CS100 习题课 4

来自 刘晓

Contents

Pointer Aliasing

- Pointer Types
- Pointer Aliasing
  - Strict Aliasing
- Legal Pointer Type Conversion
- `void *`

Pointer Types

- 指针的类型表示为 `PointeeType *`。
- 对于两个不同的类型 `T1` 和 `T2`，指针类型 `T1 *` 与 `T2 *` 被视为 **不同的类型**，*即便它们指向同一内存位置*。

```
int i = 42;
float *fp = &i;
++*fp; // 未定义行为；这与 ++i 不等价。
```

- 在 C 语言中，不同类型之间的指针可以进行隐式转换（尽管通常会伴随警告）；然而，这种做法是 **极不安全** 的，在 C++ 中是严格禁止的。

Pointer Aliasing (指针别名)

在 C 和 C++ 中，当多个指针引用同一内存位置时，就发生了 **别名** 现象。

```
void foo(double* a, double* b, int n) {
  for (int i = 0; i < n; i++) {
    b[i] += a[i];
  }
}
```

- 如果 `a` 与 `b` 指向 **独立** 的内存区域，编译器可以对循环进行优化（例如通过并行化）。
- 如果 `a` 与 `b` 存在别名（即内存重叠），则上述优化可能导致 **错误行为**。

Strict Aliasing（严格别名）

严格别名规则允许编译器假定不同类型的指针不会引用相同的内存位置。这一假设是实现诸如 指令重排序 和向量化等优化的基础。

- **Compiler Optimizations:**
  GCC 和 Clang 等编译器在 `-O2` 及更高级别下严格执行别名规则。
- **Risks:**
  违反严格别名规则——通过指向 ***不兼容类型*** 的指针访问内存——会因编译器基于该假设进行优化而导致 ***未定义行为***。

Undefined Behavior Due to Strict Aliasing Violation

```c
#include <stdio.h>

void update(int *ip, float *fp) {
  int x = *ip;
  *fp = 1.0f;
  int y = *ip;
  if (x != y)
    printf("Not same.\n");
  else
    printf("Same.\n");
}

int main(void) {
  int a = 42;
  update(&a, (float *)&a);
  return 0;
}
```

使用如下命令编译该程序：

```
gcc main.c -O2 -o main
```

可能会产生输出：

```
Same.
```

该行为为 ***未定义***，因为 `int *` 与 `float *` 违反了严格别名规则。

Legal Pointer Type Conversion - CppReference

当对一个实际指向类型为 `T2` 的对象的 `T1*` 指针进行解引用时，会导致 ***未定义行为***，

除非 `T1` 与 `T2` 满足下列条件之一：

- **Compatible Types** - CppReference
  `T1` 与 `T2` 必须具有足够的 ***相似性*** 或完全相同。

```
int x = 1;
int *p = &x;
printf("%d\n", *p);  // *p 为 int 类型，与 x 的类型相同。
```

Legal Pointer Type Conversion (cont.)

- **Qualification Conversion:**
  当 `T2` 为一个经过 `const`、`volatile` 或 `restrict` 限定的与 `T1` 兼容的类型时。

```
int x = 1;
const int *p = &x;
printf("%d\n", *p);   // *p 为 'const int' 类型，与 'int' 兼容。
```

- **Signed/Unsigned Conversion:**
  如果 `T2` 是与 `T1` 兼容的类型的有符号或无符号版本。

```
int x = 1;
unsigned int *p = (unsigned int*)&x;
printf("%u\n", *p);   // *p 为 'unsigned int' 类型。
```

Legal Pointer Type Conversion (cont.)

- **Character Types:**
  如果 `T2` 为字符类型（`char`、`signed char` 或 `unsigned char`），则指向 `T2` 的指针可以合法转换。

  ```
  int x = 65;
  char *p = (char *)&x;
  printf("%c\n", *p); // *p 为 'char' 类型。
  ```

  这种转换在检查原始内存的函数中被广泛应用，因为字符指针提供了一种标准化的访问任意数据的机制。

有关正式规范，请参阅 C11 标准 (ISO/IEC 9899:201x, Section 6.5 Expressions, Paragraph 7)。

Pointer Type: `void *`

指向 **任意类型对象** 的指针可以隐式转换为 `void *`，反之亦然。

- 在 C++ 中，从 `void *` 转换到其他指针类型需要 **显式转换**。

示例：

```
int n = 1;
int* p = &n;
void* pv = p; // 从 int* 到 void* 的隐式转换
int* p2 = pv; // 从 void* 到 int* 的隐式转换 (C++ 中不允许)
```

`void *`: Common Use Cases

`void *` 类型为 C 的类型系统带来了灵活性，并在处理未知类型对象时被广泛使用。常见应用包括：

Dynamic Memory Allocation

`malloc` 返回 `void *`，使调用者可以将分配的内存解释为任何对象类型。

[Best Practice - *Stack Overflow*]
避免对 `malloc` 的返回值进行类型转换。应使用：

```
int* p = malloc(sizeof(int));
```

而不是：

```
int* p = (int *)malloc(sizeof(int));  // 不必要的类型转换
```

`void *`: Usage (cont.)

Generic Function Interfaces:

`qsort` 期望一个在 `const void*` 参数上运行的比较函数。

- 函数原型：

```
void qsort(void* base, size_t count, size_t size,
                       int (*comp)(const void*, const void*));
```

Null Pointers and Null Pointer Values

- 空指针指向 **无** 任何对象。
- 空指针与指向 **对象** 的任何指针比较均不相等。
- 它们通过 **空指针常量** 进行初始化或赋值。

Null Pointers Usage

- 表示 **对象不存在**。
- 用于指示 **错误情况**；例如，`malloc` 在分配失败时返回空指针。

```
int* ptr = malloc(sizeof(int));
if (ptr == NULL) { // 或 if (!ptr)
  printf("Allocation failed\n");
}
free(ptr);
```

通常，接受指针参数的函数应 **检查指针是否为空** 并做出相应处理。

Null Pointer Values

- *空指针值* 是指针类型的 **零值**。
- 每种指针类型都有一个特殊的空指针值。
- 根据 comparison operators 的定义，两个空指针值总是相等的。

Null Pointer Constants: Categories

- `NULL`:
  定义于许多标准头文件（例如 `<stddef.h>`、`<stdio.h>`、`<stdlib.h>`）。

```
int* ptr = NULL;
```

- **Integer Constant Zero:**

```
double* ptr_1 = 0;
double* ptr_2 = 2 - 2;
```

`NULL`, `nullptr`, and *Type Safety*

> 如果对数据的操作受到数据类型的限制，则该语言是类型安全的。

C 在 **有限的上下文** 中是类型安全的：

- 例如， `printf("%d", 12);` 使用 `%d` 来指示一个在 **运行时** 的整数。
- …

`NULL`, `nullptr`, and *Type Safety*

宏 `NULL` 是一个 **实现定义的** 空指针常量，其可能为：

- 值为 `0` 的整型常量表达式（例如 `2 - 2`），
- 值为 `0` 的整型常量表达式并转换为 `void *` （例如 `(void *)0`），
- 或者，[自 C23 起] 预定义常量 `nullptr`。

类型安全问题：

问题： `func(NULL)` 匹配哪个函数?

```
void func(int a){
    //...
}
```

```
void func(char* ptr){
    //...
}
```

`NULL`, `nullptr`, and *Type Safety* (cont.)

```
void func(int a){
    //...
}
```

```
void func(char* ptr){
    //...
}
```

- 请记住：`NULL` 是一个空指针 **常量**，而非指针本身。
- 类型转换（例如，`func((void *)0)` 或在 `ptr == NULL` 时调用 `func((int)ptr)`）可能令人困惑且容易出错。

`nullptr`: *Type-Safe* Null Pointer

关键字 `nullptr` 表示一个具有增强类型安全性的预定义空指针常量。

- 它的类型为 `nullptr_t`，与所有指针和整型均不同。
- 它可以隐式转换为任何指针类型或布尔类型（其值为 `false`）。
- 它 **不能** 隐式转换为整数，从而防止歧义和意外转换。

[Best Practice]
在现代 C++（自 C++11 起）和 C（自 C23 起）中，应使用 **`nullptr`** 以保证 **类型安全**、**代码清晰**，并避免隐式转换错误。

Subscript Operator `[ ]`

Syntax:

- 形式1：*指针表达式* `[` *整数表达式* `]`
- 形式2：*整数表达式* `[` *指针表达式* `]`

Definition:

`E1[E2]` 等价于 `*((E1) + (E2))`。

- 当 *指针表达式* 为数组时，会进行 **隐式数组到指针转换**，转换为指向首个元素的指针。
- 对于定义如下的数组：

  ```
  int a[3] = {1, 2, 3};
  ```

  表达式 `a[2]` 等价于 `2[a]`。

`const` Qualification

- `const` Semantics
- Low-Level `const` -ness of Pointers
- Top-Level `const` -ness of Pointers

`const` Semantics

`const` 限定符应用于具有标识性的 *左值* 表达式。这类表达式不可被修改或重新赋值。

```
const int n = 1; // n 被 const 限定
n = 2; // 错误: n 不能被修改

int x = 2; // x 是可变的
const int* p = &x;
*p = 3; // 错误: 无法修改 *p (只读)
```

用 `const` 声明的对象 ***可能会*** 被放置在 **只读内存** 中。如果从未取得 `const` 对象的地址，则该对象甚至可能不会被存储。

Low-Level `const`-ness of Pointers

- **指向 `const` 的指针（低级）：**
  `const` 限定符应用于所指向的对象。
  - 指向 `const` 的指针可以引用非 `const` 对象。
  - 允许从非 `const` 指针隐式转换为指向 `const` 的指针；但反向转换需要显式转换，这可能导致 **未定义行为**。

```
int n = 42;                 // n 的值为 42
int* p = &n;                // p 指向 n
const int* cp = p;          // cp 为指向 const int 的指针（安全）
p = cp;                     // 警告：丢弃 const 限定符
p = (int*)cp;               // 显式转换（风险较高）
*p = 50;                    // 未定义行为：修改 const 值
```

Top-Level `const`-ness of Pointers

- **常量指针（顶级）：**
  指针本身是常量，但所指向的对象仍然可修改。

```
int n = 42;
int m = 100;
int* const p = &n; // p 是一个指向 int 的常量指针

*p = 50;  // 可以：所指对象可被修改
p = &m;   // 错误：p 为常量指针，不能重新赋值
```

Arrays

- Variable-Length Array (VLA)
- `sizeof` Operator
- Array to Pointer Conversion

Variable-Length Arrays (VLA)

> 在 C 语言中，数组 `Type arr[N]` 中的大小 `N` 应为 **编译时常量**:
>
> ```
> int n[10];
> char str[] = "abc";
> ```

如果数组大小在 **运行时** 决定，则称为 **可变长度数组 (VLA)**:

```
int n;
scanf("%d", &n);
int a[n]; // VLA
```

VLA Characteristics

```c
int n;
scanf("%d", &n);
int a[n]; // VLA
```

- 可变长度数组的大小在每次到达其声明时均被重新计算。
- 该数组在 **栈上分配**，其生命周期在超出作用域时结束。
- 可变长度数组在 **C99** 中引入，但在 **C11** 中成为 **可选** 特性。

Limitations and Best Practices

- 可变长度数组尺寸不固定，对于大规模分配可能导致栈溢出。
- **移植性问题：** 部分编译器不支持可变长度数组。
- **C++ 限制：** 可变长度数组曾在 **C++14** 中提议，但最终 **被拒绝**。

Best Practice: Avoid VLAs

应改用 `malloc` 或 `calloc`：

```
int* a = malloc(n * sizeof(int));
if (!a) { /* 处理分配失败 */ }
```

更多细节请参见 Gkxx 的 CSDN 博客。

Array to Pointer Conversion

- 数组上的 `sizeof` 操作符
- 数组衰变为指针

`sizeof` Operator on Arrays

数组和指针是不同的类型，尽管数组常常会转换为指针。

- `sizeof` 返回一种类型的字节大小，其返回类型为 `size_t`，是一种无符号整数类型。
- **注意：** 对于可变长度数组，`sizeof` 在运行时计算。

示例：

```c
#include <stdio.h>
int main(void) {
  int a[] = {1, 2, 3, 4, 5};
  printf("Array: %zu\n", sizeof(a));    // 例如，20 字节
  printf("Pointer to array: %zu\n", sizeof(&a));    // 例如，8 字节
  printf("Pointer to element: %zu\n", sizeof(&a[0])); // 例如，8 字节
}
```

Decay from Arrays to Pointers

衰变 指的是数组隐式转换为指向其首个元素的指针。

- 类型为 `Type[N]` 的数组会衰变为类型为 `Type *` 的指针。
- 例如，`p = &arr[0]` 可以写作 `p = arr`。
- 同样，`*arr` 等价于 `arr[0]`。

Exceptions

这种转换 **不会** 在以下情况下发生：

- 作为取地址操作符 (`&`) 的操作数，
- 作为 `sizeof` 的操作数，
- 用作字符串字面量进行初始化。

C-Style Strings

- String I/O
- C Strings Library

String I/O

- **Error-Prone Input:**
  像 `scanf`（使用 `"%s"`）和 `gets`（在 C11 中已移除）等函数存在安全隐患。
- **Memory-Safe Input:**
  使用 `fgets` 进行边界检查的输入。
- **Output:**
  使用 `puts` 输出字符串。

Error-Prone Input: `scanf`, `gets`

- `scanf` / `printf`：使用 `"%s"`
  - 使用格式说明符 `"%s"` 的 `scanf` 函数存在内存安全问题，因为它只接受 `char *` 类型的指针，但数组的长度未被指定。

Error-Prone Input: `scanf`, `gets`

- `gets`：读取字符串时没有进行 **边界检查**.

```
char* gets( char* str );
```

  - 该函数自 C11 起已 **从标准中移除**。
  - 进行边界检查的 `gets` 替代函数为 `gets_s`，但并非所有编译器（包括 GCC）均支持。

    ```
    char* gets_s(char* str, rsize_t n);
    ```

    `gets_s` 函数最多写入 `n - 1` 个字符，以确保有足够空间容纳终止符。

Memory-Safe Input: `fgets`

Signature: (定义于头文件 `<stdio.h>`)

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

`fgets` 相较于其他输入函数更具可移植性、通用性及安全性，因为它包含边界检查：

- 它最多从指定流中读取 `count - 1` 个字符。
- 当遇到换行符（`\n`）或文件结束符（`EOF`）时停止读取。
- 成功时返回 `str` 的指针；失败时返回空指针。

```
char str[100];
fgets(str, sizeof(str), stdin);
```

Output: `puts`

Signature: (定义于头文件 `<stdio.h>`)

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

- `puts` 函数将 null 终止的字符串 `str` 中的每个字符写入输出流 `stdout`，并在末尾附加一个换行符 (`\n`)。
- `str` 中的终止 null 字符 (`\0`) 不会被写入输出。

Question:

为什么 `puts` 的参数声明为 `const`，而 `fgets` 的参数却不是?

C Strings Library

- Review of In-Class Implementation of `strlen`
- Implementation of `strcpy`
- Overview of C Strings Library

Review of In-Class Implementation of `strlen`

```
size_t my_strlen(const char* str) {
  size_t ans = 0;
  while (*str++ != '\0')   // 如何使用 while (*(++str) != '\0') ?
      ++ans;
  return ans;
}
```

- **Operator Precedence**：后缀自增运算符 `++` 的优先级高于解引用运算符 `*`。

Review of `strlen`

`strlen(s)` 函数每次调用时都会遍历整个字符串 `s` 。

```
for (size_t i = 0; i < strlen(s); ++i)
    // ...
```

在这个循环中，每次迭代结束时都会计算条件 `i < strlen(s)` 。由于 `strlen(s)` 每次调用时都需要完整遍历字符串，导致该循环性能极差，其时间复杂度为 $O\left(n^2\right)$ 。

Review of `strlen` (cont.)

为提高性能，应修改为：

```
for (size_t i = 0, n = strlen(s); i < n; ++i)
    // ...
```

通过将字符串长度计算一次并存储在变量 `n` 中，循环可以以线性时间 $O(n)$ 执行。

Review of `strlen` (cont.)

```
for (int i = 0; i < strlen(s); ++i)
    // ...
```

为什么编译器会对 `i < strlen(s)` 发出警告?

- `strlen` 的返回类型为 `size_t`，是一种无符号整数类型。
- 当 `int` 与 `size_t` 之间进行运算或比较时，`int` 值会被隐式转换为 `size_t`。
  - 因此，如果 `i` 为负（例如 `-1`），比较 `-1 < strlen(s)` 几乎总是被评估为 `false`。

*** 避免混合使用有符号与无符号整数。**

Implementation of `strcpy`

Signature: (定义于头文件 `<string.h>`)

```
char *strcpy(char* dest, const char* src);
```

- `strcpy` 函数将 `src` 所指向的 null 终止字节字符串（包括终止符 `\0`）复制到由 `dest` 指向的字符数组中。
- 若出现以下情况，其行为均为未定义：
  - `dest` 数组的大小不足以容纳 `src` 的内容。
  - 两个字符串存在重叠。
  - 或者 `dest` 不是指向字符数组的指针，或 `src` 不是指向 null 终止字节字符串的指针。

Exercise 2: Implement Your Own `strcpy` Function

```
char *strcpy(char* dest, const char* src);
```

Overview of C Strings Library

- Null-terminated byte strings (CppReference)

Dynamic Memory Management

- `malloc`, `calloc`
- `realloc`
- `free`
- In-Class Examples

`malloc`, `calloc`

```
void* malloc(size_t size);
void* calloc(size_t num, size_t each_size);
```

- `malloc`：分配 `size` 字节的 **未初始化内存**，即所分配内存中元素的值是 *不确定的*。
- `calloc`：为包含 `num` 个对象的数组分配内存，每个对象大小为 `each_size`，并将所有字节 **初始化为零**。
  - 由于 **对齐要求**，分配的字节数可能大于 `num * size`。
  - 注意：将所有字节初始化为零 **不保证** 浮点数值被设为 `0.0` 或指针被设为 **空指针值**。

`malloc`

Try the following code snippet

根据 C 标准，`malloc` 分配的内存包含 **不确定的值**，但在实践中，这些值通常显示为零。这只是巧合吗？（参见 Stack Overflow 的回答）

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

int main(void) {
  int n = 5;
  int *arr = malloc(n * sizeof(int));
  printf("Array values:\n");
  for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
  }
  printf("\n");
  free(arr);
  return 0;
}
```

- 分配的内存可能来自于先前释放的内存的 **回收** 或由操作系统直接提供。
- 出于安全考虑，操作系统通常返回 **零初始化的内存** 以避免暴露其他进程的数据。

`realloc`: reallocates a specified area of memory

```
void* realloc(void* ptr, size_t new_size);
```

参数 `ptr` 可以是 `NULL` 或者之前由 `malloc`、`calloc` 或 `realloc` 分配的指针，并且 ***尚未被释放***（通过调用 `free` 或 `realloc`）。

- 如果 `ptr` 为 `NULL`，其行为等同于 `malloc(new_size)`。
- 如果 `ptr` 不为 `NULL`，则分配过程为：
  - 扩展或收缩 `ptr` 指向的现有内存区域，或
  - 分配一个大小为 `new_size` 字节的新内存块，将较新或旧尺寸较小部分的内存数据复制过去，并释放旧块。

如果没有足够内存可用，则旧内存块不会被释放，并返回空指针。

`malloc`, `calloc`, and `realloc`

- 为防止内存泄漏，任何由这些函数返回的指针都必须使用 `free`（或 `realloc`）进行释放。需要注意的是，在重新分配后，原始指针 `ptr` 将失效，对其的后续访问会导致未定义行为，即使重新分配发生在原地。
- `malloc(0)`、`calloc(0, N)`、`calloc(N, 0)` 和 `realloc(ptr, 0)` 的行为是 **实现定义的**。
  - 这些函数可能不分配内存并返回空指针。
  - 或者，它们可能分配一定量的内存并返回指向该内存起始位置的指针。
    - **但是，解引用该指针会导致未定义行为。**
    - 为避免内存泄漏，该分配的内存 ***仍必须被释放***。

`free`: `void free(void* ptr)`

允许的操作：

- 释放空指针：无需检查 `ptr != NULL`。

未定义行为：

- 对一个已经被 `free` 释放的悬挂指针进行使用或释放。
- 释放一个其地址未由 `malloc`、`calloc`、`realloc` 或 C11 的 `aligned_alloc` 返回的指针。
    - 该指针必须指向分配内存块的 **起始位置**，而非中间位置。

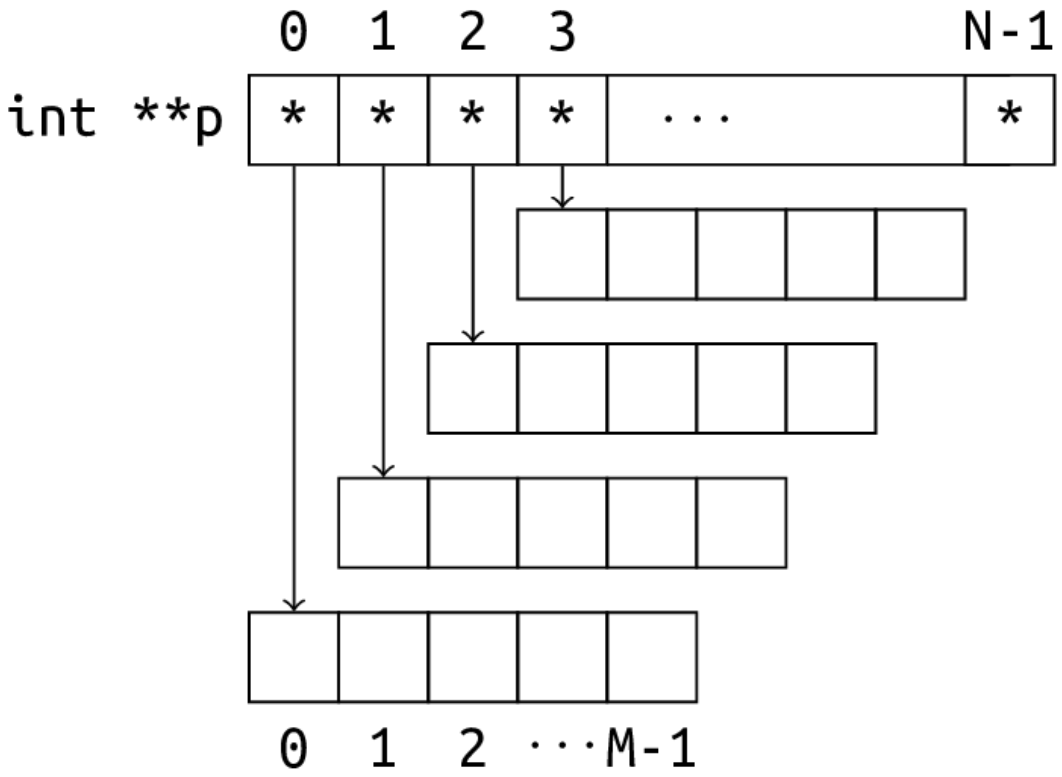
In-Class Examples

- Creating a Dynamic 2-Dimensional Array
- Reading a String of Unknown Length
  - `memcpy`
  - `ungetc`

Creating a Dynamic 2-Dimensional Array

```
// Method 1
int **p = malloc(sizeof(int *) * n);
for (int i = 0; i != n; ++i)
  p[i] = malloc(sizeof(int) * m);
for (int i = 0; i != n; ++i)
  for (int j = 0; j != m; ++j)
    p[i][j] = /* ... */
for (int i = 0; i != n; ++i)
  free(p[i]);
free(p);
```

```
// Method 2
int *p = malloc(sizeof(int) * n * m);
for (int i = 0; i != n; ++i)
  for (int j = 0; j != m; ++j)
    p[i * m + j] = /* ... */
// ...
free(p);
```



Reading a String of Unknown Length

假设我们需要读取一串**非空白**字符，且长度未知。在这种情况下，我们可以使用 `malloc` 动态分配内存，并根据需要使用 `realloc` 调整大小。步骤包括：

- 动态分配内存以存储字符串。
- 如果缓冲区满了，则重新分配一个更大的缓冲区，并将现有数据复制到新缓冲区。
- 完成后，释放分配的内存以避免内存泄漏。

Reading a String of Unknown Length (Snippet 1)

```c
char *read_string(void) {
  // 忽略前导空白字符
  char c = getchar();
  while (isspace(c))
    c = getchar();

  char *buffer = malloc(INITIAL_SIZE);
  int capacity = INITIAL_SIZE;
  int cur_pos = 0;
```

Reading a String of Unknown Length (Snippet 2)

```
while (!isspace(c)) {
  if (cur_pos == capacity - 1) {
    char *new_buffer = malloc(capacity * 2);

    // 将已存储的所有内容复制到新缓冲区
    memcpy(new_buffer, buffer, cur_pos);

    // 别忘了释放旧内存。
    free(buffer);

    capacity *= 2;
    buffer = new_buffer;
  }

  buffer[cur_pos++] = c;
  c = getchar();
}
```

Reading a String of Unknown Length (Snippet 3)

```
    // 此时，c 为空白字符，不属于所需内容。
    ungetc(c, stdin); // 将该空白字符放回输入流。
    buffer[cur_pos] = '\0'; // 记得添加终止符！！
    return buffer;
}
```

`memcpy`

定义于头文件 `<string.h>`：

```
void* memcpy(void* dest, const void* src, size_t count);
```

- **Function**: 将 `src` 所指内存位置的 `count` **个字符**复制到 `dest` 所指内存位置。该函数返回 `dest` 。
- **Undefined Behavior**:
  - 如果 `dest` 没有足够空间容纳 `count` 个字符。
  - 如果 `src` 与 `dest` 的内存区域 **重叠**。
  - 如果 `src` 或 `dest` 为无效或空指针。
- **`memcpy` 比 `strcpy` 更快**，后者在遇到 `\0` 时停止，而 `memcpy` 则严格按照指定字节数复制 **而不扫描数据**。

ungetc

```
int ungetc(int ch, FILE* stream);   // 定义于头文件 <stdio.h>
```

- `ungetc` 将字符 `ch` 推回到流 `stream` 的输入缓冲区中。下次从该流读取时将获得此被推回的字符。
  - **注意**：该操作不会修改与该流相关联的外部设备。
- **Buffer Limitations**:
  - 仅保证一次字符推回（缓冲区大小为 1）。如果在读取或重定位前多次调用 `ungetc`，其行为为 **实现定义**，且可能失败。
  - 如果进行了多次成功的 `ungetc` 调用，被推回的字符将以 **逆序** 读取。

CS100 习题课 4

来自 刘晓