Advanced Uses of Pointers (2)

Program Design (II)

2022 Spring

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

Updated Schedule (March. 24)

7	3/29		Advanced Uses of Pointers	Ch 17.1 - 17.2
	3/31	*Holiday*	Spring Break	
8	4/5	*Holiday*		
	4/7	**Homework 2**	Advanced Uses of Pointers	Ch 17.3 - 17.5
9	4/12		Advanced Uses of Pointers	Ch 17.5 - 17.6
	4/14		Advanced Uses of Pointers	Ch 17.7 - 17.9
10	4/19		Midterm Exam *Paper exam*	
	4/21	Homework 3	Declarations & Final Project Introduction *Announce Final Project*	Ch 18.1 - 18.3

Updated Schedule: Quiz Section (March. 24)

The Domjudge of Ex4 will be closed on 3/31.

6	3/22		Structures, Unions, and Enumerations	Practice of Structures, Unions, and Enumerations
	3/24		Structures, Unions, and Enumerations	
7	3/29		Advanced Uses of Pointers	**3/29 Demo Section** **3/31 No Quiz Section**
	3/31	*Holiday*		
8	4/5		Advanced Uses of Pointers	**4/5 No Quiz Section** **4/7 Demo Section**
	4/7	Homework 2	Advanced Uses of Pointers	
9	4/12		Advanced Uses of Pointers	**No Quiz Section**
	4/14		Midterm Exam	

Spring Break

Quick Recap of Advanced Uses of Pointers (1)

- Dynamic storage allocation
  - `stdlib.h`
  - `malloc`
- Null Pointers
- Using `malloc` to Allocate Storage for an Array
- Deallocating Storage
  - `free()`
- pointer to pointer (double pointer)
  - `int **ptr2`

Let’s do some tests to see if you have some basic understandings of these topics before we continue!

slido

What is the correct content for the gap (1) and (2) if I want to allocate a memory block of 100 bytes which is pointed by pointer p?

ⓘ Start presenting to display the poll results on this slide.

slido

What are the correct content for the gap (3) if I want to check if the allocation is failed?

ⓘ Start presenting to display the poll results on this slide.

slido

What is the correct expression for the gap (1) to allocate memory space enough for n numbers of struct person

ⓘ Start presenting to display the poll results on this slide.

slido

Now we have n numbers of struct person. What is the correct expression to let us assign value to the member name and age for each struct person (gap (1))?

ⓘ Start presenting to display the poll results on this slide.

If you don’t know how to answer the quiz at all…

- Check the slides and video of Advanced Uses of Pointers (1) on eCourse2
- Extra reading and watching to help you recap these topics
  - Dynamic Memory Allocation - CS50 Shorts: https://www.youtube.com/watch?v=xa4ugmMDhiE

Outline

- Dynamic storage allocation
- Dynamically allocated strings
- Dynamically Allocated Arrays
  - `calloc` and `realloc`

Dynamic storage allocation

- Dynamic storage allocation is done by calling a memory allocation function.
- The `<stdlib.h>` header declares three memory allocation functions
  - the unit of memory block is **byte**

```
#include <stdio.h>
#include <stdlib.h>
…
// malloc: allocates a block of memory but doesn’t initialize it
// calloc: Allocates a block of memory and clears it.
// realloc: Resizes a previously allocated block of memory.
```

Dynamic storage allocation

- These functions return a value of type `void *` (a "generic" pointer).
- We can now access the allocated memory block by the pointer `p`
- if these functions can't locate enough memory, they will return a null pointer (`NULL`)

```
#include <stdio.h>
#include <stdlib.h>
…
void *p;
p = malloc(1000);
if (p == NULL) {
      /* allocation failed; take appropriate action */
}
```

Dynamically Allocated Strings

- Dynamic storage allocation is often **useful** for working with **strings**.
- Strings are stored in **character** arrays
  - it can be hard to anticipate how long these arrays (a string) need to be.
  - For example, ask users to input their name (the longest personal name have more than 600 letters!)
- By **allocating strings dynamically,** our program can be more flexible when using strings.
  - because we can decide the length of strings after users make their inputs

Using `malloc` to Allocate Memory for a String

- A call of `malloc` that allocates memory for a string of `n` characters
- Each character requires one byte of memory; adding 1 to `n` leaves room for the null character.
- The generic pointer that malloc returns will be **converted** to `char *` when the assignment is performed

```
#include <stdio.h>
#include <stdlib.h>
…
char *p;
p = malloc(n+1);
```

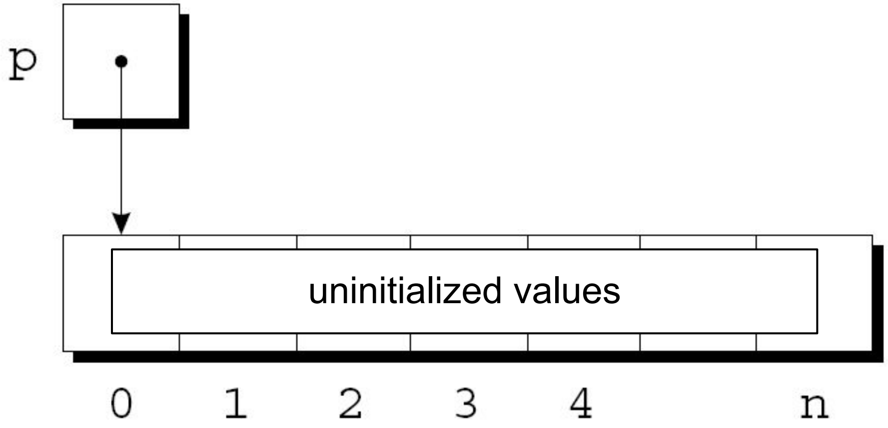
Using `malloc` to Allocate Memory for a String

- Some programmers prefer to **cast** `malloc`'s return value, although the cast is not required

```
#include <stdio.h>
#include <stdlib.h>
…
char *p;
p = (char *) malloc(n+1);
```

Using `malloc` to Allocate Memory for a String

- `malloc` will not clear or initalize the allocated memory, so `p` will point to an uninitialized array of `n` + 1 characters:

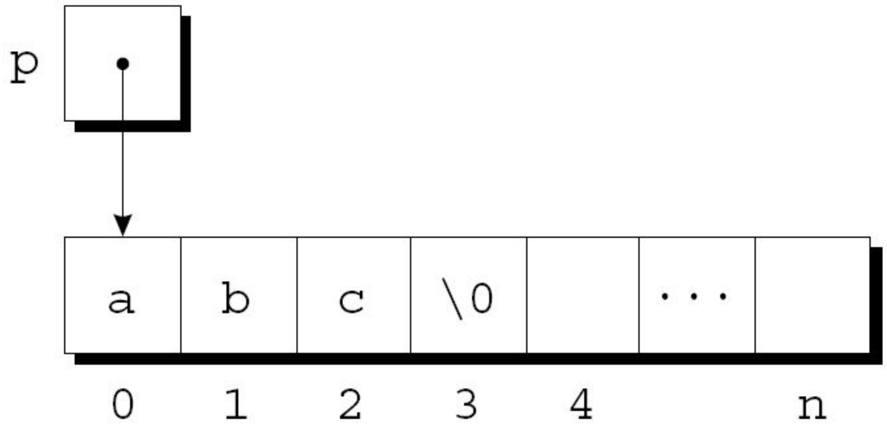


Using `malloc` to Allocate Memory for a String

- Calling `strcpy` is one way to initialize this array:

```
strcpy(p, "abc");
```

- The first four characters in the array will now be `a`, `b`, `c`, and `\0`:



Using Dynamic Storage Allocation in String Functions

- Dynamic storage allocation makes it possible to **write functions** that **return** a **pointer** to a “new” string.
- Consider the problem of writing a function that concatenates two strings **without changing either one.**
  - `strcat` modifies the first string passed to it

```
char *strcat(char *s1, const char *s2); //function prototype

strcpy(str1, "abc");
strcat(str1, "def"); /* str1 now contains "abcdef" */
```

Using Dynamic Storage Allocation in String Functions

- We declare the function `concat` with parameters `char *s1` and `s2`
- Since we need to return a pointer to a "new" string, the return type is `char *`
- Then, we create a string pointer `result` that wil be returned by `concat`

```
char *concat(const char *s1, const char *s2){
  char *result;

  result = malloc(strlen(s1) + strlen(s2) +
1);
  if (result == NULL) {
    printf("Error: malloc failed in
concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

Using Dynamic Storage Allocation in String Functions

- The function will measure the **lengths** of the two strings to be concatenated
  - `strlen()`
- then call `malloc` to allocate the right amount of space for the result.
  - check if the allocation success or not
- string pointer `result` now pointed to the new allocated space

```
char *concat(const char *s1, const char *s2){
  char *result;

  result = malloc(strlen(s1) + strlen(s2) +
1);
  if (result == NULL) {
    printf("Error: malloc failed in
concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

Using Dynamic Storage Allocation in String Functions

- Next, use `strcpy` to copy the first string into the new space pointed by `result`
- Then, call `strcat` to concatenate the second string to result
- Finally, return `result`

```
char *concat(const char *s1, const char *s2){
  char *result;

  result = malloc(strlen(s1) + strlen(s2) +
1);
  if (result == NULL) {
    printf("Error: malloc failed in
concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

Using Dynamic Storage Allocation in String Functions

- We can use this function `concat` by the following function call
- After the call, `p` will point to the string `"abcdef"`, which is stored in a dynamically allocated array.

```
char *p;
p = concat("abc", "def");
printf("%s", p);
```

```
abcdef
```

Using Dynamic Storage Allocation in String Functions

- **However**, functions such as `concat` that dynamically allocate storage must be used with care.
- When the string that `concat` returns is **no longer needed**, we'll want to **call** the `free` function to **release** the **space** that the string occupies.
- If we don't, the program may eventually **run out of memory**.

```
char *p;
p = concat("abc", "def");
free(p); It's important to free the space pointed by p  before make p point to other space
p = concat("123", "456");
printf("%s", p);
```

Let’s Take A Break!

Dynamically Allocated Arrays

- Dynamically allocated arrays have the same advantages as dynamically allocated strings.
- Although `malloc` can allocate space for an array, the `calloc` function is sometimes used instead, since `calloc` **initializes** the **memory** that it allocates.
- The `realloc` function allows us to make an array "**grow**" or "**shrink**" as needed.

```
#include <stdio.h>
#include <stdlib.h>
…
// malloc: allocates a block of memory but doesn't initialize it
// calloc: Allocates a block of memory and clears it.
// realloc: Resizes a previously allocated block of memory.
```

Using `malloc` to Allocate Storage for an Array

- Suppose a program needs an array of `n` integers, where `n` is computed during program execution.
- Once the value of `n` is known, call `malloc` to allocate space for the array
- we can assign a `void *` value to a variable of any pointer type

```
int *a; // declare a pointer variable
int n;
… // n is known
a = malloc(n * sizeof(int));
```

Using `malloc` to Allocate Storage for an Array

- Always use the `sizeof` operator to calculate the amount of space required for each element.
    - `sizeof` operator: return the numbers of byte of the given type; `sizeof(char)` is 1
- because we want to allocate an integer array, so the size of each element is `sizeof(inte)`

```
int *a; // declare a pointer variable
int n;
… // n is known
a = malloc(n * sizeof(int));
```

Using `malloc` to Allocate Storage for an Array

- We can now ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers in C.
  - use pointer as array name
- For example, we could use the following loop to initialize the array that `a` points to:

```
int *a, n;
…
a = malloc(n * sizeof(int));
for (i = 0; i < n; i++)
    a[i] = 0;
```

The `calloc` Function

- The `calloc` function is an alternative to `malloc`.
- Properties of `calloc`:
  - Allocates space for an array with `nmemb` **elements**, **each** of which is `size` bytes **long**.
  - Returns a **null pointer** if the requested space **isn't** available.
  - **Initializes** allocated memory by setting all bits to **0**.

```
void *calloc(size_t nmemb, size_t size); //prototype
```

The `calloc` Function

- A call of `calloc` that allocates space for an array of `n` integers
- By calling `calloc` with 1 as its first argument, we can allocate space for a data item of any type.
  - `p` is a pointer variable pointing to the object with data type of `struct point`

```
int *a, n;
…
a = calloc(n, sizeof(int));

struct point { int x, y; } *p;
p = calloc(1, sizeof(struct point));
```

The `realloc` Function

- The `realloc` function can resize a dynamically allocated array.
- `ptr` must **point** to a **memory** block **obtained by** a previous call of `malloc`, `calloc`, or `realloc`.
- `size` represents the **new size** of the block, which may be larger or smaller than the original size.

```
void *realloc(void *ptr, size_t size); //prototype
```

The `realloc` Function

- For example, we can reallocate the memory block pointed by `s`, which we obtained using `malloc`
- we use the other pointer `s_reall` to get the returned value (`void *`) of realloc
- From the output, we can see that the `realloc` did expand the original memory block
  - The address is the same

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

int main()
{
    char *s, *s_r;

    //allocate memory
    s = (char *) malloc(3+1);
    strcpy(s, "abc");
    printf("String = %s,  Address = %u\n", s, s);

    s_r = (char *) realloc(s, 5+1);
    strcat(s_r, "de");
    printf("String = %s,  Address = %u\n", s_r, s_r);
    printf("String = %s,  Address = %u\n", s, s);

    return 0;
}
```

```
String = abc,   Address = 3435868832
String = abcde,   Address = 3435868832
String = abcde,   Address = 3435868832
```

https://www.tutorialspoint.com/c_standard_library/c_function_realloc.htm

The `realloc` Function

- Properties of `realloc`:
  - When it **expands** a memory block, `realloc` **doesn't initialize** the bytes that are added to the block.
  - If `realloc` **can't enlarge** the memory block as requested, it returns a **null pointer**; the data in the old memory block is **unchanged**.

```
…
s_reall = (char *) realloc(s, 5+1);
if(!s_reall){ //check if returned NULL pointer
    //reallocation fail
}
```

The `realloc` Function

- Properties of `realloc`:
  - If `realloc` is called with a **null pointer** as its first argument, it behaves **like** `malloc`.
  - If `realloc` is called with 0 as its second argument, it **frees** the memory block.

```
char *s;

s = (char *) realloc(NULL, 5+1);
printf("String = %s,  Address = %u\n", s, s);
strcpy(s, "abc");

s = (char *) realloc(s, 0);
printf("String = %s,  Address = %u\n", s, s);
```

```
String = abc,  Address = 3141403296
String = (null),  Address = 0
```

The `realloc` Function

- We expect `realloc` to be reasonably efficient:
  - When asked to reduce the size of a memory block, `realloc` should **shrink** the original block
  - `realloc` should always attempt to **expand** a memory block **without** moving it.
- We can see from the previous example, the modified memory block has the same address with the original one

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

int main()
{
    char *s, *s_r;

    //allocate memory
    s = (char *) malloc(3+1);
    strcpy(s, "abc");
    printf("String = %s,  Address = %u\n", s, s);

    s_r = (char *) realloc(s, 5+1);
    strcat(s_r, "de");
    printf("String = %s,  Address = %u\n", s_r, s_r);
    printf("String = %s,  Address = %u\n", s, s);

    return 0;
}
```

```
String = abc,  Address = 3435868832
String = abcde,  Address = 3435868832
String = abcde,  Address = 3435868832
```

https://www.tutorialspoint.com/c_standard_library/c_function_realloc.htm

The `realloc` Function

- **If it can't enlarge a block**, `realloc` will **allocate** a **new block elsewhere**, then **copy** the **contents** of the old block **into the new one.**
  - the returned pointer by `realloc` will point to different address
- Once `realloc` has returned, be sure to **update all pointers** to the memory block in case it has been moved.
  - make all pointers pointed to original memory block point to the **new** memory block returned by `realloc`

Summary

- Dynamic storage allocation
- Dynamically allocated strings
  - how to create `char *concat(const char *s1, const char *s2)`
- Dynamically Allocated Arrays
  - Properties of `calloc` and `realloc`