data_tructures(&algorithms)

Doan Trung Tung, PhD – University of Greenwich (Vietnam)

Plan

Introduction

Course overview, what is data structures, what is algorithms

Coding convention

Naming, Formatting

Review important concepts

Memory (static, heap), pointer, allocate memory, struct

Procedural Programming

Function, recursive function, procedural programming

Introduction

Course overview, what is data structures, what is algorithms

COURSE OVERVIEW

- Introduction
- Array & Linked List
- Sorting
- Searching
- Stack
- Queue
- Graph

ALGORITHMS

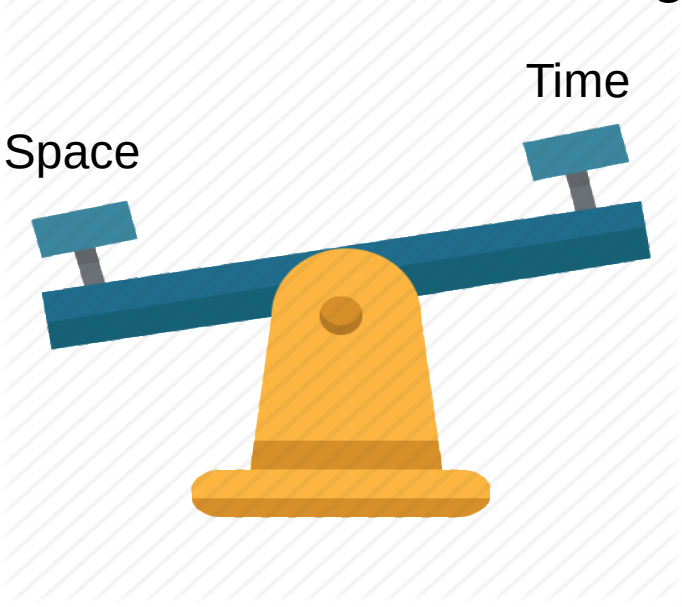
- Simple definition: A set of instructions to perform a specific task in a program.
- Popular algorithms:
  - Swap 2 numbers
  - Find min, max
  - Calculate sum, average
- Business related algorithms:
  - Show all employees information
  - Analyze a log file to detect errors

ALGORITHMS CHARACTERISTICS

- **Finiteness**: An algorithm must always terminate after a finite number of steps.
- **Independent**: An algorithm must not depend on any programming languages
- **Effectiveness**: An algorithm is also generally expected to be effective.
- **Unambiguous**: Algorithm should be clear and unambiguous.
- **Input / Output**: An algorithm takes input and produce output

EFFECTIVENESS

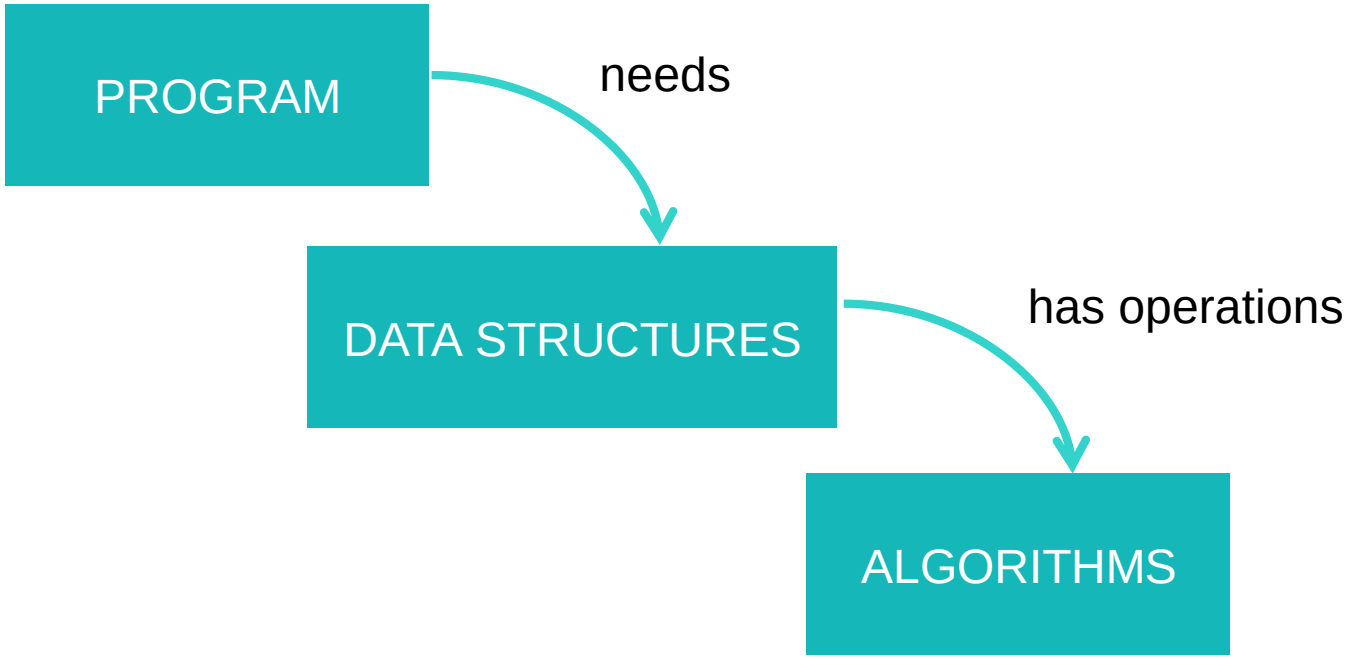
- Space: Algorithm should not consume too much space (memory)
- Time: Algorithm should run fast to get output



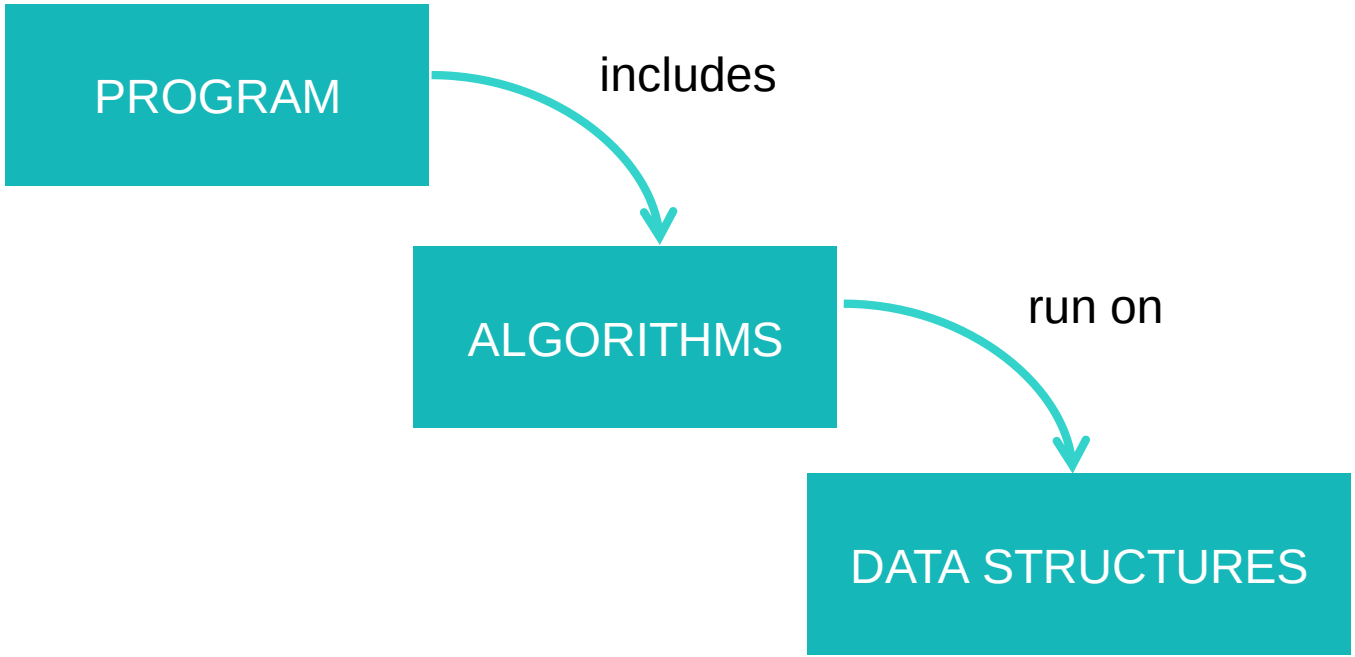
DATA STRUCTURE

- Scalar data: numbers, characters, logical values
  - Operations: +, -, *, /, AND, OR, etc.
- Collection data: string, array
  - Operations: size, length, insert, get at, find, etc.
- Struct data: date time
  - Operations: based on struct
- Built-in data is not enough for your problem?
  - Implement your own data structure: list, stack, queue, g raph, tree, etc.

DATA STRUCTURES & ALG.



DATA STRUCTURES & ALG.



BEFORE REVIEW

BASIC CODING CONVENTION

NAMING

- Variables: nouns in lower cases + underscore
  - name, first_name, start_time
- Constants: nouns in UPER CASES + UNDERSCORE
  - PI, MAX_SCORE, TOTAL_INTEREST
- Functions: verbs in lower cases + underscore
  - max, get_salary, quick_sort, show_info
- Names should be clear, explainable
- Plural form vs singular form:
  - for scalar: student, employee, house
  - for collection: students, employees, houses
- Number of: n_dogs, n_points

FORMATTING

- Use spaces and tabs correctly
  - Declaration and assignment:
    - Do : `int x = 5;`
    - Don't: `int x=5;`
  - Declaring and passing parameter:
    - Do : `my_func(int x, int y);`
    - Don't: `my_func ( int x ,int y ) ;`
    - Do : `my_func(6, 5);`
    - Don't: `my_func( 6,5 );`
  - Control statements:
    - Do : `if (a + b == 5)`
    - Don't: `if ( a+b==5 )`
    - Do : `for (int i = 0; i < 10; i++)`
    - Don't: `for(int i = 0 ;i <10;i++ )`

- Do : Tab
- Don't: Spaces

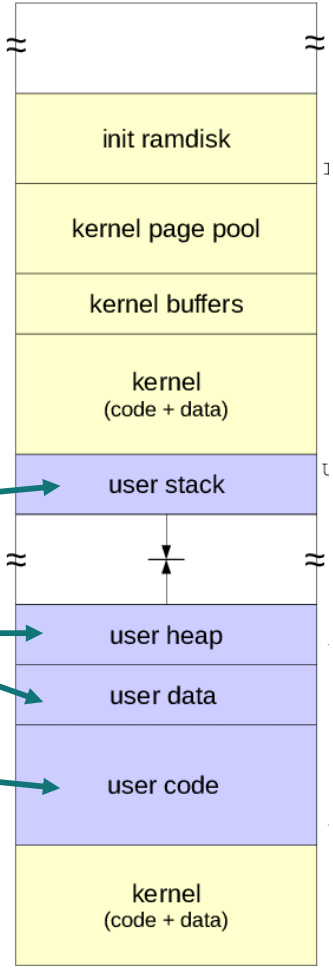
```
11 void test(int n)
12 {
13     for (int i = 0; i < n; i++)
14     {
15         printf("%d\n", i);
16     }
17 }
```

REVIEW

Memory

MEMORY IN A PROGRAM

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  float x = 10;
5
6  int main(int argc, const char * argv[]) {
7      int a = 10;
8      int *p = (int*)malloc(sizeof(int));
9      *p = a;
10     printf("Hello, World!\n");
11     return 0;
12 }
```



POINTER

- Pointers are variables whose values are memory addresses.
- Pointer can only point to memory block of same type (except void)

```
2  int a = 10;
3  // p contains address of a
4  int *p = &a;
5  // q contain address in heap allocated by malloc
6  int *q = (int*) malloc(sizeof(int));
7
8  printf("p = %x\nq = %x\n", p, q);
```

POINTER OPERATORS: &, *

- &: Address operator, returns address of a variable
- *: Dereferencing operator, returns value where the pointer points to.

```c
// Fig. 7.4: fig07_04.c
// Using the & and * pointer operators.
#include <stdio.h>

int main(void)
{
   int a = 7;
   int *aPtr = &a; // set aPtr to the address of a

   printf("The address of a is %p"
          "\nThe value of aPtr is %p", &a, aPtr);

   printf("\n\nThe value of a is %d"
          "\nThe value of *aPtr is %d", a, *aPtr);

   printf("\n\nShowing that * and & are complements of "
          "each other\n&*aPtr = %p"
          "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
}
```

POINTER OPERATORS: &, *

- &: Address operator, returns address of a variable
- *: Dereferencing operator, returns value where the pointer points to.

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

[1] C How to program 8th edition, Paul Deitel

POINTER OPERATORS: ++, --

- We can move pointer around by increment, decrement:
  - ++: move to the next memory block
  - --: move to the previous memory block

```
 2  int a[10];
 3  int *p = a; // p points to a or a[0]
 4
 5  *p = 5;     // a[0] = 5
 6  p++;        // p points to a[1]
 7
 8  *p = 10;    // a[1] = 10
 9
10  p--;        // p points to a[0]
11
12  p--;        // will cause problem or runtime error later
```

POINTER OPERATORS: ARITHMETICS

- We can apply some arithmetic operators on pointers (p, q must be the same type):
  - p = q + 2: move to the next 2 memory block
  - p = q - 2: move to the previous 2 memory block
  - other arithmetic operators (*, /, %) are not allowed
  - a = p – q: how many memory blocks between p & q
  - p == q: if p & q point to the same memory block
  - p = q: p & q point to the same memory block
  - p != q: p & q point to different memory blocks

POINTER: NULL & VOID

- A pointer can point to nowhere
  - int *p = NULL;
  - int *q = 0; // same as above but NULL is more prefer

- A pointer can point to anywhere
  - void *p = &a; // a is integer
  - p = &x; // x is float

POINTER: CONST

- A constant pointer always points to one memory block, cannot point to some where else.
  - Array is a constant pointer:
    - int a[10];
    - a = p; // error because a is constant
  - Constant pointer need to initiate at declaration:
    - int * const p = a;
    - p = q; // error: p can only points to a
  - Pointer that points to a constant value:
    - int b = 10;
    - const int* p = &b;
    - b = 20; // ok
    - *p = 20; // error: cannot change value through p
  - Constant pointer that points to a constant value:
    - const int* const p; // cannot change p, cannot change *p

ALLOCATE & DEALLOCATE

- Everything in stack will be removed automatically when ‘out of scope’
- Everything in heap can only be removed manually (deallocate)
- To allocate a memory block in heap:
  - int *p = (int *) malloc(sizeof(int));
- To allocate n memory blocks in heap:
  - int *p = (int *) malloc(n * sizeof(int));
  - int *p = (int *) calloc(n, sizeof(int));
- To deallocate memory in heap:
  - free(p);

WHY POINTER? WHY HEAP?

- Passing parameters to a function by reference

```c
int main()
{
    int a = 5, b = 6;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);

    return 0;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

WHY POINTER? WHY HEAP?

- Dynamic size arrays

```
 1  int main() {
 2      int a[5] = {1, 2, 3, 4, 5}; // a is fixed with always 5 elements
 3      print(a, 5);
 4      // allocate an array in heap with 5 elements
 5      int *b = (int*) malloc(5 * sizeof(int));
 6      // copy array a from stack to the array in heap
 7      copy(a, b, 0, 0, 5);
 8      print(b, 5);
 9      // reallocate the array in heap to 10 elements
10      b = (int*) realloc(b, 10 * sizeof(int));
11      // copy array a from stack to the new 5 elements
12      copy(a, b, 0, 5, 5);
13      print(b, 10);
14      free(b);
15      return 0;
16  }
```

WHY POINTER? WHY HEAP?

- Creating, returning an array in a function
- Working with large array

```
1  int* create_static_array(const int n)
2  {
3      int a[n];
4      for (int i = 0; i < n; i++) a[i] = i;
5      return a;
6  }
7
8  int* create_dynamic_array(const int n)
9  {
10     int *a = (int*) malloc(n * sizeof(int));
11     for (int i = 0; i < n; i++) a[i] = i;
12     return a;
13 }
```

```
int *a = create_static_array(5);
print(a, 5);

a = create_dynamic_array(5);
print(a, 5);

free(a);
```

```
  0 48-27263262432766-272632848
  0  1  2  3  4
Program ended with exit code: 0
```

MEMORY ERRORS

- Access to memory bock which has not been allocated
  - Out of range
  - NULL
- Forget to free memory (leaked memory)
- Deallocate a memory block in heap which has already freed before

MEMORY ERRORS

```
1  char *s = "hello";
2  strcpy(s, "hello world");          // error: access memory that is out of range
3
4  int *p;
5  *p = 10;                           // error: access memory which is NULL
6  int *q = NULL;
7  *q = 10;                           // error: access memory which is NULL
8
9  q = (int*) malloc(sizeof(int));
10 p = (int*) malloc(sizeof(int));
11
12 p = q;                             // leaked memory: now memory pointed by p has no ref
13 free(p);
14 free(q);                           // error: free memory block again
```

REVIEW

Struct

STRUCT

- Structures are collections of related variables under one name.
- Define and initialize a struct

```c
struct card
{
    char *face;
    char *suit;
};

struct student
{
    int id;
    char* name;
};
```

```c
int main(int argc, const char * argv[]) {

    struct card ace_heart = {"Ace", "Heart"};

    struct student john;
    john.id = 1;
    john.name = "John Lennon";

    return 0;
}
```

STRUCT

- Structures are collections of related variables under one name.
- Define and initialize a struct using typedef

```c
typedef struct
{
    char *face;
    char *suit;
} card;

typedef struct
{
    int id;
    char* name;
} student;
```

```c
int main(int argc, const char * argv[]) {

    card ace_heart = {"Ace", "Heart"};

    student john;
    john.id = 1;
    john.name = "John Lennon";

    return 0;
}
```

STRUCT

- Accessing members of a struct
  - Structure member operator (.)

```
17      student john;
18      john.id = 1;
19      john.name = "John Lennon";
```

  - Structure pointer operator (->)

```
25      student *john = (student*) malloc(sizeof(student));
26      john->id = 1;
27      john->name = "John Lennon";
```

REVIEW

Functions & procedural programming

CODING CONVENTION

- Length: should not be long, normally fit in a screen or ~ 25 lines of code
- Function should do one thing and only one thing
- Name should be clear and describe what ‘one thing’ is
- Number of parameters should be limited

PASS BY VALUE VS REFERENCE

- Parameter can be passed to a function by value
  - Value is copied from outside to the argument inside the function

```
1  void swap(int a, int b)
2  {
3      int temp = a;
4      a = b;
5      b = temp;
6  }
```

doesn't work as expected

- or by reference (pointer)
  - Variable outside and memory block (pointed by pointer) inside the function is the same

```
1  void swap(int *a, int *b)
2  {
3      int temp = *a;
4      *a = *b;
5      *b = temp;
6  }
```

work as expected

PASS ARRAY TO A FUNCTION

- Array must be initiated before passing, need size as well
- Cannot change array inside the function but can change content of array

```
void print_array(int a[], const int n)
{
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
}
```

```
void enter_array(int a[], const int n)
{
    printf("Enter %d element: ", n);
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);
}
```

```
int a[5];
enter_array(a, 5);
print_array(a, 5);
```

DYNAMIC ARRAY TO A FUNCTION

- Must use pointer instead of array
- May not need to be initiated before passing
- Can change pointer (if needed), can change size

```
void create_array(int **p, const int n)
{
    printf("Enter array: ");
    *p = (int*) malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) scanf("%d", &(*p)[i]);
}
```

```
void double_array(int *a, const int n)
{
    a = realloc(a, n * 2);
    for (int i = n; i < 2 * n; i++) a[i] = 0;
}
```

```
int *p;
create_array(&p, 5);
print_array(p, 5);
double_array(p, 5);
print_array(p, 10);
```

FUNCTION RETURNS ARRAY

- If array is created in stack, return it outside is meaningless
- Must use pointer and dynamic array in heap

```
1  int* create_static_array(const int n)
2  {
3      int a[n];
4      for (int i = 0; i < n; i++) a[i] = i;
5      return a;
6  }
7
8  int* create_dynamic_array(const int n)
9  {
10     int *a = (int*) malloc(n * sizeof(int));
11     for (int i = 0; i < n; i++) a[i] = i;
12     return a;
13 }
```

```
int *a = create_static_array(5);
print(a, 5);

a = create_dynamic_array(5);
print(a, 5);

free(a);
```

```
  0 48-27263262432766-272632848
  0  1  2  3  4
Program ended with exit code: 0
```

RECURSIVE FUNCTION

```
1  // Fig. 5.18: fig05_18.c
2  // Recursive factorial function.
3  #include <stdio.h>
4
5  unsigned long long int factorial(unsigned int number);
6
7  int main(void)
8  {
9     // during each iteration, calculate
10    // factorial(i) and display result
11    for (unsigned int i = 0; i <= 21; ++i) {
12       printf("%u! = %llu\n", i, factorial(i));
13    }
14 }
15
16 // recursive definition of function factorial
17 unsigned long long int factorial(unsigned int number)
18 {
19    // base case
20    if (number <= 1) {
21       return 1;
22    }
23    else { // recursive step
24       return (number * factorial(number - 1));
25    }
26 }
```

- A recursive function is one that calls itself either directly or indirectly through another function.
  - base case: where it stops
  - recursive case: where it goes on

[1] C How to program 8th edition, Paul Deitel

PASSING A STRUCT TO A FUNCTION

- Pass by value:
  - `void print(student s);`
- Pass by reference:
  - constant content: `void print(const student *s)`
  - constant content & pointer: `void print(const student* const s);`
  - allow to change: `void enter_student(student *s);`

```
 7  typedef struct
 8  {
 9      int id;
10      char* name;
11  } student;
```

PROCEDURAL PROGRAMMING

- Demo: A small menu-based program to manage students