操作都会打印出执行自增前的值和执行自增后的值

__sync_fetch_and_sub

- 简介：
  - __sync_fetch_and_sub() 是一个GCC内置函数，用于实现原子的自减操作。它可以在多线程环境下对共享变量进行原子操作，确保操作的原子性和可见性
- 原型：

```
T __sync_fetch_and_sub(T* ptr, T value);
```

- 参数：
  - ptr：指向共享变量的指针。
  - value：要从共享变量减去的值
- 返回值：
  - 返回执行自减操作前的共享变量的值
- 详解：
  - 函数功能：
    - __sync_fetch_and_sub() 函数用于对给定的共享变量进行原子的自减操作。它从共享变量（通过指针 ptr 引用）中减去指定的值（value），并返回执行自减操作前的共享变量的值
  - 原子操作的特性确保了多个线程对同一共享变量进行自减操作时的正确性。该操作在执行期间不会被中断，以确保其他线程不会在此期间修改共享变量的值
- 示例：

```
/*下面是一个简单的示例，演示了如何使用 __sync_fetch_and_sub() 函数对共享变量进行原子的自减操作*/
#include <iostream>

int main() {
    int sharedVariable = 10;

    for (int i = 0; i < 10; i++) {
        int previousValue = __sync_fetch_and_sub(&sharedVariable, 1);
        std::cout << "Previous value: " << previousValue << ", New value: " << sharedVariable << std::endl;
```

```
    }

    return 0;
}
```

- 注：
  - 上述示例使用 __sync_fetch_and_sub() 函数对共享变量 sharedVariable 进行了10次原子的自减操作。每次自减操作都会打印出执行自减前的值和执行自减后的值

__sync_fetch_and_or

- 简介：
  - __sync_fetch_and_or() 是一个GCC内置函数，用于实现原子的按位或操作。它可以在多线程环境下对共享变量进行原子操作，确保操作的原子性和可见性
- 原型：

```
T __sync_fetch_and_or(T* ptr, T value);
```

- 参数：
  - ptr：指向共享变量的指针。
  - value：要进行按位或操作的值
- 返回值：
  - 返回执行按位或操作前的共享变量的值
- 详解：
  - 函数功能：
    - __sync_fetch_and_or() 函数用于对给定的共享变量进行原子的按位或操作。它将指定的值（value）与共享变量（通过指针 ptr 引用）进行按位或操作，并返回执行按位或操作前的共享变量的值
  - 原子操作的特性确保了多个线程对同一共享变量进行按位或操作时的正确性。该操作在执行期间不会被中断，以确保其他线程不会在此期间修改共享变量的值
- 示例：

```
/* 下面是一个简单的示例，演示了如何使用 __sync_fetch_and_or() 函数对共享变量进行原子的按位或操作*/
#include <iostream>

int main() {
    int sharedVariable = 0b1100;  // 初始值为 12，二进制为 1100

    int newValue = 0b0011;  // 要进行按位或操作的值，二进制为 0011
```

```
    int previousValue = __sync_fetch_and_or(&sharedVariable, newValue);
    std::cout << "Previous value: " << previousValue << ", New value: " <<
sharedVariable << std::endl;

    return 0;
}
```

- 注：
  - 上述示例使用 __sync_fetch_and_or() 函数对共享变量 sharedVariable 进行了原子的按位或操作。在示例中，共享变量初始值为 12，二进制表示为 1100，要进行按位或操作的值为 3，二进制表示为 0011。执行按位或操作后，共享变量的值变为 15，二进制表示为 1111。示例中打印出了执行按位或操作前的值和执行按位或操作后的值

__sync_and_and_fetch

- 简介：
  - __sync_and_and_fetch() 是一个GCC内置函数，用于实现原子的按位与操作。它可以在多线程环境下对共享变量进行原子操作，确保操作的原子性和可见性
- 原型：

```
T __sync_and_and_fetch(T* ptr, T value);
```

- 参数：
  - ptr：指向共享变量的指针。
  - value：要进行按位与操作的值
- 返回值：
  - 返回执行按位与操作后的共享变量的值
- 详解：
  - 函数功能：
    - __sync_and_and_fetch() 函数用于对给定的共享变量进行原子的按位与操作。它将指定的值（value）与共享变量（通过指针 ptr 引用）进行按位与操作，并返回执行按位与操作后的共享变量的值
  - 原子操作的特性确保了多个线程对同一共享变量进行按位与操作时的正确性。该操作在执行期间不会被中断，以确保其他线程不会在此期间修改共享变量的值
- 示例：

```
/*下面是一个简单的示例，演示了如何使用 __sync_and_and_fetch() 函数对共享变量进行原子的按位与操作*/
#include <iostream>
```

```
int main() {
    int sharedVariable = 0b1111;  // 初始值为 15，二进制为 1111

    int value = 0b1100;  // 要进行按位与操作的值，二进制为 1100

    int result = __sync_and_and_fetch(&sharedVariable, value);
    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

- 注：
  - 上述示例使用 __sync_and_and_fetch() 函数对共享变量 sharedVariable 进行了原子的按位与操作。在示例中，共享变量初始值为 15，二进制表示为 1111，要进行按位与操作的值为 12，二进制表示为 1100。执行按位与操作后，共享变量的值变为 12，二进制表示为 1100。示例中打印出了执行按位与操作后的值

__sync_bool_compare_and_swap

- 简介：
  - __sync_bool_compare_and_swap() 是一个GCC内置函数，用于实现原子的比较并交换操作。它可以在多线程环境下对共享变量进行原子操作，确保操作的原子性和可见性
- 原型：

```
bool __sync_bool_compare_and_swap(T* ptr, T oldval, T newval);
```

- 参数：
  - ptr：指向共享变量的指针。
  - oldval：期望的旧值。
  - newval：要替换的新值
- 返回值：
  - 如果共享变量的当前值与旧值相等，则将新值赋给共享变量，并返回 true。
  - 如果共享变量的当前值与旧值不相等，则不进行替换，并返回 false
- 详解：
  - 函数功能：
    - __sync_bool_compare_and_swap() 函数用于比较共享变量的当前值与旧值是否相等，如果相等则替换为新值。该操作是原子的，确保在执行期间不会被中断，以确保其他线程不会在此期间修改共享变量的值
  - 函数返回 true 表示替换操作成功，共享变量的值已被替换为新值。返回 false 表示替换操作失败，共享变量的值与旧值不相等，替换未执行
- 示例：

```
/*下面是一个简单的示例，演示了如何使用 __sync_bool_compare_and_swap() 函数进行原子的比较并交换操作*/
#include <iostream>

int main() {
    int sharedVariable = 10;  // 共享变量的初始值为 10

    int oldValue = 10;  // 期望的旧值
    int newValue = 20;  // 要替换的新值

    bool result = __sync_bool_compare_and_swap(&sharedVariable, oldValue, newValue);
    std::cout << "Result: " << result << ", Shared variable: " << sharedVariable << std::endl;

    return 0;
}
```

- 注：
  - 上述示例使用 __sync_bool_compare_and_swap() 函数对共享变量 sharedVariable 进行了比较并交换操作。在示例中，共享变量初始值为 10，我们期望将其替换为新值 20。通过 __sync_bool_compare_and_swap() 函数，我们将共享变量的当前值与旧值 10 进行比较，如果相等则替换为新值 20。示例中打印出了操作的结果和最终的共享变量值

__sync_val_compare_and_swap

- 简介：
  - __sync_val_compare_and_swap() 是一个GCC内置函数，用于实现原子的比较并交换操作。它可以在多线程环境下对共享变量进行原子操作，确保操作的原子性和可见性
- 原型：

```
T __sync_val_compare_and_swap(T* ptr, T oldval, T newval);
```

- 参数：
  - ptr：指向共享变量的指针。
  - oldval：期望的旧值。
  - newval：要替换的新值
- 返回值：
  - 返回共享变量在比较并交换操作之前的值
- 详解：
  - 函数功能：

        - __sync_val_compare_and_swap() 函数用于比较共享变量的当前值与旧值是否相等，如果相等则替换为新值，并返回比较交换之前的共享变量的值。该操作是原子的，确保在执行期间不会被中断，以确保其他线程不会在此期间修改共享变量的值
    - 函数返回共享变量在比较并交换操作之前的值，可以用于判断操作是否成功执行

- 示例：

```
/*下面是一个简单的示例，演示了如何使用 __sync_val_compare_and_swap() 函数进行原子的比较并交换操作*/
#include <iostream>

int main() {
    int sharedVariable = 10;  // 共享变量的初始值为 10

    int oldValue = 10;  // 期望的旧值
    int newValue = 20;  // 要替换的新值

    int result = __sync_val_compare_and_swap(&sharedVariable, oldValue, newValue);
    std::cout << "Result: " << result << ", Shared variable: " << sharedVariable << std::endl;

    return 0;
}
```

- 注：
    - 上述示例使用 __sync_val_compare_and_swap() 函数对共享变量 sharedVariable 进行了比较并交换操作。在示例中，共享变量初始值为 10，我们期望将其替换为新值 20。通过 __sync_val_compare_and_swap() 函数，我们将共享变量的当前值与旧值 10 进行比较，如果相等则替换为新值 20，并返回比较交换之前的共享变量的值。示例中打印出了操作的结果和最终的共享变量值

sleep()

- 简介：
    - sleep() 函数是用于在指定的时间内使程序暂停执行的函数。它可以用于创建一定的延迟或控制程序的执行速度

- 原型：

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

- 参数：
    - seconds : 表示要暂停的秒数

- 返回值：
    - 如果函数成功暂停了指定的秒数，返回为 0。
    - 如果函数由于被信号中断而返回，则返回剩余的秒数
- 示例：

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Start\n");

    // 暂停 2 秒
    sleep(2);

    printf("End\n");

    return 0;
}
```

- 注：
    - 在上面的示例中，使用 sleep(2) 函数使程序暂停 2 秒。在输出 "Start" 后，程序会暂停 2 秒钟，然后输出 "End"
    - sleep() 函数在头文件 unistd.h 中声明，并且是一个标准库函数。它会导致当前进程阻塞，即程序会进入睡眠状态，不会执行其他操作，直到指定的时间过去
    - 需要注意的是，sleep() 函数的时间精度是秒级别的，并且在暂停期间，程序会处于完全休眠状态，不会对其他任务进行处理。如果需要更精确的时间控制或需要同时处理其他任务，可以考虑使用定时器或多线程等机制来实现
    - 另外，sleep() 函数在 POSIX 标准中定义，因此在不同的操作系统或编译器中可能会有一些差异。在特定平台或环境下使用 sleep() 函数时，建议查看相应的文档和标准以获取更详细的信息

---

calloc()

- 简介：
    - calloc() 函数用于动态分配内存，并将分配的内存块初始化为零。它是 malloc() 函数的一种变体，用于分配连续的内存空间
- 原型：

```c
#include <stdlib.h>
void* calloc(size_t num, size_t size);
```

- 参数：
    - num: 要分配的元素数量。

    - size: 每个元素的大小（以字节为单位）

- 返回值：

    - 如果分配成功，返回指向分配内存块的指针（指向首字节的指针）。
    - 如果分配失败，返回 NULL

- 示例：

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* arr;

    // 分配包含 5 个整数的内存块，并初始化为零
    arr = (int*)calloc(5, sizeof(int));
    if (arr == NULL) {
        printf("内存分配失败\n");
        return 1;
    }

    // 输出分配的内存块中的值
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // 释放内存块
    free(arr);

    return 0;
}
```

- 注：
    - 在上面的示例中，使用 calloc(5, sizeof(int)) 函数分配了包含 5 个整数的内存块，并将其初始化为零。返回的指针 arr 指向分配内存块的首字节
    - calloc() 函数在头文件 stdlib.h 中声明，并且是一个标准库函数。它在内存分配时会自动将分配的内存块中的所有字节初始化为零值。这与使用 malloc() 函数后需要手动进行内存清零的过程相比，提供了方便和效率
    - 需要注意的是，calloc() 函数返回的指针必须通过 free() 函数进行释放，以避免内存泄漏
    - 另外，如果分配的内存块过大或者系统内存不足，calloc() 函数可能会分配失败，此时返回 NULL。在使用 calloc() 函数分配内存时，建议检查返回的指针是否为 NULL，以确保分配成功
    - 总结而言，calloc() 函数是一种方便的内存分配函数，可以分配连续的内存块并初始化为零。在需要分配内存并初始化为零的情况下，可以考虑使用 calloc() 函数

---

strlen()

- 简介：

    - strlen() 函数用于计算字符串的长度，即字符串中字符的个数（不包括终止符 \0）

- 原型：

```
#include <string.h>
size_t strlen(const char* str);
```

- 参数：
    - str : 要计算长度的字符串，必须是以 \0 结尾的字符数组或字符指针
- 返回值：
    - 返回类型为 size_t（无符号整数），表示字符串的长度
- 示例：

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

int main() {
    char str[] = "Hello, world!";
    size_t length = strlen(str);

    printf("字符串的长度是：%zu\n", length);

    return 0;
}
```

- 注：
    - 在上面的示例中，使用 strlen() 函数计算了字符串 str 的长度，并将结果存储在变量 length 中。然后，使用 printf() 函数打印字符串的长度
    - strlen() 函数在头文件 string.h 中声明，并且是一个标准库函数。它通过逐个扫描字符，直到遇到字符串终止符 \0 来计算字符串的长度
    - 需要注意的是，strlen() 函数只能用于以 \0 结尾的字符串。如果传递的字符串不是以 \0 结尾或者是一个空指针（NULL），则 strlen() 函数的行为是未定义的。因此，在使用 strlen() 函数时，必须确保字符串是以 \0 结尾的有效字符串
    - 另外，由于 strlen() 函数返回的是 size_t 类型，因此在使用 %zu 格式符打印字符串的长度时，需要使用 size_t 类型的变量作为参数
    - 总结而言，strlen() 函数是一种方便的计算字符串长度的函数，可以用于获取字符串中字符的个数（不包括终止符 \0）。在需要获取字符串长度的情况下，可以使用 strlen() 函数。

---

dup2()

- 简介：

    - dup2() 函数用于复制文件描述符，并将其复制到指定的目标文件描述符。通过 dup2() 函数，可以实现重定向文件描述符的功能

- 原型：

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- 参数：
    - oldfd：要复制的旧文件描述符。
    - newfd：要复制到的新文件描述符。
- 返回值：
    - 如果复制成功，返回新的文件描述符。
    - 如果复制失败，返回 -1，并设置 errno 错误码
- 示例：

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int oldfd = open("input.txt", O_RDONLY); // 打开旧文件描述符
    int newfd = 3; // 新文件描述符

    int result = dup2(oldfd, newfd); // 复制文件描述符
    if (result == -1) {
        perror("dup2");
        return 1;
    }

    // 使用新的文件描述符读取数据
    char buffer[100];
    ssize_t bytesRead = read(newfd, buffer, sizeof(buffer) - 1);
    if (bytesRead == -1) {
        perror("read");
        return 1;
    }
    buffer[bytesRead] = '\0';

    printf("读取的数据：%s\n", buffer);

    close(oldfd); // 关闭旧文件描述符

    return 0;
}
```

- 注：
  - 在上面的示例中，首先通过 open() 函数打开一个文件，并获得旧的文件描述符 oldfd。然后，定义一个新的文件描述符 newfd，将其设置为 3
  - 接下来，使用 dup2(oldfd, newfd) 函数将旧文件描述符复制到新的文件描述符。如果复制成功，dup2() 函数将返回新的文件描述符
  - 在示例中，我们使用新的文件描述符 newfd 执行 read() 函数，读取数据到缓冲区 buffer 中，并在末尾添加字符串终止符。最后，我们打印读取的数据。
  - 需要注意的是，dup2() 函数会关闭新的文件描述符（如果它已经打开）。因此，在使用 dup2() 函数之前，应确保新的文件描述符没有被使用或需要关闭
  - 另外，通过使用 dup2() 函数，还可以实现重定向标准输入、输出和错误的功能。例如，将标准输出重定向到文件中可以使用 dup2(fileDescriptor, STDOUT_FILENO)
  - 总结而言，dup2() 函数是一个用于复制文件描述符的函数，可以将一个文件描述符复制到另一个指定的文件描述符。通过 dup2() 函数，可以实现文件描述符的重定向和复制功能

---

write()

- 简介：
  - write()函数用于将数据从文件描述符（通常是文件、管道、套接字等）写入到指定的缓冲区
- 原型：

```
ssize_t write(int fd, const void *buf, size_t count);
```

- 参数：
  - fd：文件描述符，表示要写入的文件或设备
  - buf：指向要写入的数据的缓冲区的指针
  - count：要写入的字节数
- 返回值：
  - 如果写入成功，返回值为实际写入的字节数
  - 如果发生错误，返回值为-1，并设置errno变量以指示错误类型
- 详解：
  - write()函数会尽可能地写入指定数量的字节到文件描述符中。它不保证一次性写入全部数据，可能会根据底层实现的不同进行分多次写入。因此，在使用write()函数时，需要检查返回值以确保写入了所需的字节数
- 示例：

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
```

```c
int main() {
    int fd; // 文件描述符
    const char *data = "Hello, World!"; // 要写入的数据
    size_t len = strlen(data); // 数据长度

    // 打开文件（如果不存在则创建）
    fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("Failed to open file");
        return 1;
    }

    // 写入数据到文件
    ssize_t bytes_written = write(fd, data, len);
    if (bytes_written == -1) {
        perror("Failed to write data to file");
        return 1;
    }

    printf("Data written to file: %ld bytes\n", bytes_written);

    // 关闭文件
    close(fd);

    return 0;
}
```

- 注：
  - 在上述示例中，我们首先使用open()函数打开一个文件，如果文件不存在则创建它。然后，我们使用write()函数将数据写入文件。最后，我们使用close()函数关闭文件
  - 请注意，上述示例仅演示了将数据写入文件的情况。在实际应用中，你可能需要根据需要调整文件打开模式和权限，并处理错误情况

read()

- 简介：
  - read() 函数用于从文件描述符中读取数据
- 原型

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- 参数
  - fd：文件描述符，指定要读取的文件或其他数据源
  - buf：用于存储读取数据的缓冲区的指针

    - count：要读取的最大字节数

- 返回值：

    - 如果成功读取数据，返回实际读取的字节数
    - 如果到达文件末尾，返回0
    - 如果发生错误，返回-1，并设置 errno 变量来指示具体的错误原因

- 详解：

    - read() 函数将尽可能读取指定数量的字节，并将其存储在提供的缓冲区中。它会阻塞，直到读取到足够的字节或者遇到文件末尾。如果文件中没有足够的数据可供读取，那么 read() 函数将一直等待，直到有足够的数据可供读取或者遇到其他读取条件（如文件被关闭、读取超时等）

- 示例

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 1024

int main() {
    int fd = open("file.txt", O_RDONLY);
    if (fd == -1) {
        perror("Failed to open file");
        return 1;
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytesRead;

    while ((bytesRead = read(fd, buffer, BUFFER_SIZE)) > 0) {
        write(STDOUT_FILENO, buffer, bytesRead);
    }

    if (bytesRead == -1) {
        perror("Failed to read file");
        close(fd);
        return 1;
    }

    close(fd);
    return 0;
}
```

- 注：
    - 上述示例代码使用 open() 函数打开文件，并使用 read() 函数从文件中读取数据，然后使用 write() 函数将数据写入到标准输出。循环读取直到 read() 函数返回值为0，表示已读取完所有数据

    - 注意：在实际使用 read() 函数时，需要注意处理返回值、错误检查和错误处理，以确保数据的正确读取和错误的及时处理

mmap()

- 简述：
    - mmap() 函数是 C/C++ 中的一个系统调用，用于将文件映射到进程的虚拟内存空间。
- 原型：

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
```

- 参数：
    - addr：映射的起始地址。可以传入 NULL，表示由系统自动选择一个合适的地址。
    - length：映射的长度，以字节为单位。
    - prot：映射的保护模式，指定对内存区域的访问权限。常用的取值包括 PROT_READ（可读）、PROT_WRITE（可写）和 PROT_EXEC（可执行）
    - flags：映射的标志位，用于指定映射的属性。常用的取值包括 MAP_SHARED（与其他进程共享）和 MAP_PRIVATE（私有映射）
    - fd：文件描述符，指定要映射的文件。如果不需要映射文件，可以传入 -1。
    - offset：文件偏移量，指定从文件的哪个位置开始映射
- 返回值：
    - 如果映射成功，返回映射区域的起始地址。
    - 如果映射失败，返回 MAP_FAILED，即 (void*)-1。
- 详解：
    - 使用 mmap() 函数可以将文件映射到进程的内存空间，从而可以直接通过内存访问文件的内容，而无需进行频繁的文件 I/O 操作。映射的文件内容可以像访问内存一样进行读写操作，对映射区域的修改也会反映到原始文件上
- 示例：

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>

int main() {
    const char* filename = "example.txt";
    int fd = open(filename, O_RDONLY);
```

```
    if (fd == -1) {
        std::cerr << "Failed to open file: " << filename << std::endl;
        return 1;
    }

    struct stat file_stat;
    if (fstat(fd, &file_stat) == -1) {
        std::cerr << "Failed to get file size" << std::endl;
        close(fd);
        return 1;
    }

    off_t file_size = file_stat.st_size;

    void* addr = mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED) {
        std::cerr << "Failed to map file to memory" << std::endl;
        close(fd);
        return 1;
    }

    // 在 addr 指向的内存区域中可以直接访问文件的内容

    munmap(addr, file_size);
    close(fd);

    std::cout << "File mapped to memory successfully" << std::endl;

    return 0;
}
```

- 注：
  - 上述代码打开名为 "example.txt" 的文件，获取其大小后，使用 mmap() 函数将文件内容映射到内存中。如果映射成功，将输出 "File mapped to memory successfully"。最后，使用 munmap() 函数解除映射，关闭文件
  - 请注意，使用 mmap() 函数时应格外谨慎，确保正确地处理内存映射区域，避免出现悬挂指针和访问越界等问题。同时，对映射区域的读写操作可能会影响到原始文件，需要注意数据的同步与持久化

dirname()

- 简介：
  - dirname() 函数用于提取路径字符串中的目录部分。它返回一个新的字符串，其中包含给定路径的目录部分
- 原型：

```
#include <libgen.h>
char* dirname(char* path);
```

- 参数：
    - path : 要提取目录的路径字符串
- 返回值：
    - 返回一个指向目录部分的字符串的指针。
    - 如果路径不包含目录部分（如根目录 / 或空路径），则返回一个句点 .
- 示例：

```c
#include <stdio.h>
#include <libgen.h>

int main() {
    char path[] = "/usr/local/bin/executable";
    char* dir = dirname(path);

    printf("目录部分：%s\n", dir);

    return 0;
}
```

- 注：
    - 在上面的示例中，使用 dirname() 函数从路径字符串 /usr/local/bin/executable 中提取目录部分。然后，将返回的目录字符串 dir 打印出来
    - dirname() 函数在头文件 libgen.h 中声明，并且是一个标准库函数。它会修改传入的路径字符串，并返回指向目录部分的指针。注意，传入的路径字符串应为可修改的字符数组（C 字符串），而不是字符串常量
    - 需要注意的是，dirname() 函数的操作是基于路径字符串的规则来进行的，它不会检查目录是否存在或路径是否有效。它只是从给定的路径字符串中提取目录部分，并返回一个新的字符串
    - 另外，由于 dirname() 函数会修改传入的路径字符串，所以如果需要保留原始路径字符串，应在调用 dirname() 函数之前创建一个副本
    - 总结而言，dirname() 函数是一个用于提取路径字符串中目录部分的函数。它可以帮助您从路径字符串中获取目录信息，以便进行进一步的处理

basename()

- 简介：
    - basename() 函数用于提取路径字符串中的文件名部分。它返回一个新的字符串，其中包含给定路径的文件名部分
- 原型：

```c
#include <libgen.h>
char* basename(char* path);
```

- 参数：
    - path : 要提取文件名的路径字符串。
- 返回值：
    - 返回一个指向文件名部分的字符串的指针
- 示例：

```c
#include <stdio.h>
#include <libgen.h>

int main() {
    char path[] = "/usr/local/bin/executable";
    char* file = basename(path);

    printf("文件名部分：%s\n", file);

    return 0;
}
```

- 注：
    - 在上面的示例中，使用 basename() 函数从路径字符串 /usr/local/bin/executable 中提取文件名部分。然后，将返回的文件名字符串 file 打印出来。
    - basename() 函数在头文件 libgen.h 中声明，并且是一个标准库函数。它会修改传入的路径字符串，并返回指向文件名部分的指针。注意，传入的路径字符串应为可修改的字符数组（C 字符串），而不是字符串常量
    - 需要注意的是，basename() 函数的操作是基于路径字符串的规则来进行的，它不会检查文件是否存在或路径是否有效。它只是从给定的路径字符串中提取文件名部分，并返回一个新的字符串。
    - 另外，由于 basename() 函数会修改传入的路径字符串，所以如果需要保留原始路径字符串，应在调用 basename() 函数之前创建一个副本。
    - 总结而言，basename() 函数是一个用于提取路径字符串中文件名部分的函数。它可以帮助您从路径字符串中获取文件名信息，以便进行进一步的处理。

---

readlink()

- 简介：
    - readlink() 函数用于读取符号链接文件的目标路径。它可以帮助我们获取符号链接文件所指向的实际文件或目录
- 原型：

```c
#include <unistd.h>
ssize_t readlink(const char* path, char* buf, size_t bufsiz);
```

- 参数：
    - path：符号链接文件的路径。
    - buf：用于存储目标路径的缓冲区。
    - bufsiz：缓冲区的大小
- 返回值：
    - 如果读取成功，返回实际读取的字节数。
    - 如果读取失败，返回 -1，并设置 errno 错误码
- 示例：

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    char path[] = "symlink.txt";
    char target[100];

    ssize_t len = readlink(path, target, sizeof(target) - 1);
    if (len == -1) {
        perror("readlink");
        return 1;
    }

    target[len] = '\0';

    printf("符号链接 %s 的目标路径是：%s\n", path, target);

    return 0;
}
```

- 注：
    - 在上面的示例中，我们使用 readlink() 函数读取名为 "symlink.txt" 的符号链接文件的目标路径。我们提供一个缓冲区 target 来存储目标路径，并指定缓冲区的大小
    - 如果读取成功，readlink() 函数返回实际读取的字节数，并将目标路径存储在缓冲区 target 中。我们在目标路径末尾添加了字符串终止符，以确保字符串的正确性。
    - 需要注意的是，readlink() 函数要求传入的路径是符号链接文件的路径，而不是目标文件的路径。如果路径不是符号链接文件，或者符号链接文件不存在，readlink() 函数将返回 -1，并设置适当的错误码
    - 此外，为了避免缓冲区溢出，我们应该根据实际情况来确定缓冲区的大小，并在调用 readlink() 函数之前确保缓冲区足够大
    - 总结而言，readlink() 函数是一个用于读取符号链接文件目标路径的函数。它可以帮助我们获取符号链接文件所指向的实际文件或目录

---

strdup()

- 简述：转储字符串，返回一个字符指针，其存储的内容和参数`s`相同，获得的内存是使用`malloc()`完成的，因此可以使用`free()`完成
- 声明：`char* strdup(const char *s);`
- 参数：
  - `s` -- 需要转储的字符串
- 返回值：
  - 返回新存储的字符指针

mkdir()

- 简述：创建一个目录`path`，并将文件权限设置为`mode`
- 声明：`int mkdir(const char *path, __model_t mode);`
- 参数：
  - `path` -- 目录文件路径
  - `model` -- 创建的目录文件的权限
- 返回值：
  - 成功 --
  - 失败 --

chmod()

- 简述：将文件`file`的权限设置为`mode`
- 声明：`int chmod(const char* file, mode_t mode);`
- 参数：
  - `file` -- 需要修改的文件
  - `mode` -- 权限
- 返回值：
  - 成功 --
  - 失败 --

open()

- 简述：以指定模式打开指定文件
- 声明：`int open(const char *path, int __oflag, ...);`
- 参数：
  - `path` -- 要操作的文件，绝对路径
  - `oflag` -- 通常由一下常量通过或运算`I`组成
    - `O_RDONlY` 只读打开
    - `O_WRONlY` 只写打开
    - `O_RDWR` 读写打开
    - `O_EXEC` 只执行打开
    - `O_SEARCH` 只搜索（对于目录有此选项）
    - 在以上五个常量中必须指定且只能指定一个，而以下常量为可选的
      - `O_APPEND` 每次写入追加到文件末尾
      - `O_CLOEXEC` 把FD_CLOEXEC设定为文件描述符
      - `O_CREATE` 若文件不存在则创建， 需要指定文件权限位 ， 即mode_t 参数。
      - `O_DIRECTORY` 若path指向的不为目录，则出错。

        - O_EXCL 若同时指定O_CREATE且文件不存在，则出错。可以将测试文件存在和创建文件封装为原子操作
        - O_NOCTTY 若path引用的是终端设备，则不将该设备分配作为该进程的控制终端
        - O_NONBLOCK 若path引用的是一个FIFO，一个块特殊文件或者字符特殊文件，则此选项将本次文件的打开操作和后续的IO操作设置为非阻塞模式
        - O_SYNC 每次操作需要等待物理IO完成，包括更新文件属性而需要的物理IO
        - O_TURNC 若文件存且为只写或读写打开，那么将其长度截断为零。
        - O_DSYNC 每次写入需要等待物理IO完成，但是如果不影响读取，则不需要更新文件属性
        - O_FSYNC 使每一个 以文件描述符为参数的进行的read操作等待，直到所有对文件同一部分的挂起写操作都完成
- 返回值：
    - 成功 -- 非负整数来表示打开的文件描述符
    - 失败 -- 返回-1，并设置errno
- 需求：
    - 头文件：#include <fcntl.h>
- 注意：
    - open的实现必须检测的错误及相应的错误码

atoi()

- 简介：
    - atoi()是C++标准库中的一个函数，用于将字符串转换为整数。
- 原型：

```
int atoi(const char* str);
```

- 参数：
    - str : 要转换为整数的字符串
- 返回值：
    - 如果转换成功，返回对应的整数值。
    - 如果无法进行有效的转换，返回0
- 详解：
    - 函数功能
        - atoi()函数会尝试将参数str解析为整数。它会从字符串的开头开始解析，直到遇到第一个非数字字符为止（或者字符串结束）。它会忽略前导空格字符。如果字符串为空或无法进行有效的转换，atoi()会返回0
    - 注意事项
        - atoi()函数只能转换纯整数字符串，它不能处理带有小数点、指数符号或其他非数字字符的字符串。
        - 如果字符串表示的整数超出了int类型的范围，结果可能是未定义的。

    - 为了更安全和更灵活的字符串转换，可以使用C++11引入的std::stoi()函数或更高级的字符串转换库，如Boost库中的boost::lexical_cast

- 示例：

```
#include <iostream>
#include <cstdlib>

int main() {
    const char* str = "12345";
    int num = atoi(str);
    std::cout << "Converted integer: " << num << std::endl;
    return 0;
}
```

atol()

- 简述：把参数 str 所指向的字符串转换为一个长整数（类型为 long int 型）
- 声明：`long int atol(const char *str);`
- 参数：
    - `str` -- 要转换为长整数的字符串
- 返回值
    - 成功 -- 返回转换后的长整数
    - 失败 -- 返回零

atof()

- 简述：把参数 str 所指向的字符串转换为一个浮点数（类型为 double 型）
- 声明：`double atof(const char *str);`
- 参数：
    - `str` -- 要转换为浮点数的字符串
- 返回值：
    - 成功 -- 返回转换后的双精度浮点数
    - 失败 -- 返回零(0.0)

clock_gettime()

- 简介：

    - clock_gettime() 是一个 C 标准库函数，用于获取系统时钟的时间值。它提供了更高分辨率的时间测量，可以用于精确计时和性能分析

- 原型：

```
#include <time.h>

int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

- 参数：
  - clk_id：一个表示时钟源的标识符，指定要获取时间值的时钟。常见的时钟源包括
    - CLOCK_REALTIME：表示实时时钟，可用于获取实际时间。
    - CLOCK_MONOTONIC：表示单调时钟，可用于测量经过的时间，不受系统时间的影响。
    - CLOCK_PROCESS_CPUTIME_ID：表示与当前进程相关的 CPU 时间。
    - CLOCK_THREAD_CPUTIME_ID：表示与当前线程相关的 CPU 时间
  - tp：一个指向 struct timespec 结构的指针，用于存储获取的时间值
- 返回值：
  - 成功时，返回 0。
  - 失败时，返回 -1，并设置 errno 变量表示错误的原因
- 详解：
  - clock_gettime() 函数用于获取指定时钟源的时间值。
  - 要使用 clock_gettime() 函数，需要包含头文件 <time.h>。
  - 通过指定不同的时钟源，可以获取不同类型的时间值，如实际时间、单调时间或 CPU 时间。
  - 时间值以 struct timespec 结构的形式返回，该结构包含了秒数和纳秒数的值。
  - clock_gettime() 函数提供了更高精度的时间测量，相比于传统的 time() 函数，可以提供更精确的时间分辨率。
  - 具体可用的时钟源和支持的精度可能因操作系统而异
- 示例：

```c
#include <stdio.h>
#include <time.h>

int main() {
    struct timespec tp;

    // 获取实时时钟的时间值
    if (clock_gettime(CLOCK_REALTIME, &tp) == -1) {
        perror("clock_gettime");
        return 1;
    }

    // 输出时间值的秒数和纳秒数
    printf("Seconds: %ld\n", tp.tv_sec);
    printf("Nanoseconds: %ld\n", tp.tv_nsec);

    return 0;
}
```

- 注：
  - 在此示例中，我们使用 clock_gettime() 函数获取实时时钟的时间值，并将结果存储在 struct timespec 结构 tp 中。然后，我们将秒数和纳秒数分别打印出来

    - 需要注意的是，不同的操作系统可能支持不同的时钟源和精度，并且可能会受到系统时钟的限制。因此，在使用 clock_gettime() 函数时，应该查阅相关文档并了解操作系统的特定行为。

getline()

- 简述：从打开的文件流中按行读取，将保存行字符的缓冲区地址保存到`*line_ptr`
- 声明：`size_t getline(char **lineptr, size_t *n, FILE *stream);`
- 参数：
    - `lineptr` -- 保存读取的字符的缓冲区
    - `n`
    - `stream` -- 使用`fopen`打开的文件句柄
- 返回值：
    - 成功 -- 存储在缓冲区中的字符数，也就是读取的字符数
    - 失败 -- -1
- 需求
    - 头文件：`#include <stdio.h>`
- 注意：
    - `lineptr`，如果在调用函数之前被置为`NULl`，则在函数执行过程中会自动申请内存资源，所以需要在调用失败时，手动释放`lineptr`指向的内存资源

getdelim()

- 简述：
    - 函数应从流读取，直到遇到与定界符字符匹配的字符为止。
    - 定界符参数是一个int，应用程序应确保其为终止读取过程的无符号字符表示的字符。
    - 如果定界符参数具有任何其他值，则行为是不确定的，换言之，从给定文件中读取流，遇到定界符参数就终止
- 声明：`_IO_ssize_t __getdelim(char **__restrict __lineptr, size_t *__restrict __n, int __delimiter, FILE *__restrict __stream);`
- 参数：
    - `lineptr` -- 指向初始缓冲区或空指针的指针
    - `n` -- 指向初始缓冲区大小的指针
    - `delimiter`-- 定界线字符
    - `stream` -- 有效输入流，由`fopen()`打开
- 返回值：
    - 成功 -- 存储在缓冲区中的字符数，包括定界符，但是不包括空字符
    - 失败 -- -1

iscntrl()

- 简述：检查所传的字符是否是控制字符
    - 根据标准 ASCII 字符集，控制字符的 ASCII 编码介于 0x00 (NUL) 和 0x1f (US) 之间，以及 0x7f (DEL)，某些平台的特定编译器实现还可以在扩展字符集（0x7f 以上）中定义额外的控制字符
- 声明：`int iscntrl(int c);`
- 参数：
    - `c` -- 要检查的字符
- 返回值：
    - 如果c 是一个控制字符，则返回非零值

    - 否则，返回0
- 需求：
    - 头文件：`#include <ctype.h>`

strcpy()

- 简述：把 `src` 所指向的字符串复制到 `dest`
- 声明：`char *strcpy(char *dest, const char *src);`
- 参数：
    - dest -- 指向用于存储复制内容的目标数组
    - src -- 要复制的字符串
- 返回值：
    - 成功 -- 该函数返回一个指向最终的目标字符串 dest 的指针
    - 失败 -- NULL
- 注意：
    - 如果目标数组 dest 不够大，而源字符串的长度又太长，可能会造成缓冲溢出的情况

strncpy()

- 简述：把 src 所指向的字符串复制到 dest，最多复制 n 个字符。当 src 的长度小于 n 时，dest 的剩余部分将用空字节填充。
- 声明：`char *strncpy(char *dest, const char *src, size_t n);`
- 参数：
    - `dest` -- 指向用于存储复制内容的目标数组。
    - `src` -- 要复制的字符串。
    - `n` -- 要从源中复制的字符数。
- 返回值：
    - 返回最终复制的字符串
- 注意：
    - strncpy 没有自动加上终止符的，需要手动加上不然会出问题的。

`stdio.h` 库宏

- `NULL` -- 这个宏是一个空指针常量的值。
- `_IOFBF`、`_IOLBF` 和 `_IONBF` -- 这些宏扩展了带有特定值的整型常量表达式，并适用于 `setvbuf` 函数的第三个参数
- `BUFSIZ` -- 这个宏是一个整数，该整数代表了 `setbuf` 函数使用的缓冲区大小
- `EOF` -- 这个宏是一个表示已经到达文件结束的负整数
- `FOPEN_MAX` -- 这个宏是一个整数，该整数代表了系统可以同时打开的文件数量。
- `FILENAME_MAX` -- 这个宏是一个整数，该整数代表了字符数组可以存储的文件名的最大长度。如果实现没有任何限制，则该值应为推荐的最大值
- `L_tmpnam` -- 这个宏是一个整数，该整数代表了字符数组可以存储的由 `tmpnam` 函数创建的临时文件名的最大长度
- `SEEK_CUR`、`SEEK_END` 和 `SEEK_SET` -- 这些宏是在 `fseek` 函数中使用，用于在一个文件中定位不同的位置
- `TMP_MAX` -- 这个宏是 `tmpnam` 函数可生成的独特文件名的最大数量

- `stderr`、`stdin` 和 `stdout` -- 这些宏是指向 `FILE` 类型的指针，分别对应于标准错误、标准输入和标准输出流

`stdio.h`库函数

- `int fclose(FILE *stream);` -- 关闭流 `stream`。刷新所有的缓冲区。
- `void clearerr(FILE *stream);` -- 清除给定流 `stream` 的文件结束和错误标识符。
- `int feof(FILE *stream);` -- 测试给定流 `stream` 的文件结束标识符
- `int ferror(FILE *stream);` -- 测试给定流 `stream` 的错误标识符
- `int fflush(FILE *stream);` -- 刷新流 `stream` 的输出缓冲区
- `int fgetpos(FILE *stream, fpos_t *pos);` -- 获取流 `stream` 的当前文件位置，并把它写入到 `pos`
- `FILE *fopen(const char *filename, const char *mode);` -- 使用给定的模式 `mode` 打开 `filename` 所指向的文件
- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);` -- 从给定流 `stream` 读取数据到 `ptr` 所指向的数组中
- `FILE *freopen(const char *filename, const char *mode, FILE *stream);` -- 把一个新的文件名 `filename` 与给定的打开的流 `stream` 关联，同时关闭流中的旧文件
- `int fseek(FILE *stream, long int offset, int whence);` -- 设置流 `stream` 的文件位置为给定的偏移 `offset`，参数 `offset` 意味着从给定的 `whence` 位置查找的字节数
- `int fsetpos(FILE *stream, const fpos_t *pos);` -- 设置给定流 `stream` 的文件位置为给定的位置。参数 `pos` 是由函数 `fgetpos` 给定的位置
- `long int ftell(FILE *stream);` -- 返回给定流 `stream` 的当前文件位置
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);` -- 把 `ptr` 所指向的数组中的数据写入到给定流 `stream` 中。
- `int remove(const char *filename);` -- 删除给定的文件名 `filename`，以便它不再被访问
- `int rename(const char *old_filename, const char *new_filename);` -- 把 `old_filename` 所指向的文件名改为 `new_filename`
- `void rewind(FILE *stream);` -- 设置文件位置为给定流 `stream` 的文件的开头
- `void setbuf(FILE *stream, char *buffer);` -- 定义流 `stream` 应如何缓冲。
- `int setvbuf(FILE *stream, char *buffer, int mode, size_t size);` -- 另一个定义流 `stream` 应如何缓冲的函数
- `FILE *tmpfile(void);` -- 以二进制更新模式(`wb+`)创建临时文件
- `char *tmpnam(char *str);` -- 生成并返回一个有效的临时文件名，该文件名之前是不存在的

- `int fprintf(FILE *stream, const char *format, ...);` -- 发送格式化输出到流 `stream` 中
- `int printf(const char *format, ...);` -- 发送格式化输出到标准输出 `stdout`
- `int sprintf(char *str, const char *format, ...);` -- 发送格式化输出到字符串
- `int vfprintf(FILE *stream, const char *format, va_list arg);` -- 用参数列表发送格式化输出到流 `stream` 中
- `int vprintf(const char *format, va_list arg);` -- 使用参数列表发送格式化输出到标准输出 `stdout`
- `int vsprintf(char *str, const char *format, va_list arg);` -- 使用参数列表发送格式化输出到字符串
- `int fscanf(FILE *stream, const char *format, ...);` -- 从流 `stream` 读取格式化输入
- `int scanf(const char *format, ...);` -- 从标准输入 `stdin` 读取格式化输入
- `int sscanf(const char *str, const char *format, ...);` -- 从字符串读取格式化输入
- `int fgetc(FILE *stream);` -- 从指定的流 `stream` 获取下一个字符（一个无符号字符），并把位置标识符往前移动
- `char *fgets(char *str, int n, FILE *stream);` -- 从指定的流 `stream` 读取一行，并把它存储在 `str` 所指向的字符串内。当读取 (n-1) 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定
- `int fputc(int char, FILE *stream);` -- 把参数 `char` 指定的字符（一个无符号字符）写入到指定的流 `stream` 中，并把位置标识符往前移动
- `int fputs(const char *str, FILE *stream);` -- 把字符串写入到指定的流 `stream` 中，但不包括空字符
- `int getc(FILE *stream);` -- 从指定的流 `stream` 获取下一个字符（一个无符号字符），并把位置标识符往前移动
- `int getchar(void);` -- 从标准输入 `stdin` 获取一个字符（一个无符号字符）
- `char *gets(char *str);` -- 从标准输入 `stdin` 读取一行，并把它存储在 `str` 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定
- `int putc(int char, FILE *stream);` -- 把参数 `char` 指定的字符（一个无符号字符）写入到指定的流 `stream` 中，并把位置标识符往前移动
- `int putchar(int char);` -- 把参数 `char` 指定的字符（一个无符号字符）写入到标准输出 `stdout` 中
- `int puts(const char *str);` -- 把一个字符串写入到标准输出 `stdout` ，直到空字符，但不包括空字符。换行符会被追加到输出中

- `int ungetc(int char, FILE *stream);` -- 把字符 `char`（一个无符号字符）推入到指定的流 `stream` 中，以便它是下一个被读取到的字符
- `void perror(const char *str);` -- 把一个描述性错误消息输出到标准错误 stderr。首先输出字符串 `str`，后跟一个冒号，然后是一个空格
- `int snprintf(char *str, size_t size, const char *format, ...);` -- 格式字符串到 `str` 中

fopen()

- 简述：使用给定的模式(mode)打开(filename)所指向的文件
- 声明：`FILE *fopen(const char *filename, const char *mode);`
- 参数：
  - `filename` -- 字符串，表示要打开的文件名称
  - `mode` -- 字符串，表示文件的访问模式
    - `r` -- 打开一个用于读取的文件。该文件必须存在
    - `w` -- 创建一个用于写入的空文件。如果文件名称与已存在的文件相同，则会删除已有文件的内容，文件被视为一个新的空文件
    - `a` -- 追加到一个文件。写操作向文件末尾追加数据。如果文件不存在，则创建文件
    - `b` -- 以二进制模式打开文件
    - `r+` -- 打开一个用于更新的文件，可读取也可写入。该文件必须存在
    - `w+` -- 创建一个用于读写的空文件
    - `a+` -- 打开一个用于读取和追加的文件
- 返回值：
  - 成功 -- 返回一个`FILE`指针
  - 失败 -- 返回`NULL`，且设置全局变量`errno`来标识错误

fwrite()

- 简述：把 `ptr` 所指向的数组中的数据写入到给定流 `stream` 中
- 声明：`size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
- 参数：
  - `ptr` -- 这是指向要被写入的元素数组的指针
  - `size` -- 这是要被写入的每个元素的大小，以字节为单位
  - `nmemb` -- 这是元素的个数，每个元素的大小为 size 字节
  - `stream` -- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个输出流
- 返回值：
  - 成功 -- 返回一个 size_t 对象，表示元素的总数，该对象是一个整型数据类型
  - 失败 -- 返回一个错误

fprintf()

- 简述：发送格式化输出到流 stream 中
- 声明：`int fprintf(FILE *stream, const char *format, ...);`
- 参数：
  - stream -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

    - format -- 这是 C 字符串，包含了要被写入到流 stream 中的文本。
        - 它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。
        - format 标签属性是 `%[flags][width][.precision][length]specifier`
- 返回值：
    - 成功 -- 返回写入的字符总数
    - 失败 -- 返回一个负数

fflush()

- 简述：刷新流 `stream` 的输出缓冲区
- 声明：`int fflush(FILE *stream);`
- 参数：
    - `stream` -- 这是指向 `FILE` 对象的指针，该 `FILE` 对象指定了一个缓冲流
- 返回值：
    - 成功 -- 返回零值
    - 失败 -- 返回 EOF，且设置错误标识符（即 feof）
- 头文件：
    - `stdio.h`

fseek()

- 简述：设置流 `stream` 的文件位置为给定的偏移 `offset`，参数 `offset` 意味着从给定的 `whence` 位置查找的字节数。
- 声明：`int fseek(FILE *stream, long int offset, int whence);`
- 参数：
    - stream -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流
    - offset -- 这是相对 whence 的偏移量，以字节为单位
    - whence -- 这是表示开始添加偏移 offset 的位置。它一般指定为下列常量之一：
        - SEEK_SET -- 文件的开头
        - SEEK_CUR -- 文件指针的当前位置
        - SEEK_END -- 文件的末尾
- 返回值：
    - 成功 -- 返回零值
    - 失败 -- 返回非零值

ftell()

- 简述：返回给定流 stream 的当前文件位置
- 声明：`long int ftell(FILE *stream);`
- 参数：
    - stream -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流
- 返回值：
    - 成功 -- 该函数返回位置标识符的当前值
    - 失败 -- 返回 -1L，全局变量 errno 被设置为一个正值。

fgets()

- 简述：从指定的流 stream 读取一行，并把它存储在 str 所指向的字符串内
- 声明：`char *fgets(char *str, int n, FILE *stream);`
- 参数：
    - str -- 这是指向一个字符数组的指针，该数组存储了要读取的字符串。
    - n -- 这是要读取的最大字符数（包括最后的空字符）。通常是使用以 str 传递的数组长度
    - stream -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要从中读取字符的流。
- 返回值：
    - 成功 -- 返回相同的 str 参数。如果到达文件末尾或者没有读取到任何字符，str 的内容保持不变，并返回一个空指针。
    - 失败 -- 返回一个空指针。
- 注意：
    - 当读取 (n-1) 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定
- 简介：
    - fgets()函数是C标准库中用于从文件中读取一行数据的函数。
- 原型：

```
char *fgets(char *str, int size, FILE *stream);
```

- 参数：
    - str：指向字符数组的指针，用于存储读取的数据。
    - size：指定要读取的最大字符数（包括换行符），即字符数组str的大小
    - stream：指定要从中读取数据的文件流指针
- 返回值：
    - 如果成功读取到数据，fgets()会返回指向字符数组str的指针
    - 如果遇到文件结束或发生读取错误，fgets()会返回NULL
- 详解：
    - fgets()函数从指定的文件流stream中读取一行数据（包括换行符），并将其存储到字符数组str中。读取的数据包括换行符，且最多读取size - 1个字符。如果一行数据的长度超过了size - 1个字符，则fgets()会截断剩余的字符
- 示例：

```
char buffer[100];
FILE *file = fopen("example.txt", "r");
if (file != NULL) {
    if (fgets(buffer, sizeof(buffer), file) != NULL) {
```

```
            printf("Read line: %s", buffer);
        }
        fclose(file);
    }
```

- 注：
    - 上述示例中，打开名为"example.txt"的文件并将其用于读取，然后调用fgets()函数从文件中读取一行数据存储到buffer数组中，并通过printf()函数将读取的行打印出来。最后关闭文件

rewind()

- 简述：设置文件位置为给定流 `stream` 的文件的开头
- 声明：`void rewind(FILE *stream);`
- 参数：
    - `stream` -- 这是指向 `FILE` 对象的指针，该 `FILE` 对象标识了流。
- 返回值：
    - 无
- 头文件：
    - `stdio.h`

access()

- 简述：确定文件或文件夹的访问权限。即，检查某个文件的存取方式，例如说是只读方式，只写方式等。
- 声明：`int access (const char *__name, int __type);`
- 参数
    - `name` -- 文件路径
    - `type` -- 判断参数
        - `R_OK 4` -- 读权限
        - `W_OK 2` -- 写权限
        - `X_OK 1` -- 执行权限
        - `F_OK 0` -- 是否存在
- 返回值：
    - 指定的存取方式有效，则函数返回0，
    - 指定的存取方式无效，则返回-1
- 需求：
    - 头文件：`#include <unistd.h>`

remove()

- 简介
    - remove() 是C标准库中的一个函数，用于删除文件
- 原型：

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

```
int remove(const char *filename);
```

- 参数：
  - remove() 函数接受一个字符串参数 filename，表示要删除的文件名（包括路径）
- 返回值：
  - 如果文件删除成功，则返回值为 0
  - 如果删除失败，则返回一个非零值。
- 详解：
  - 需要注意的是，remove() 函数只能删除普通的文件，无法删除目录。如果要删除目录，需要使用操作系统提供的特定函数或命令
- 示例：

```
#include <stdio.h>

int main() {
    const char *filename = "file.txt";

    if (remove(filename) == 0) {
        printf("文件删除成功\n");
    } else {
        printf("文件删除失败\n");
    }

    return 0;
}
```

- 注：
  - 在上述示例中，我们尝试删除名为 "file.txt" 的文件。如果文件删除成功，则输出 "文件删除成功"，否则输出 "文件删除失败"
  - 需要注意的是，删除文件是一个具有风险的操作，因此在执行删除操作之前应该谨慎确认。确保要删除的文件存在且不再需要，以免误删重要文件

rename()

- 简述：重命名一个文件
- 声明：`int rename(const char* old, const char* new);`
- 参数：
  - `old` -- 旧的文件名
  - `new` -- 新的文件名
- 返回值：
  - 成功 --
  - 失败 --
- 需求：

- `#include <stdio.h>`

assert()

- 简述：C库宏 允许诊断信息被写入到标准错误文件中。换句话说，它可用于在C程序中添加诊断
- 声明：`void assert(int expression)`
- 参数：
  - `expression`:这可以是一个变量或任何C表达式。
    - 如果`expression`为真，`assert()`不执行任何动作。
    - 如果`expression`为假，`assert()`会在标准错误`stderr`上显示错误消息，并中止程序执行。

getenv()

- 简述：搜索 name 所指向的环境字符串，并返回相关的值给字符串，
- 声明：`#include <stdlib.h> char* getenv (const char* name);`
- 参数：
  - name -- 包含被请求变量名称的C字符串
- 返回值：
  - 成功 -- 返回一个以NULL结尾的字符串，该字符串为被请求环境变量的值
  - 失败 -- 返回NULL
- 注意：
  - 如果没有将第一次返回的字符串拷贝到缓冲区中去，多次调用`getenv`就要特别小心
  - `getenv`的某些实现为返回字符串使用了静态缓冲区，**每次调用时都会重写缓冲区**

to_string()

- 简述:一个字符串对象，包含val作为字符序列的表示形式
- 声明：`string to_string(int val);`
- 参数：
  - `val` -- 数值

strcat()

- 简述：将源字符串的副本附加到目标字符串。目标中的终止空字符被源的第一个字符覆盖，并且在目标中两者连接形成的新字符串的末尾包含一个空字符。
- 声明：`char* strcat(char* destination, const char* source);`
- 参数：
  - `destination` ： 指向目标数组的指针，它应该包含一个C字符串，并且足够大以包含连接的结果字符串。
  - `source`:要附加的C字符串，这部应该与目标字符串相同。

puts()

- 简述：
  - 将字符串写入标准输出。将str指向的C字符串写入标准输出(stdout)并且附加一个换行符(`'\n'`)。
  - 该函数从指定的地址（str）开始复制，直到到达终止空字符（'\0'）。此终止空字符不会复制到流中

    - 注意，puts不仅与fputs不同，它使用标准输出作为目标，而且它还在末尾自动附加一个环行符（fputs则没有）
- 声明：`int puts(const char* str);`
- 参数：
    - `str`：需要打印的C字符串
- 返回值
    - 成功 -- 返回一个非负值
    - 失败 -- 该函数返回EOF并且设置错误指示符(ferror)

sprintf()

- 简述：发送格式化输出到str所指向的字符串
- 声明： `int sprintf(char *str, const char* format, ... )`
- 参数：
    - `str`：这是指向一个字符数组的指针，该数组存储了C字符串
    - `format`:这是字符串，包含了要被写入到字符串`str`的文本。它可以包含嵌入的`format`标签，`format`标签可被随后的附加参数中指定的值替换，并按需求进行格式化。`format`标签属性是`%[flags][width][.precision][length]specifier`
    - 具体详情：
    - `specifier, 类型说明符`：
        - `c`:字符
        - `d, i`:有符号的十进制整数
        - `e`:使用`e`字符的科学计数法（尾数和指数）
        - `E`:使用`E`字符的科学计数法（尾数和指数）
        - `f`:十进制浮点数
        - `g`:自动选择`%e`或`%f`中合适的表示法
        - `G`:自动选择`%E`或`%f`中合适的表示法
        - `o`:有符号八进制
        - `s`:字符的字符串
        - `u`:无符号十进制整数
        - `x`:无符号十六进制整数
        - `X`:无符号十六进制整数（大写字母）
        - `p`:指针地址
        - `n`:无输出
        - `%`:字符
- 返回值：
    - 成功 -- 返回写入的字符总数，不包括字符串追加在字符串末尾的空字符
    - 失败 -- 返回一个负数
- 需求：
    - 头文件：`#include <stdio.h>`

snprintf()

- 简述：将格式化的输出写入大小适中的缓冲区
- 声明：`int snprintf(char* s, size_t n, const char* format, ...);`
- 参数：
    - `s` -- 指向存储C字符串的缓冲区的指针。这个缓冲区的大小至少要能存储`n`个字符

    - `n` -- 在缓冲区存储的最大字节数。通常，生成的字符串最大长度为`n-1`，留下一个空间存储空字符
    - `format` -- 和printf格式一样，C字符串要存储的格式
- 返回值
    - 如果写入正常，返回被写入的字符数
    - 如果错误，返回一个负数
    - 只有当返回的不是一个负数，且小于`n`时，字符串才被完全写入

sscanf()

- 简述：
    - 从s读取数据并根据参数格式将它们存储到附加参数给出的位置，就像使用了scanf一样，但从s而不是标准输入（stdin）读取数据。
    - 附加参数应该指向已分配的对象，其类型由格式字符串中的相应格式说明符指定。
- 声明：`int sscanf(const char* s, const char* format, ...);`
- 参数：
    - `s` -- 存储数据的指针
    - `format` -- 格式化字符串
- 返回值：
    - 成功 -- 返回参数列表中成功填充的项目数。
    - 失败 -- 匹配预期的项目数或更少（甚至为零）
- 需求：
    - 头文件：`#include <stdio.h>`

c_str()

- 返回指向包含以null结尾的字符序列（即C字符串）的数组的指针，该字符序列表示字符串对象的当前值。
- 这个数组包含构成字符串对象值的相同字符序列，以及末尾的附加终止空字符`\0`
- 返回值：指向字符串对象值的C字符串表示形式的指针。

strtok()

- 简述： 分解字符串 str 为一组字符串，delim 为分隔符
- 声明：`char *strtok(char *str, const char *delim);`
- 参数：
    - str -- 要被分解成一组小字符串的字符串。
    - delim -- 包含分隔符的 C 字符串。
- 返回值：
    - 函数返回被分解的第一个子字符串
    - 如果没有可检索的字符串，则返回一个空指针。
- 注意：
    - 首次调用时，`str`指向要分解的字符串，之后再次调用要把`str`设置称为NULL
    - 当strtok()在参数s的字符串中发现参数delim中包含的分割字符时,则会将该字符改为\0 字符。
    - 在第一次调用时，strtok()必需给予参数s字符串，往后的调用则将参数s设置成NULL。每次调用成功则返回指向被分割出片段的指针
    - 需要注意的是，**使用该函数进行字符串分割时，会破坏被分解字符串的完整，调用前和调用后的s已经不一样了**。

    - 第一次分割之后，原字符串str是分割完成之后的第一个字符串，剩余的字符串存储在一个静态变量中，因此多线程同时访问该静态变量时，则会出现错误。

strtok_r()

- 简述：linux下分割字符串的安全函数
- 声明：`char *strtok_r(char *str, const char *delim, char **saveptr);`
- 参数：
    - str -- 要被分解成一组小字符串的字符串。
    - delim -- 包含分隔符的 C 字符串。
    - saveptr -- 保存剩余的字符
- 返回值：
    - 函数返回被分解的第一个子字符串
    - 如果没有可检索的字符串，则返回一个空指针。
- 注意：
    - 该函数也会破坏带分解字符串的完整性，但是其将剩余的字符串保存在saveptr变量中，保证了安全性

strchr和sscanf组合，按指定分隔符分割字符串

strtol()

- 简述：
    - 解析C字符串str，将其内容解释为指定基数的整数，该基数作为long int返回。如果endptr不是空指针，该函数还将endptr的值设置为指向数字后的第一个字符。
    - 该函数首先根据需要丢弃尽可能多的空白字符，直到找到第一个非空白字符。然后，从这个字符开始，按照依赖于基本参数的语法获取尽可能多的有效字符，并将它们解释为数值。
    - 最后，指向str中整数表示之后的第一个字符的指针存储在endptr指向的对象中。
- 声明： `long int strtol (const char* str, char** endptr, int base);`
- 参数：
    - `str`:以整数表示的C字符串开头(C-string beginning with the representation of an integral number)
    - `endptr`:引用`char*`类型的对象，其值由函数设置为`str`中数值之后的下一个字符。
    - `base`:确定有效字符及其解释的数字基数
- 返回值：
    - 成功时，该函数将转换后的整数作为long int值返回
    - 如果无法执行有效转换，则返回零值
- 需求：
    - 头文件：`#include <stdlib.h>`

reserve()

- 简述：请求更改容量
- 声明：`void reserve (size_type n);`
- 参数：
    - `n` ： 向量的最小容量。

C标准库 -- `stdarg.h`

- 简介：
  - stdarg.h头文件定义了一个变量类型va_list和三个宏，
  - 这三个宏可用于在参数个数未知（即**参数个数可变**）时获取函数中的参数。
  - 可变参数的函数通过在参数列表的末尾是使用省略号（...）定义的。
- **库变量：**
  - va_list ：这是一个适用于va_start(),va_arg()和 va_end()这三个宏存储信息的类型。
- **库宏：**
  - void va_start(va_list ap, last_arg)：
    - 这个宏初始化ap变量，它与va_arg和va_end宏是一起使用的。last_arg是最后一个传递给函数的已知的固定参数，即省略号之前的参数。这个宏必须在使用va_arg和va_end之前被调用
    - 参数：
      - ap -- 这是一个va_list类型的对象，它用来存储通过va_arg获取额外参数时所必需的信息。
      - last_arg -- 最后一个传递给函数的已知的固定参数
  - void va_end(va_list ap):
    - C库宏void va_end(va_list ap)允许使用了va_start宏的带有可变参数的函数返回。如果在从函数返回之前没有调用va_end，则结果为未定义
    - 参数：
      - ap -- 这是之前由同一函数中的va_start初始化的va_list对象。

std::get()

- 简述：
  - 返回对元组tpl的第I个元素的引用。
  - 版本2将元组的右值作为引用参数，向前应用到返回的元素
  - 版本3将const元组作为参数，返回对元素的const引用
- 声明：

```
  (1)template <size_t I, class... Types> typename tuple_element<I,
tuple<Types...>>::type& get(tuple<Types...>& tpl) noexcept;
  (2)template <size_t I, class... Types> typename tuple_element<I,
tuple<Types...>>::type&& get(tuple<Types...>&& tpl) noexcept;
  (3)template <size_t I, class... Types> typ
  ename tuple_element<I, tuple<Types...>>::type const&
get(tuple<Types...>& tpl) noexcept;
```

- 参数
  - I -- 元组中元素的位置，0为第一个元素的位置，size_t是无符号整数类型
  - Types -- 元组中元素的类型（通常从tpl隐式获得）
- 返回值
  - 对元组中指定位置的元素的引用

fstat()

- 简述：获取报告与打开的文件描述符`fildes`有关的文件的状态信息
- 声明：`int fstat(int __fd, struct stat *__buf);`
- 参数：
  - `__fd` -- 文件描述符
  - `__buf` -- 内存区域指针，用户提供的缓冲区，`fstat`将信息写入这个缓冲区
- 返回值：
  - 成功 -- 0
  - 失败 -- 1
- 需求：
  - 头文件：`#include <sys/stat.h>`

fmemopen()

- 简述：创建一个新的引用内存缓冲区的流
- 声明：`FILE* fmemopen(void *s, size_t len, const char *modes);`
- 参数：
  - `s` --
  - `len` --
  - `modes` --
- 返回值：
  - 成功 -- 返回创建的流指针
  - 失败 --
- 注意：
  - 虽然仍使用FILE指针进行访问，但其实并没有底层文件（并没有磁盘上的实际文件，因为打开的内存流fp是在内存中的）
  - 所有的I/O都是通过在缓冲区与主存（就是内存）之间来回传送字节来完成的

mmap()

- mmap(memory map，即地址的映射)，是一种内存映射文件的方法，将一个文件或者其他对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对应关系。

- `mmap()`系统调用，使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以访问普通内存一样对文件进行访问，不必再调用`read(), write()`等操作。

- mmap()系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。而POSIX或系统V的共享内存IPC则只是用于共享目的。

- Linux通过内存映像机制来提供用户程序对内存直接访问的能力。内存映像的意思是把内核中特定部分的内存空间映射到用户级程序的内存空间去。也就是说，用户空间和内核空间共享一块相同的内存。

- 相对于传统的write/read IO系统调用，必需先把数据从磁盘拷贝至内核缓冲区（页缓冲），然后再把数据拷贝至用户进程中。两者相比，mmap会少一次拷贝数据，这样带来的性能提升是巨大的

- 声明：`void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset);`

- 参数
  - `addr` -- 指定文件应被映射到进程空间的起始地址，一般被指定一个空指针，此时选择起始地址的任务留给内核来完成
  - `length` -- 指的是映射到调用进程地址空间的字节数，它从被映射文件开头offset各字节开始算起
  - `prot` -- 指定共享内存的访问权限
  - `flags` -- 常值 ： MAP_SHARED, MAP_PRIVATE, MAP_PIXED`
  - `offset` -- 一般设置为0，表示从文件头开始映射
  - `fd` -- 为即将映射到进程空间的文件描述字，一般由`open()`返回。
- 返回值
  - 函数的返回值为最后文件映射到进程空间的地址，进程可直接操作其是地址为该值的有效地值。
- 需求：
  - 头文件：`#include <sys/mman.h>`
- 具体原理：`https://blog.csdn.net/Holy_666/article/details/86532671`

munmap()

- 简述：解除任何内存映射
- 声明：`int munmap(void* __addr, size_t __len);`
- 参数：
  - `addr` -- 内存地址
  - `__len` -- 内存地址大小
- 返回值
  - 成功 -- 0
  - 失败 -- -1
- 需求：
  - 头文件：`#include <sys/mman.h>`

锁定物理内存 -- mlock家族

- 锁住内存是为了防止这段内存被操作系统交换掉(swap)，并且由于此操作风险高，仅超级用户可以执行。
- 家族成员

```
#include <sys/mman.h>
int mlock(const void* addr, size_t len);
int munlock(const void* addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

- `getpagesize()` 函数返回系统的分页大小，在X86 Linux系统上，这个值是4KB

- 如果希望程序的全部地址空间被锁定在物理内存中，使用 `mlockall` ，该函数将调用进程的全部虚拟地址空间加锁，防止出现内存交换，将该进程的地址空间交换到外存上
- `mlockall()` 将所有映射到进程地址空间的内存上锁，这些页包括 -- 代码段，数据段，栈段，共享库，共享内存, user space kernel data, memory-mapped file。当函数成功返回的时候，所有的被映射的页都在内存中
- 参数
    - `MCL_CURRENT` -- 仅当前已分配的内存会被锁定，之后分配的内存则不会
    - `MCL_FUTURE` -- 锁定之后分配的所有内存
    - `MCL_CURRENT|MCL_FUTURE` -- 将已经以及将来分配的所有内存锁定在物理内存中
- 返回值
    - 成功返回`0`
    - 出错返回`-1`
- 此函数有两个重要的应用：
    - `real-time algorithms（实时算法）` -- 对事件要求非常高
    - `high-scurity data processing（机密数据的处理）` -- 如果数据被交换到外存上，可能会泄密
- 如果进程执行了一个`execve`类函数，所有的锁都会被删除
- 内存锁不会被子进程继承
- 内存锁不会叠加，即使多次调用`mlockall()`函数，只调用一次`munlockall()`就会解锁

clock_gettime()

- 简述：用于计算时间，有秒和纳秒两种精度
- 函数声明：`int clock_gettime(clockid_t clk_id, struct timespec *tp);`
- 参数：
    - `clockid_t clk_id`有四种
        - `CLOCK_REALTIME` -- 系统实时时间，随系统实时时间改变而改变
        - `CLOCK_MONOTONIC` -- 从系统启动这一刻开始计时，不受系统时间被用户改变的影响
        - `CLOCK_PROCESS_CPUTIME_ID` -- 本进程到当前代码系统CPU花费的时间
        - `CLOCK_THREAD_CPUTIME_ID` -- 本线程到当前代码系统CPU花费的时间

vsscanf()

- 简述：**将格式化数据从字符串读取到变量参数列表中**。从s读取数据并根据参数格式将它们存储到由arg标识的变量参数列表中的元素所指向的位置。
- 声明：`int vsscanf(const char* s, const char* format, va_list arg);`
    - 在内部，该函数从由arg标识的列表中检索参数，就好像在其上使用了`va_arg`一样，因此`arg`的状态可能会被调用更改。
    - 无论如何，`arg`应该在调用之前的某个时间点由`va_start`初始化，并且预计在调用之后的某个时间点由`va_end`释放。
- 参数

    - `s` -- 函数将其处理为检索数据的源的C字符串
    - `format` -- 包含**格式字符串**的C字符串，该格式字符串遵循与`scanf`中的格式相同的规范
    - `arg` -- 一个值，用于标识使用`va_start`初始化的变量参数列表。`va_list`是在`<cstdarg>`中定义的特殊类型。
- 返回值：
    - 成功时，该函数返回参数列表中成功填充的项目数
    - 在匹配失败的情况下，此计数可以匹配预期的项目数或更少，甚至为零。
    - 如果在成功解释任何数据之前输入失败，则返回EOF

memset()

- 简介：
    - memset() 是 C 标准库中的一个函数，用于将一块内存区域的每个字节设置为指定的值
- 原型：

```
#include <string.h>

void *memset(void *ptr, int value, size_t num);
```

- 参数：
    - ptr：指向要填充的内存区域的起始地址的指针。
    - value：要设置的值，以 int 类型表示。
    - num：要设置的字节数，即要填充的内存区域的大小
- 返回值：
    - 返回 ptr 的指针
- 详解：
    - memset() 函数用于将一块内存区域的每个字节设置为指定的值。
    - 要使用 memset() 函数，需要包含头文件 <string.h>。
    - ptr 是一个指向要填充的内存区域的指针，可以是任何类型的指针。
    - value 是要设置的值，可以是 int 类型的任何有效值。通常使用 0 或 -1（表示全部位设置为 1）来初始化内存区域。
    - num 表示要填充的字节数，即要设置的内存区域的大小。
    - memset() 函数将指定的值复制到内存区域中的每个字节，可以用来初始化数组、清除敏感数据等。
    - 返回的指针与 ptr 相同，指向被填充的内存区域的起始地址
- 示例：

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

```c
int main() {
    int arr[5];

    // 使用 memset() 将数组元素设置为 0
    memset(arr, 0, sizeof(arr));

    // 打印数组元素
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

- 注：
  - 在此示例中，我们声明了一个包含 5 个整型元素的数组 arr。然后，我们使用 memset() 函数将数组的所有元素设置为零，通过将指向数组的指针、要设置的值（0）、以及要填充的字节数（sizeof(arr)）传递给 memset() 函数来实现。最后，我们打印数组的元素，可以看到它们都被设置为零
  - 需要注意的是，使用 memset() 函数时要小心，确保不会超出要填充的内存区域的边界。同时，对于包含非 char 类型元素的数组，使用 memset() 函数进行初始化时要确保所设置的值类型正确，避免产生类型不匹配的问题

max()

- 简述：返回a和b中的最大值。如果两者相等，则返回a
- 声明：`template <class T> const T& max (const T& a, const T& b);`
- 返回值
  - 作为参数传递的最大值