Pointers (5)

Program Design (I)

2021 Fall

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

Outline

- Introduction of how to take the final exam
- dynamic storage allocation
- pointer to pointer (double pointer)

Dynamic storage allocation

- C’s data structures, including arrays, are normally fixed in size.
- Fixed-size data structures can be a problem, since we’re forced to choose their sizes when writing a program.
- However, sometimes it’s hard to know the size of an array before the program is executed

Dynamic storage allocation

- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.
- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.
- We will focus on dynamic storage allocation for **array** in this lesson.

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.
```

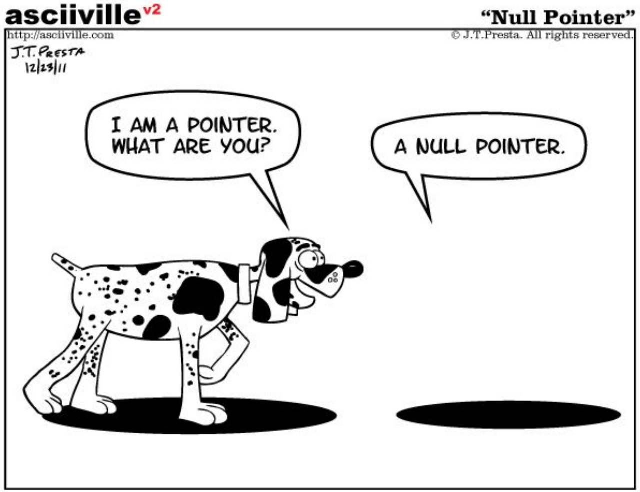
Dynamic storage allocation

- These functions return a value of type `void *` (a “generic” pointer).
- Just a pointer points to a memory address
- We can now access the allocated memory block by the pointer `p`

```
#include <stdio.h>
#include <stdlib.h>
…
void *p;
p = malloc(1000);
```

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer.***
- A null pointer is a "point to nothing (null)"
- After we've stored the `malloc` function's return value in a pointer variable `p`, we must test to see if it's a null pointer.
  - to test if the allocation is successful



https://www.freecodecamp.org/news/a-quick-and-thorough-guide-to-null-what-it-is-and-how-you-should-use-it-d170cea62840/

Null Pointers

- An example of testing `malloc`'s return value:
- `NULL` is a macro (defined in various library headers including `stdlib.h`) that represents the null pointer.

```
#include <stdio.h>
#include <stdlib.h>
…
p = malloc(10000);
if (p == NULL) {
      /* allocation failed; take appropriate action */
}
```

Dynamically Allocated Arrays

- It’s often difficult to estimate the proper size for an array
- more convenient to wait until the program is run to decide how large the array should be
- C allows a program to allocate space for an array during execution, then access the array through a pointer to its first element

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;
```

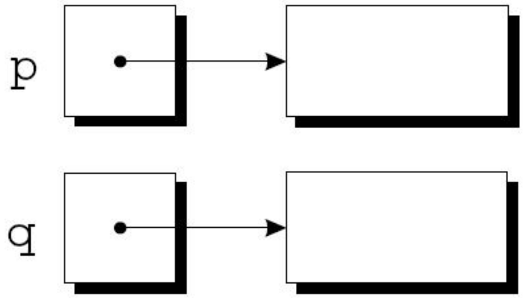
Deallocating Storage

- Calling these functions too often—or asking them for large blocks of memory—can exhaust the storage pool of computer memory (*heap*), causing the functions to return a null pointer.
    - fail to allocate memory
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.

Deallocating Storage

- pointer `p` and `q` point to memory allocated `by malloc()` respectively
- A snapshot after these two statements have been executed

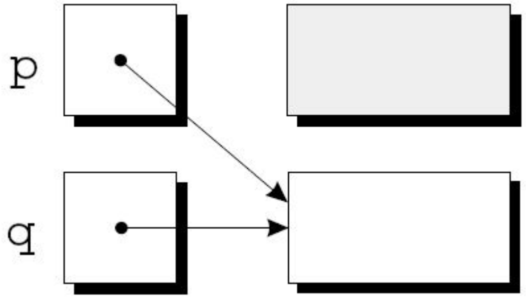
```
p = malloc(…);
q = malloc(…);
```



Deallocating Storage

- After `q` is assigned to `p`, both variables now point to the second memory block:
- There are no pointers to the first block, so we'll never be able to use it again.

```
p = malloc(…);
q = malloc(…);
p = q;
```



Deallocating Storage

- A block of memory that's no longer accessible to a program is said to be ***garbage.***
- A program that leaves garbage behind has a ***memory leak.***
- C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

The `free` Function

- Prototype for `free`: `void free(void *ptr);`
- `free` will be passed a pointer to an unneeded memory block
- Calling `free` releases the block of memory that `p` points to.

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

```c
int main()
{
    int *array_num; // {1, 2, 3, 4,}
    int n = 4;

    array_num = malloc(n * sizeof(*array_num));
    printf("sizeof = %d \n", (int) sizeof(*array_num));

    [redacted]

    free(array_num);

    return 0;
}
```

slido

sizeof = ?

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

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

void print_arr(int *a, int n){
    int *p;
    for(p = a; p < a + n; p++){
        printf("%d ", *p);
    }
}
```

```c
int main()
{
    int *array_num; // {1, 2, 3, 4,}
    int n = 4;

    array_num = malloc(n * sizeof(*array_num));
    printf("sizeof = %d \n", (int) sizeof(*array_num));

    array_num[0] = 1;
    array_num[1] = 2;
    array_num[2] = 3;
    array_num[3] = 4;

    print_arr(array_num, n);

    free(array_num);

    return 0;
}
```

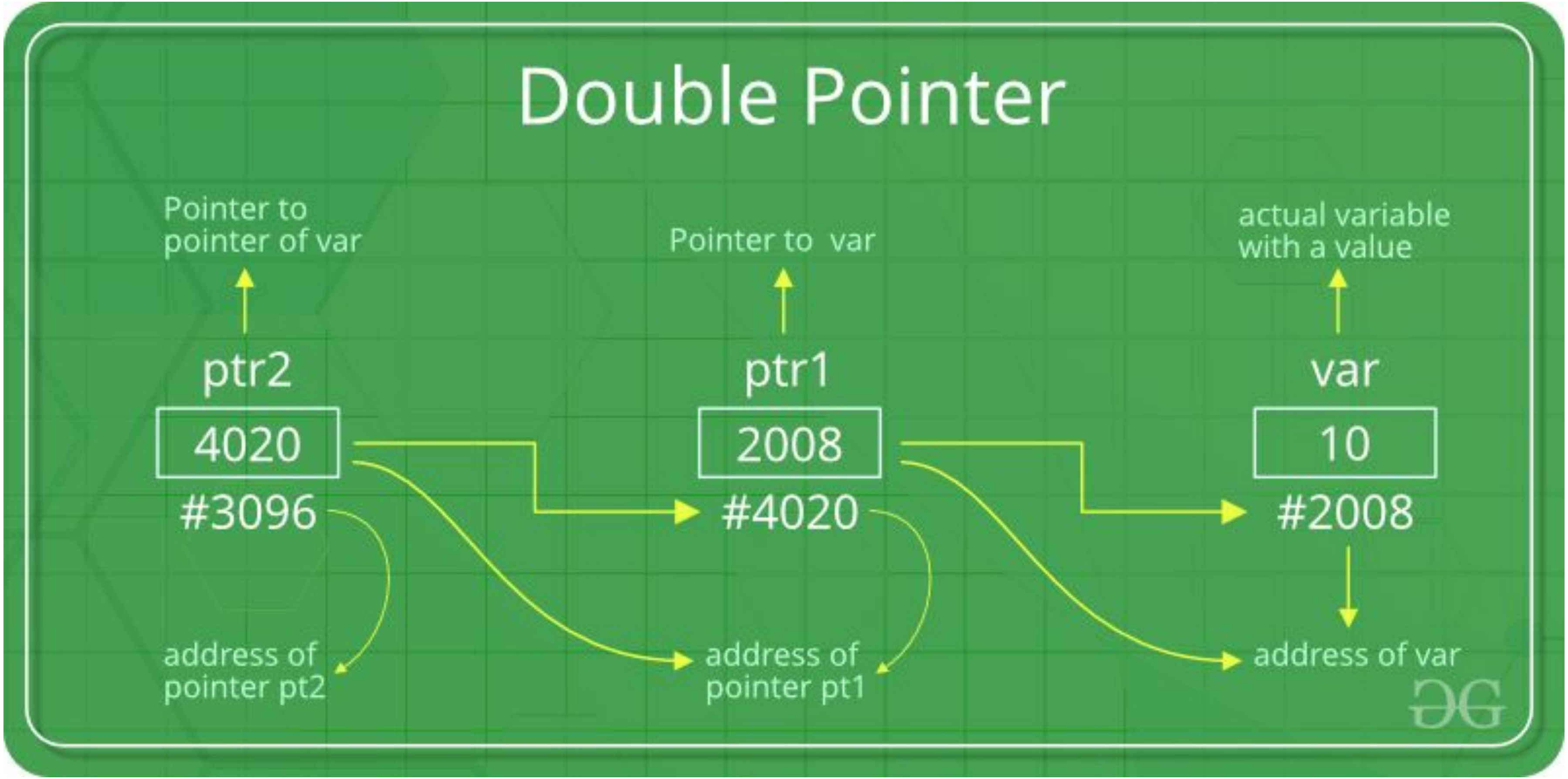
```
sizeof = 4
1 2 3 4
```

Let’s Take a Break!

Double Pointer

- The first pointer is used to store the address of the variable.
- And the second pointer is used to store the address of the first pointer.
- also know as **double** pointers

https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/



The first pointer ptr1 stores the address of the variable and the second pointer ptr2 stores the address of the first pointer.

https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/

Double Pointer

- Declare a double pointer is similar to declaring regular pointer

```
int **ptr2;
```

points to obj with the type of pointer variable pointing to an integer obj

a pointer variable

https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/

Double Pointer

```
int var = 10;
int *ptr2; // pointer for var
int **ptr1; // double pointer for ptr2

ptr2 = &var; // storing address of var in ptr2
ptr1 = &ptr2; // Storing address of ptr2 in ptr1

printf("%d\n", var ); // 10
```


https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/

Double Pointer

```
int var = 10;
int *ptr2; // pointer for var
int **ptr1; // double pointer for ptr2

ptr2 = &var; // storing address of var in ptr2
ptr1 = &ptr2; // Storing address of ptr2 in ptr1

printf("%d\n", var ); // 10
printf("%d\n", *ptr2 ); //10
```


https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/

Double Pointer

```
int var = 10;
int *ptr2; // pointer for var
int **ptr1; // double pointer for ptr2

ptr2 = &var; // storing address of var in ptr2
ptr1 = &ptr2; // Storing address of ptr2 in ptr1

printf("%d\n", var ); // 10
printf("%d\n", *ptr2 ); //10
printf("%d\n", **ptr1); //10
```


https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/

```c
int main()
{
    int *array_num; // {1, 2, 3, 4,}
    int n = 4;

    array_num = malloc(n * sizeof(*array_num));
    array_num[0] = 1;
    array_num[1] = 2;
    array_num[2] = 3;
    array_num[3] = 4;

    printf("array_num: \n");
    print_arr(array_num, n);
```

```c
    int **matrix_num; // { {1, 2, 3, 4}, {5, 6, 7, 8} }
    int m = 2;

    matrix_num = malloc(m * sizeof(*matrix_num));
    matrix_num[0] = array_num;
    matrix_num[1] = (int []){5, 6, 7, 8};

    //*(matrix_num + 0) = array_num;
    //*(matrix_num + 1) = (int []){5, 6, 7, 8};


    printf("marix_num: \n");
    print_max(matrix_num, n, m);

    free(array_num);
    free(matrix_num);
    return 0;
}
```

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

void print_arr(int *a, int n){
    int *p;
    for(p = a; p < a + n; p++){
        printf("%d ", *p);
    }
    printf("\n");
}

void print_max(int **a, int n, int m){
    int **p;
    for(p = a; p < a + m; p++){
        print_arr(*p, n);
    }
}
```

```c
int **matrix_num; // { {1, 2, 3, 4}, {5, 6, 7, 8} }
int m = 2;

matrix_num = malloc(m * sizeof(*matrix_num));
matrix_num[0] = array_num;
matrix_num[1] = (int []){5, 6, 7, 8};

//*(matrix_num + 0) = array_num;
//*(matrix_num + 1) = (int []){5, 6, 7, 8};


    printf("marix_num: \n");
    print_max(matrix_num, n, m);

    free(array_num);
    free(matrix_num);
    return 0;
}
```

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

void print_arr(int *a, int n){
    int *p;
    for(p = a; p < a + n; p++){
        printf("%d ", *p);
    }
    printf("\n");
}

void print_max(int **a, int n, int m){
    int **p;
    for(p = a; p < a + m; p++){
        print_arr(*p, n);
    }
}
```

```c
int **matrix_num; // { {1, 2, 3, 4}, {5, 6, 7, 8} }
int m = 2;

matrix_num = malloc(m * sizeof(*matrix_num));
matrix_num[0] = array_num;
matrix_num[1] = (int []){5, 6, 7, 8};

//*(matrix_num + 0) = array_num;
//*(matrix_num + 1) = (int []){5, 6, 7, 8};


    printf("marix_num: \n");
    print_max(matrix_num, n, m);

    free(array_num);
    free(matrix_num);
    return 0;
}
```

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

void print_arr(int *a, int n){
    int *p;
    for(p = a; p < a + n; p++){
        printf("%d ", *p);
    }
    printf("\n");
}

void print_max(int **a, int n, int m){
    int **p;
    for(p = a; p < a + m; p++){
        print_arr(*p, n);
    }
}
```

```c
int main()
{
    int *array_num; // {1, 2, 3, 4,}
    int n = 4;

    array_num = malloc(n * sizeof(*array_num));
    array_num[0] = 1;
    array_num[1] = 2;
    array_num[2] = 3;
    array_num[3] = 4;

    printf("array_num: \n");
    print_arr(array_num, n);

    int **matrix_num; // { {1, 2, 3, 4}, {5, 6, 7, 8} }
    int m = 2;

    matrix_num = malloc(m * sizeof(*matrix_num));
    matrix_num[0] = array_num;
    matrix_num[1] = (int []){5, 6, 7, 8};

    //*(matrix_num + 0) = array_num;
    //*(matrix_num + 1) = (int []){5, 6, 7, 8};


    printf("marix_num: \n");
    print_max(matrix_num, n, m);

    free(array_num);
    free(matrix_num);
    return 0;
}
```

```
array_num:
1 2 3 4
marix_num:
1 2 3 4
5 6 7 8
```

Extra Reading

- https://www.geeksforgeeks.org/pointer-array-array-pointer/
- https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/
- https://stackoverflow.com/questions/5580761/why-use-double-indirection-or-why-use-pointers-to-pointers/25110045
    - This one is really useful to understand pointer to pointer!

Final Exam Next Week

- **No class** next Tuseday (1/4) and Thursday (1/6)
- We will use the Quiz section to do the final exam
  - 2022/1/3 (Mon.) 7:00 - 10:00 pm
  - 2022/1/5 (Wed.) 7:00 - 10:00 pm
- We will have the **final lesson** of this course at 1/11!
- We will do a short recap of what you have done in this course and will invite TAs to review the course with us
- Hope to see you then!