Advanced Uses of Pointers (4)

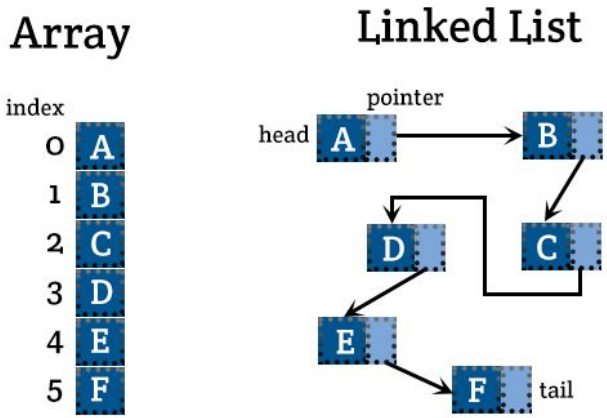
Program Design (II)

2022 Spring

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

Quick Recap

- Introduce Linked Lists
  - Declaring a Node Type
  - creating a node
- Common Operations of Linked Lists
  - inserting a node at the beginning of a (linked) list
  - searching for a node
  - deleting a node

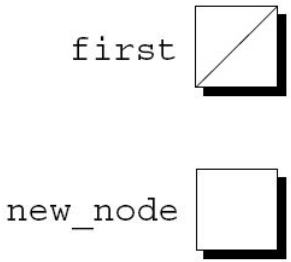


ref. Open4Tech

Mistake in last week (p18 of the slides)

Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```

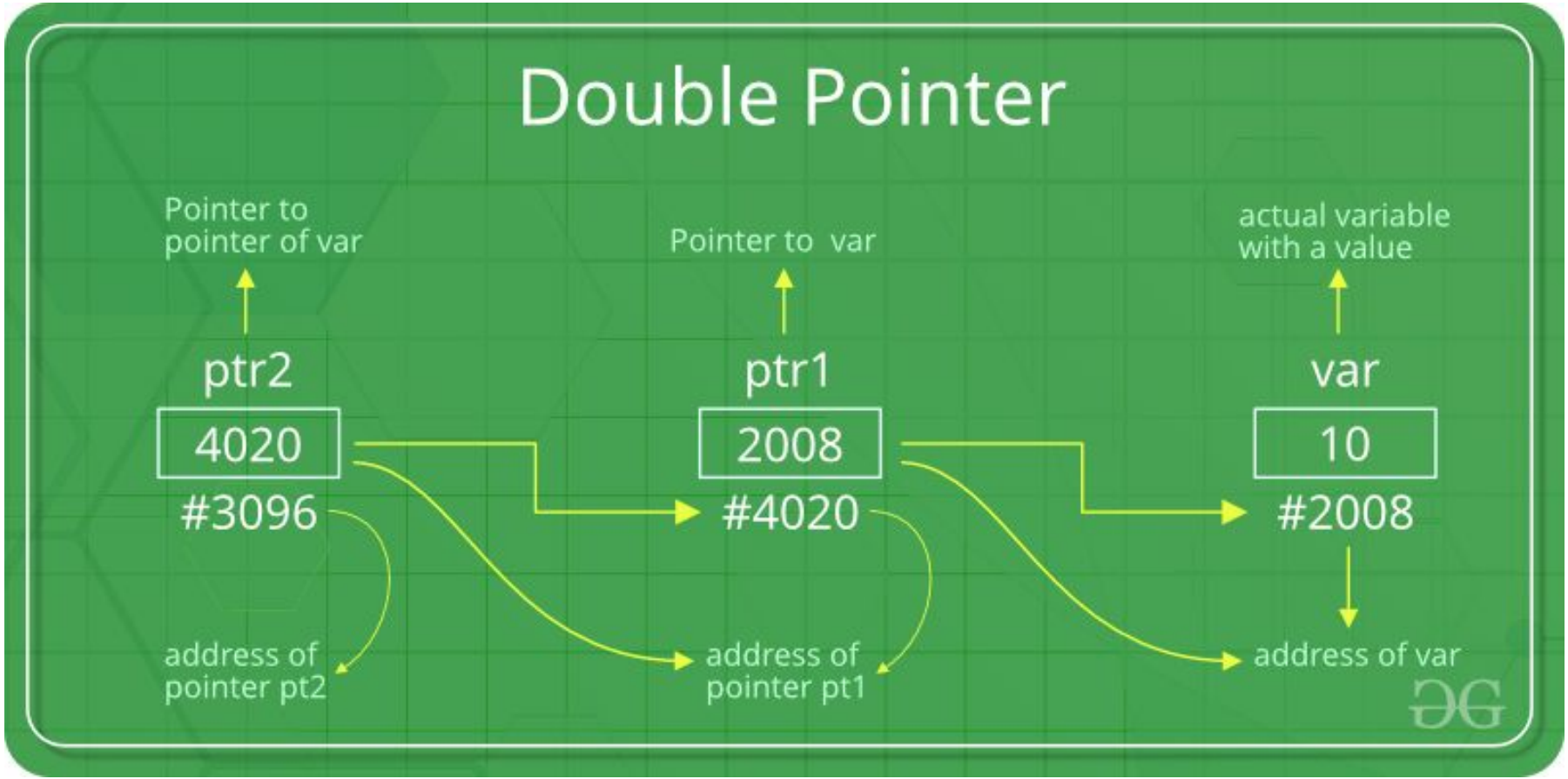


Outline

- Pointers to Pointers
- Pointers to Functions

Pointer to Pointer (Double Pointer)

- Quick Review of the topic double pointer in Advanced Uses of Pointers (1)
- 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



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/

Pointers to Pointers

- The concept of “pointers to pointers” also pops up frequently in the context of linked data structures.
- Let’s see an example!
- The `add_to_list` function for `strct node`

```
struct node {
      int value;            /* data stored in the node  */
      struct node *next;    /* pointer to the next node */
};
```

Pointers to Pointers

- The `add_to_list` function is passed a pointer `list` to the first node in a list
- it returns a pointer to the first node in the updated list

```
struct node *add_to_list(struct node *list, int n){
      struct node *new_node;
      new_node = malloc(sizeof(struct node));
      if (new_node == NULL) {
        printf("Error: malloc failed in add_to_list\n");
        exit();
      }
      new_node->value = n;
      new_node->next = list;
      return new_node;
}
```

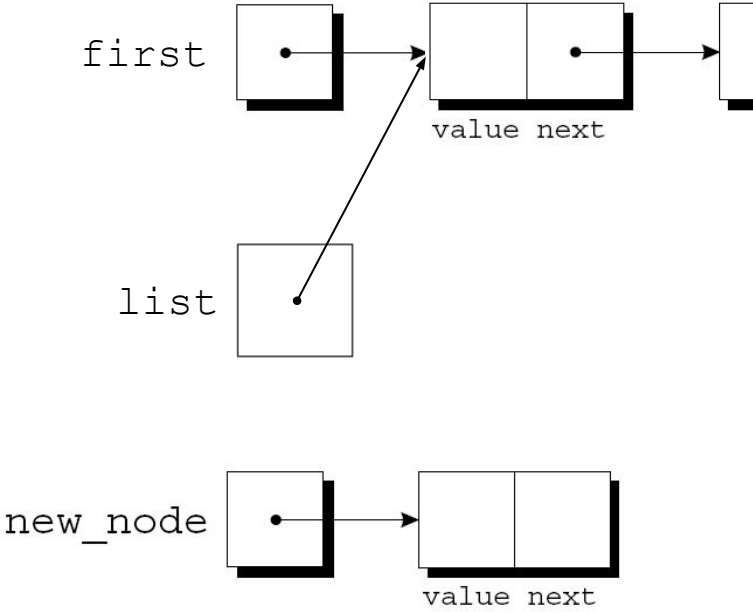
Pointers to Pointers

- When we call `add_to_list`, we'll need to store its return value into `first`

```
…
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

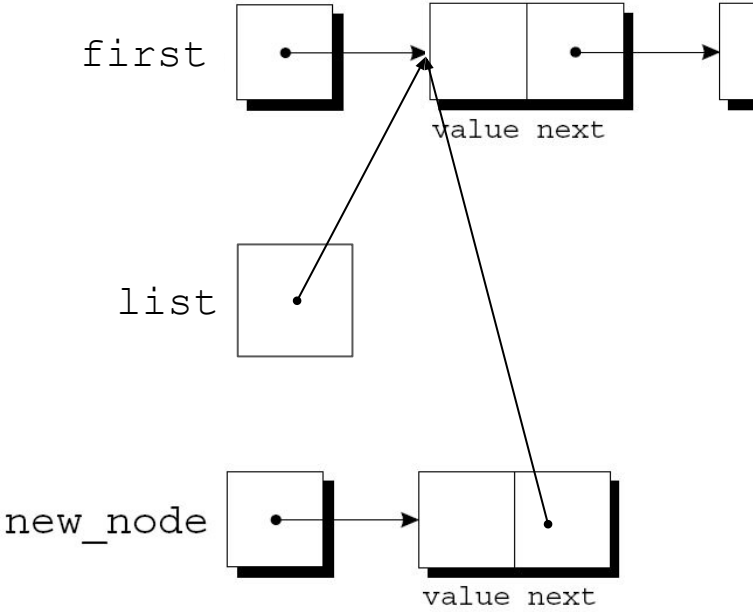
Pointers to Pointers

```
struct node *add_to_list(struct node *list,
int n){
      struct node *new_node;
      …
      new_node->value = n;
      new_node->next = list;
      return new_node;
}
…
first = add_to_list(first, 10);
```



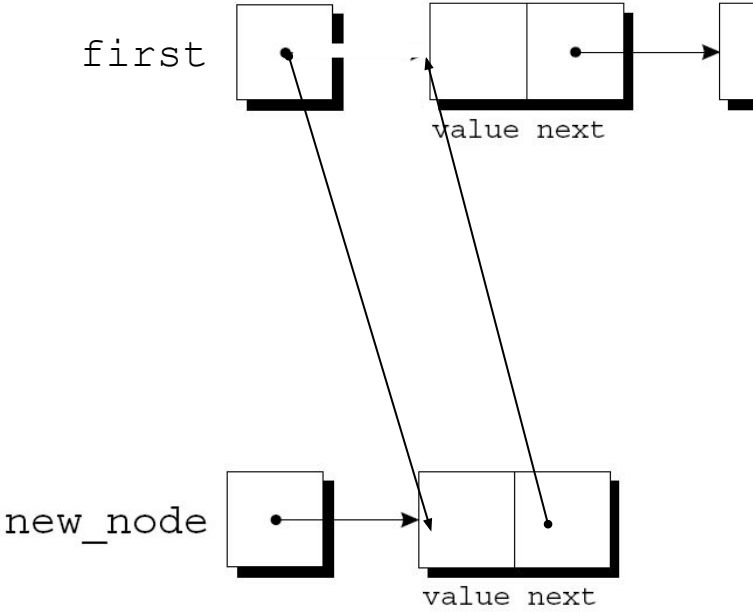
Pointers to Pointers

```
struct node *add_to_list(struct node *list,
int n){
      struct node *new_node;
      …
      new_node->value = n;
      new_node->next = list;
      return new_node;
}
…
first = add_to_list(first, 10);
```



Pointers to Pointers

```
struct node *add_to_list(struct node *list,
int n){
      struct node *new_node;
      …
      new_node->value = n;
      new_node->next = list;
      return new_node;
}
…
first = add_to_list(first, 10);
```



Pointers to Pointers

- What if we want to modify this function that it can update the original linked list pointed by `list` directly (i.e., return no value)?

```
struct node *add_to_list(struct node *list, int n){
      struct node *new_node;
      new_node = malloc(sizeof(struct node));
      if (new_node == NULL) {
        printf("Error: malloc failed in add_to_list\n");
        exit();
      }
      new_node->value = n;
      new_node->next = list;
      return new_node;
}
```

Pointers to Pointers

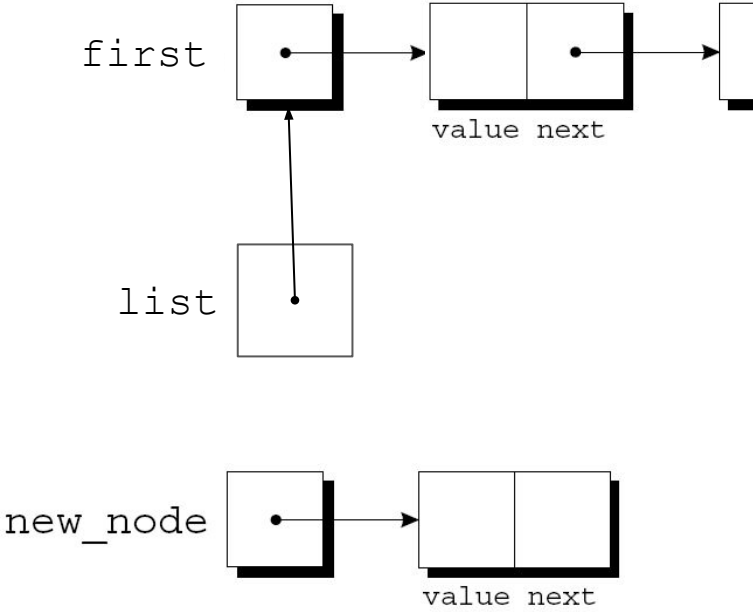
- Another alternative way of writing `add_to_list` is passing `add_to_list` a *pointer* to `first`

```
void add_to_list(struct node **list, int n){
      struct node *new_node;
      new_node = malloc(sizeof(struct node));
      if (new_node == NULL) {
        printf("Error: malloc failed in add_to_list\n");
        exit();
      }
      new_node->value = n;
      new_node->next = *list;
      *list = new_node;
}
```

Pointers to Pointers

```
void add_to_list(struct node **list, int n){
        struct node *new_node;
        new_node = malloc(sizeof(struct node));
        new_node->value = n;
        …
        new_node->next = *list;
        *list = new_node;
}
…
```

Please track this function by drawing the arrow on the right figure!



Pointers to Pointers

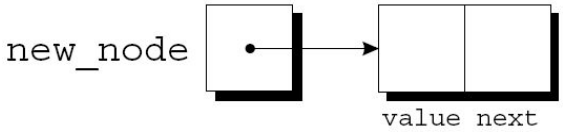
- When the new version of `add_to_list` is called, the first argument will be the address of `first`
- Since `list` is assigned the address of `first`, we can use `*list` as an alias for `first`.
- In particular, assigning `new_node` to `*list` will **modify** `first`.

```
void add_to_list(struct node **list, int n){
      …
}
…
add_to_list(&first, 10);
```

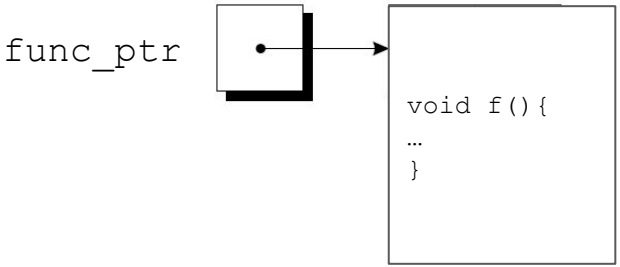
Let’s Take a Break!

Pointers to Functions

- In previous topics, we only see the pointers pointing to data. However, …
- C doesn't require that pointers point only to *data;* it's also possible to have pointers to *functions*.
- Functions occupy memory locations, so every function has an address.
- A function pointer points to code, not data. Typically a function pointer stores the start of executable code.




```
struct node *new_node;
```



https://www.geeksforgeeks.org/function-pointer-in-c/

Pointers to Functions

- We can use function pointers in much the **same** way we use pointers to data.
- Passing a function pointer as an argument is fairly **common**.
  - increase the flexbility of the program
  - several built-in functions in C used function pointers
- Let’s see the following examples to understand how to use function pointers!

Function Pointers as Arguments

- In the first example, a function named `integrate` that integrates a mathematical function `f` can be made as general as possible by passing `f` as an argument.
  - For example, if users want to integrate function named `log`, then user can pass `log` as an argument to `integrate` function
- The mathematical function `f` will have one `double` parameter and return `double` value

Function Pointers as Arguments

- Prototype and another alternative proprotype for `integrate`
- The parentheses around `*f` indicate that `f` is a pointer to a function.

```
double integrate(double (*f)(double), double a, double b);
// double integrate(double f(double), double a, double b);
```

Function Pointers as Arguments

- A call of `integrate` that integrates the `sin` (sine) function from 0 to π/2
- When a function name isn't followed by parentheses, the C compiler produces a pointer to the function.
- For example, the `sin` in the argument list of the function `integrate`
- We can do another call of `integrate` that integrates the `cos` (cosine) function
- We are not calling `sin/cos` at this point; we're passing `integrate` a pointer to `sin/cos`

```
result = integrate(sin, 0.0, PI / 2);
result = integrate(cos, 0.0, PI / 2);
```

Function Pointers as Arguments

- Within the body of `integrate`, we can call the function that `f` points to:
- `*f` represents the function that `f` points to; x is the argument to the call
- `*f` is actuall a call of `sine/cos` function

```
double integrate(double (*f)(double), double a, double b){
    …
    double y, x;
    …
    y = (*f)(x); // actually y = sin(x); or y = cos(x);
    …
}
```

Function Pointers as Arguments

- Writing `f(x)` instead of `(*f)(x)` is allowed.
- We will stick with `(*f)(x)` as reminder that `f` is a pointer to a function, not a function name

```
double integrate(double (*f)(double), double a, double b){
    …
    double y, x;
    …
    y = (*f)(x);
    …
}
```

Assign Function to Function Pointers

- Let’s see the next example to see how to assign a function to a function pointer

```
void fun(int a){
    printf("a:%d\n", a);
}

int main(){
    //declare function pointer fun_ptr pointing to fun()
    ___(1)___
    //call fun() and print “a:20”
    ___(2)___
    return 0;
}
```

slido

Based on the first example, what are the correct content for (1)?

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

slido

What are the correct content for (2) to print "a:20"?

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

The `qsort` Function

- Some of the most useful functions in the C library require a function pointer as an argument.
- One of these is `qsort`, which belongs to the `<stdlib.h>` header.
- `qsort` is a general-purpose **sorting** function that's capable of sorting any array.
- When using `qsort`, we must tell `qsort` how to determine which of two array elements is "smaller."
- This is done by **passing** `qsort` a **pointer** to a ***comparison function.***
- We need to write the comparison function by ourselves and pass its function pointer to `qsort`

The `qsort` Function

- In comparison function, when given two **pointers** `p` and `q` to **array** elements, the comparison function must return an integer that is:
  - *Negative* if `*p` is "less than" `*q`
  - *Zero* if `*p` is "equal to" `*q`
  - *Positive* if `*p` is "greater than" `*q`
- We need to define how `*p` and `*q` are compared.

The `qsort` Function

- Here is the prototype for `qsort`
- `base` must point to the first element in the array (or the first element in the portion to be sorted).
- `nmemb` is the number of elements to be sorted.
- `size` is the size of each array element, measured in bytes.

```
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

The `qsort` Function

- `compar` is a pointer to the comparison function.
  - return an intege
  - compare two elements pointed by pointers
- When `qsort` is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.

```
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

The `qsort` Function

- A call of `qsort` that sorts the `inventory` array (Structures, Unions, and Enumerations (2))
  - `inventory` (array name) is a pointer to the first element of the array
- `compare_parts` is a function that compares two `part` structures.

```
struct part {
     int number;
     char name[NAME_LEN+1];
     int on_hand;
};

struct part inventory[100];
qsort(inventory, 100, sizeof(struct part), compare_parts);
```

The `qsort` Function

- `qsort` requires that its parameters have type `void *`, but we can't access the members of a `part` structure through a `void *` pointer.
- To solve the problem, `compare_parts` will assign its parameters, `p` and `q`, to variables of type `struct part *`, `p1` and `q2`

The `qsort` Function

```
int compare_parts(const void *p,   const void *q){
     const struct part *p1 = p;
     const struct part *q1 = q;
     if (p1->number < q1->number){
       return -1;}
     else if (p1->number == q1->number){
       return 0;}
     else{
       return 1;}
}
```

- *Negative* if `*p` is "less than" `*q`
- *Zero* if `*p` is "equal to" `*q`
- *Positive* if `*p` is "greater than" `*q`

The `qsort` Function

```
int compare_parts(const void *p, const void *q){
     if (((struct part *) p)->number <
         ((struct part *) q)->number){
       return -1;}
     else if (((struct part *) p)->number ==
              ((struct part *) q)->number){
       return 0;}
     else{
       return 1;}
}
```

Most C programmers would write the function more concisely

The `qsort` Function

```
int compare_parts(const void *p, const void *q){
      return ((struct part *) p)->number -
             ((struct part *) q)->number;
}
```

Can be made even shorter by removing the if statements

The `qsort` Function

- A version of `compare_parts` that can be used to sort the `inventory` array by part **name** instead of part number
- by using `strcmp()` function

```
int compare_parts(const void *p, const void *q){
      return strcmp(((struct part *) p)->name,
                    ((struct part *) q)->name);
}
```

Summary

- Pointers to Pointers
  - Revised version of `add_to_list`
- Pointers to Functions
  - Function Pointers as Arguments
  - example of `integrate` and `void fun(int a)`
  - The `qsort` function