Advanced Uses of Pointers (5)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

All materials are provided by C Programming: A Modern Approach by K. N. King, 2nd edition, 2008, W. W. Norton & Company.

Outline

- Other Uses of Function Pointers
- Restricted Pointers
- Flexible Array Members

Other Uses of Function Pointers

- Like we have mentioned, C treats pointers to functions **just like** pointers to data.
- They can be **stored in variables** or used as **elements of** an **array** or as **members** of a structure or union.
- Let’s see how we can **declare an array whose elements** are function pointers!

Other Uses of Function Pointers

- Suppose that we are writing a program that displays a **menu** of **commands** for users to choose from.
- We can first write functions that implement these commands (`_cmd`) and **store pointers** to **these functions** in an array `file_cmd`

```
void (*file_cmd[])(void) = {new_cmd,
                            open_cmd,
                            close_cmd,
                            close_all_cmd,
                            save_cmd,
                            save_as_cmd,
                            save_all_cmd,
                            print_cmd,
                            exit_cmd};
```

Other Uses of Function Pointers

- **A call of the function** stored in position **n** of the `file_cmd` array
- We could get a **similar** effect with a `switch` statement, but using an **array** of function pointers provides more **flexibility**.
- Because the elements of the array can be **changed** as the program is **running**.
  - For example, we can remove or add functions to the `file_cmd[]` array

```
void (*file_cmd[])(void) = {new_cmd, ...};
…
(*file_cmd[n])();  /* or file_cmd[n](); */
```

Restricted Pointers

- In C99, the keyword `restrict` may appear in the declaration of a **pointer**
- `p` is said to be a ***restricted pointer.***
- `restrict` is used for **optimization**
  - Most programmers won't use `restrict` unless they're **fine-tuning** a program to achieve the **best** possible **performance**.

```
int * restrict p;
```

Restricted Pointers

- The **intent** is that if `p` points to an **object** that is later modified, then that object is **not accessed** in **any way other than** through `p`.
- In other words, when we use **restrict** with a pointer `p`, it **tells** the **compiler** that `p` is the **only way to access** the **object** pointed by it, in other words, there's **no other pointer** pointing to the same object

```
int * restrict p;
```


https://www.geeksforgeeks.org/restrict-keyword-c/

Restricted Pointers

- Consider the following example.
- Normally it would be legal to **copy** `p` into `q` and then modify the integer through `q`
- Because `p` is a restricted pointer, the effect of executing the statement `*q = 0;` is undefined.
- We can only modify the int `obj` through `p`

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

q = p; // make q point to where p point to.
*q = 0; //causes undefined behaviors, since p is restrict pointer
```

https://www.geeksforgeeks.org/restrict-keyword-c/

Restricted Pointers

- To illustrate the use of `restrict`, consider the `memcpy` and `memmove` functions which belong to the `<string.h>` header.
- The prototype for `memcpy`, which **copies bytes** from one object (pointed to by `s2`) to another (pointed to by `s1`)
- The use of `restrict` with both `s1` and `s2` indicates that the objects to which they point shouldn't **overlap**.
  - `s1` and `s2` point to different memory spaces

```
void *memcpy(void * restrict s1, const void * restrict s2,
             size_t n);
```

Restricted Pointers

- In contrast, `restrict` doesn't appear in the prototype for `memmove`
- `memmove` is similar to `memcpy`, but is guaranteed to work even if the source and destination overlap.
- Example of using `memmove` to shift the elements of an array instrad of `memcpy`

```
void *memmove(void *s1, const void *s2, size_t n);
…
int a[100];
…
memmove(&a[0], &a[1], 99 * sizeof(int));
```

Restricted Pointers

- The prototypes for the two functions were nearly identical:
- The use of `restrict` in the C99 version of `memcpy`'s prototype indicates that the `s1` and `s2` objects should not overlap.

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2,size_t n);
```

Flexible Array Members

- In some applications, we'll need to define a structure that contains an array of an unknown size.
- For example, we might want a structure that stores the characters in a string together with the string's length.
- In this case, we can use the ***flexible array member*** (C99 feature)

```
struct vstring {
    int len;
    char chars[]; // flexible array member
};
```

Flexible Array Members

- The length of the array isn't determined until memory is allocated for a `vstring` structure
- For example, we want to allocate `n` character for `char chars[]` when allocating memory for the `vstring` structure pointed by `str`

```
struct vstring {
int len;
char chars[]; // flexible array member
};

struct vstring *str = malloc(sizeof(struct vstring) +n);
str->len = n;
```

Flexible Array Members

- `sizeof` ignores the `chars` member when computing the size of the structure.
- So, we need to allocate extra `n` bytes for `char chars[]`
  - The length of `chars` will become `n`
  - So the member `len` will also been assigned the value `n`

```
struct vstring {
int len;
char chars[]; // flexible array member
};

struct vstring *str = malloc(sizeof(struct vstring) +n);
str->len = n;
```

Flexible Array Members

- Special rules for structures that contain a flexible array member:
  - The flexible array must be the last member.
  - The structure must have at least one other member.
- Copying a structure that contains a flexible array member will copy the other members but not the flexible array itself.

Flexible Array Members

- A structure that contains a flexible array member is an ***incomplete type.***
- An incomplete type is missing part of the information needed to determine how much memory it requires.
- Incomplete types are subject to various restrictions.
- In particular, an incomplete type can’t be a member of another structure or an element of an array.
- However, an array may contain **pointers** to **structures** that have a **flexible** array member.

Summary

- Other Uses of Function Pointers
  - array whose elements are function pointers
- Restricted Pointers
  - example of `memcpy` and `memmove`
- Flexible Array Members
  - introduction and restrictions