ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY
한-베정보통신기술대학교

Nhân bản – Phụng sự – Khai phóng

C/C++ Language Review

Data Structures & Algorithms

CONTENT

- Reminder of C/C++
- Structures
- Pointers
- Dynamic memory allocation
- Recursion

CONTENT

- **Reminder of C/C++**
- Structures
- Pointers
- Dynamic memory allocation
- Recursion

- **Input**
  - cin >> (C++) / scanf() (C)
- **Output**
  - cout << (C++) / printf() (C)
- **Data types**
  - int, short, long
  - float, double
  - char
  - bool (C++)/ C99 standard for C language
  - string (C++)
- **Control statements**
  - if … else, switch … case
  - while, do … while, for
  - break, continue
- **Functions**
- **References** (C++)
- **Pointers**
- **Arrays**
- **C-string**
- **Files**

CONTENT

- Reminder of C/C++
- **Structures**
- Pointers
- Dynamic memory allocation
- Recursion

Structures

- **Structure**
  - C/C++ construct that allows multiple variables to be grouped together
- **General Format**

```
struct <StructName>
{
  type1 field1;
  type2 field2;
  . . .
};
```

```
struct Student
{
   int studentID;
   string name;
    short yearInSchool;
    double gpa;
};
```

structure name

structure members

Example `struct` Declaration

- **Defining Variables & Accessing Members**
  - To define variables, use structure name as type name

```
Student bill;
```

  - Use the dot (.) operator to refer to members of struct variables:

```
cin >> stu1.studentID;
getline(cin, stu1.name);
stu1.gpa = 3.75;
```

- **Comparing `struct` Variables**
  - Cannot compare `struct` variables directly:

```
if (bill == william)    // won't work
```

  - Instead, must compare on a field basis:

```
if (bill.studentID == william.studentID)...
```

- **Arrays of Structures**
  - Structures can be defined in arrays
  - Can be used in place of parallel arrays

```
const int NUM_STUDENTS = 20;
Student stuList[NUM_STUDENTS];
```

  - Individual structures accessible using subscript notation
  - Fields within structures accessible using dot notation

```
cout << stuList[5].studentID;
```

- **Nested Structures**
  - A structure can contain another structure as a member

```
struct PersonInfo
{       string name,
                address,
                city;
};

struct Student
{       int studentID;
        PersonInfo pData;
        short yearInSchool;
        double gpa;
};
```

- **Members of Nested Structures**
  - Use the dot operator multiple times to refer to fields of nested structures

```
Student s;
s.pData.name = "Joanne";
s.pData.city = "Tulsa";
```

- **Structures as Function Arguments**
  - May pass members of struct variables to functions

```
computeGPA(stu.gpa);
```

  - May pass entire struct variables to functions

```
showData(stu);
```

  - Can use reference parameter if function needs to modify contents of structure variable

- Example

```
struct InventoryItem
{
    int partNum;                    // Part number
    string description;             // Item description
    int onHand;                     // Units on hand
    double price;                   // Unit price
};
```

```
void showItem(InventoryItem p)
{
    cout << fixed << showpoint << setprecision(2);
    cout << "Part Number: " << p.partNum << endl;
    cout << "Description: " << p.description << endl;
    cout << "Units On Hand: " << p.onHand << endl;
    cout << "Price: $" << p.price << endl;
}
```

- **Structures as Function Arguments – Notes**
  - Using value parameter for structure can slow down a program, waste space
  - Using a reference parameter will speed up program, but function may change data in structure
  - Using a `const` reference parameter allows read-only access to reference parameter, does not waste space, speed

- **Revised showItem Function**

```
void showItem(const InventoryItem &p)
{
   cout << fixed << showpoint << setprecision(2);
   cout << "Part Number: " << p.partNum << endl;
   cout << "Description: " << p.description << endl;
   cout << "Units On Hand: " << p.onHand << endl;
   cout << "Price: $" << p.price << endl;
}
```

- **Returning a Structure from a Function**
  - Function can return a `struct`

```
Student getStudentData();   // prototype
stu1 = getStudentData();    // call
```

  - Function must define a local structure
    - for internal use
    - for use with `return` statement

- **Example**

```
Student getStudentData()
{   Student tempStu;
    cin >> tempStu.studentID;
    getline(cin, tempStu.pData.name);
    getline(cin, tempStu.pData.address);
    getline(cin, tempStu.pData.city);
    cin >> tempStu.yearInSchool;
    cin >> tempStu.gpa;
    return tempStu;
}
```

CONTENT

- Reminder of C/C++
- Structures
- **Pointers**
- Dynamic memory allocation
- Recursion

- Pointer is a special variable that stores address of another variable
- Definition

```
int  *intptr;
```

- Read as

"intptr can hold the address of an int"

- Spacing in definition does not matter

```
int * intptr;  // same as above
int*  intptr;  // same as above
```

- **Assigning an address to a pointer variable**

```
int *intptr;
intptr = &num;
```

- **The indirection operator (*) dereferences a pointer**

- **It allows you to access the item that the pointer points to**

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

This prints 25

- **Pointers to Structures**
  - A structure variable has an address
  - Pointers to structures are variables that can hold the address of a structure

```
Student *stuPtr;
```

  - Can use `&` operator to assign address

```
stuPtr = & stu1;
```

  - Structure pointer can be a function parameter

- **Accessing Structure Members via Pointer Variables**
  - Must use () to dereference pointer variable, not field within structure

```
cout << (*stuPtr).studentID;
```

  - Can use structure pointer operator to eliminate () and use clearer notation

```
cout << stuPtr->studentID;
```

- Example

```
void getData(Student *s)
{
    // Get the student name.
    cout << "Student name: ";
    getline(cin, s->name);

    // Get the student ID number.
    cout << "Student ID Number: ";
    cin >> s->idNum;

    // Get the credit hours enrolled.
    cout << "Credit Hours Enrolled: ";
    cin >> s->creditHours;

    // Get the GPA.
    cout << "Current GPA: ";
    cin >> s->gpa;
}
```

CONTENT

- Reminder of C/C++
- Structures
- Pointers
- **Dynamic memory allocation**
- Recursion

- **Allocating storage for a variable while program is running**
- **Computer returns address of newly allocated variable**
- **Uses `new` operator to allocate memory (C++)**

```
double *dptr;

dptr = new double;
```

⇨ **`new`** returns address of memory location

- **Can also use `new` to allocate array**

```
const int SIZE = 25;
arrayPtr = new double[SIZE];
```

- **Can then use `[]` or pointer arithmetic to access array**

```
for(i = 0; i < SIZE; i++)
    *arrayptr[i] = i * i;
```

or

```
for(i = 0; i < SIZE; i++)
    *(arrayptr + i) = i * i;
```

- **Program will terminate if not enough memory available to allocate**

- **Releasing Dynamic Memory**
  - Use **`delete`** to free dynamic memory (C++)

```
delete dptr;
```

  - Use **`[]`** to free dynamic array

```
delete [] arrayptr;
```

- **malloc(size_t size)** – C language
  - Allocates size bytes and returns a pointer to the allocated memory.
  - The memory is not cleared.

- **free(void * p)** – C language
  - Frees the memory space pointed to by p, which must have been returned by a previous call to **malloc()**, **calloc()**, or **realloc()**.
  - If free(p) has already been called before, undefined behavior occurs.
  - If p is NULL, no operation is performed.

- **Example**

```
#include <stdlib.h>
...
int *p = malloc(sizeof(int) * 3);
p[0] = 10;
p[1] = 20;
p[2] = 30;
...

free(p);
```

C++

- **Allocating memory**
  - Operator **new**
- **Releasing memory**
  - Operator **delete**
  - Operator **delete [ ]**

C

- **Allocating memory**
  - Functions **malloc(), calloc()**
- **Releasing memory**
  - Function **free()**

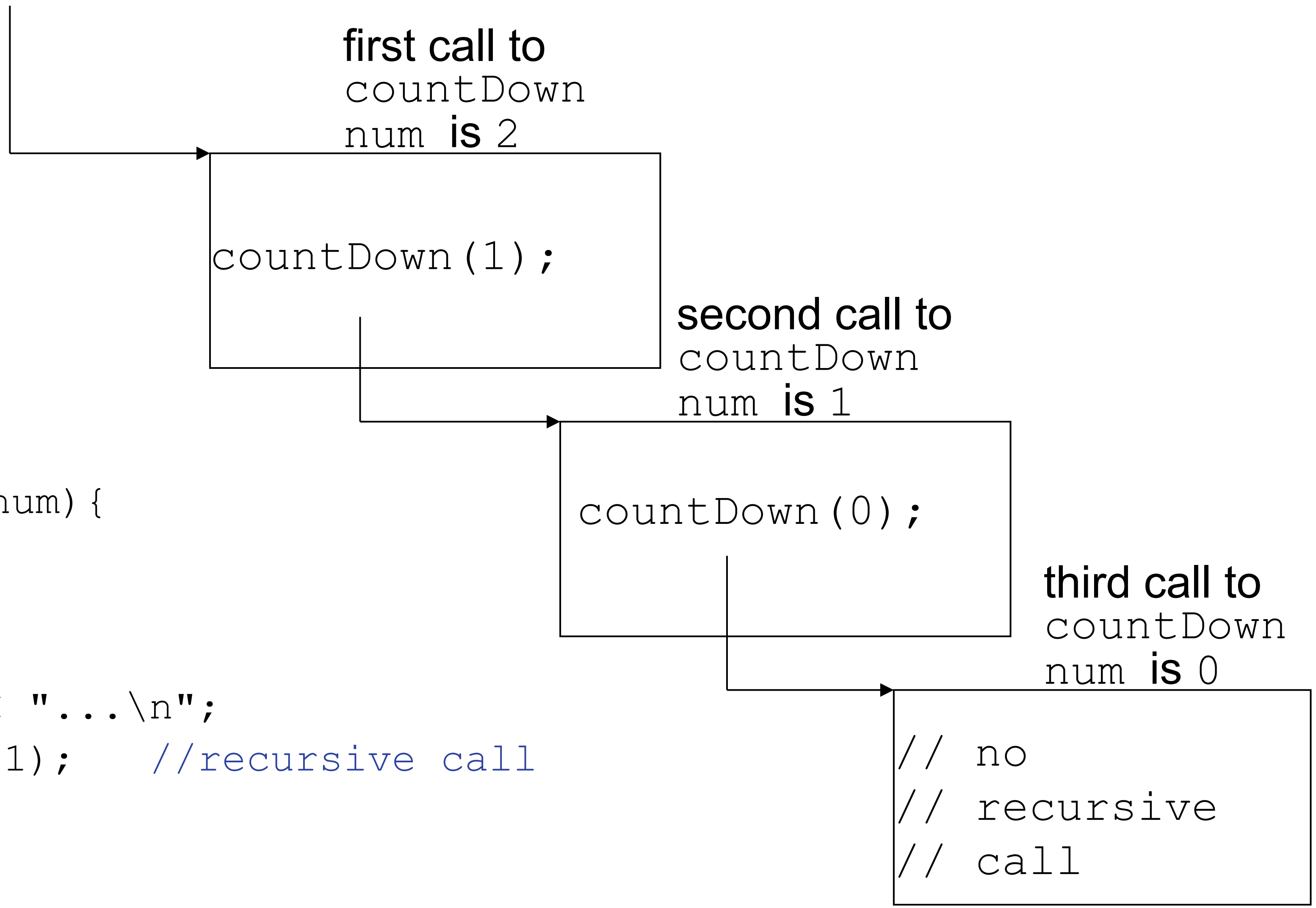
CONTENT

- Reminder of C/C++
- Structures
- Pointers
- Dynamic memory allocation
- **Recursion**

- A recursive function contains a call to itself

```
void countDown(int num)
{
   if (num == 0)
        cout << ”Go!";
   else
   {
        cout << num << "...\n";
        countDown(num-1);    //recursive call
   }
}
```

- **What happens when called?**




```
void countDown(int num){
   if (num == 0)
      cout << "Go!";
   else{
      cout << num << "...\n";
      countDown(num-1);    //recursive call
   }
}
```

output:

2...

1...

Go!

- **Recursive Functions - Purpose**
  - Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
  - The simplest-to-solve problem is solved directly
  - The simplest-to-solve problem is known as the base case/ stopping case
  - Recursive calls stop when the base case is reached

- **Stopping the Recursion**
  - A recursive function must always include **two** cases
    - a recursive call should be made until meeting the stopping case / base case
    - the recursion should stop
  - In the example, the stopping case is

```
if (num == 0)
```

- **Stopping the Recursion**

```
void countDown(int num)
   if (num == 0)
       cout << ”Go!";
   else{
       cout << num << "...\n";
       countDown(num-1);// note that the value
   }                     // passed to recursive
}                        // calls decreases by
                         // one for each call
```

- **Example**
  - The factorial function

    `n! = n*(n-1)*(n-2)*...*3*2*1` if `n > 0`

    `n! = 1` if `n = 0`
  - Can compute factorial of `n` if the factorial of `(n-1)` is known

    `n! = n * (n-1)!`
  - `n = 0` is the base case

- **Example**

```
int factorial (int num){
   if (num == 0)
         return 1;
   else
         return num * factorial(num - 1);
}
```

- **Fibonacci numbers**
  - Fibonacci numbers

    `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`

  - After the starting `0,1,` each number is the sum of the two preceding numbers

  - Recursive solution

    `fib(n) = fib(n – 1) + fib(n – 2)`

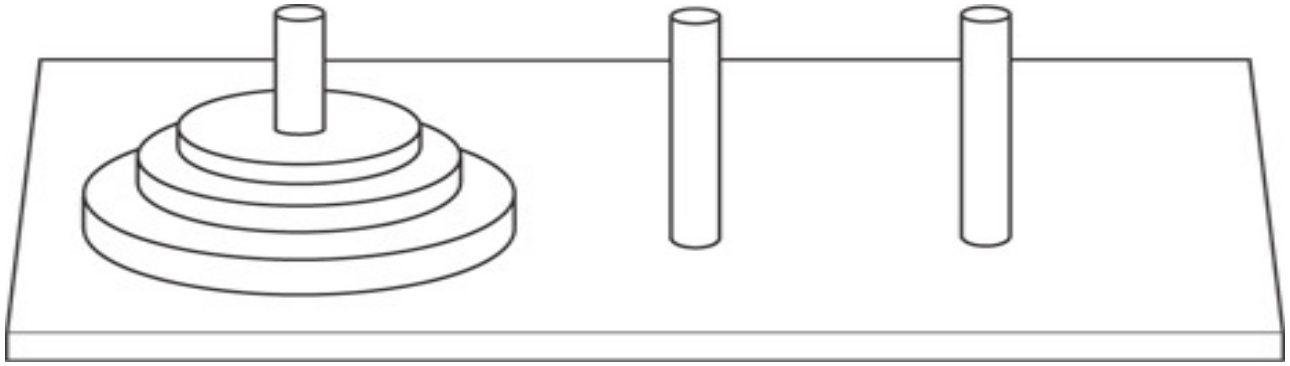
  - Base cases

    `n <= 0, n == 1`

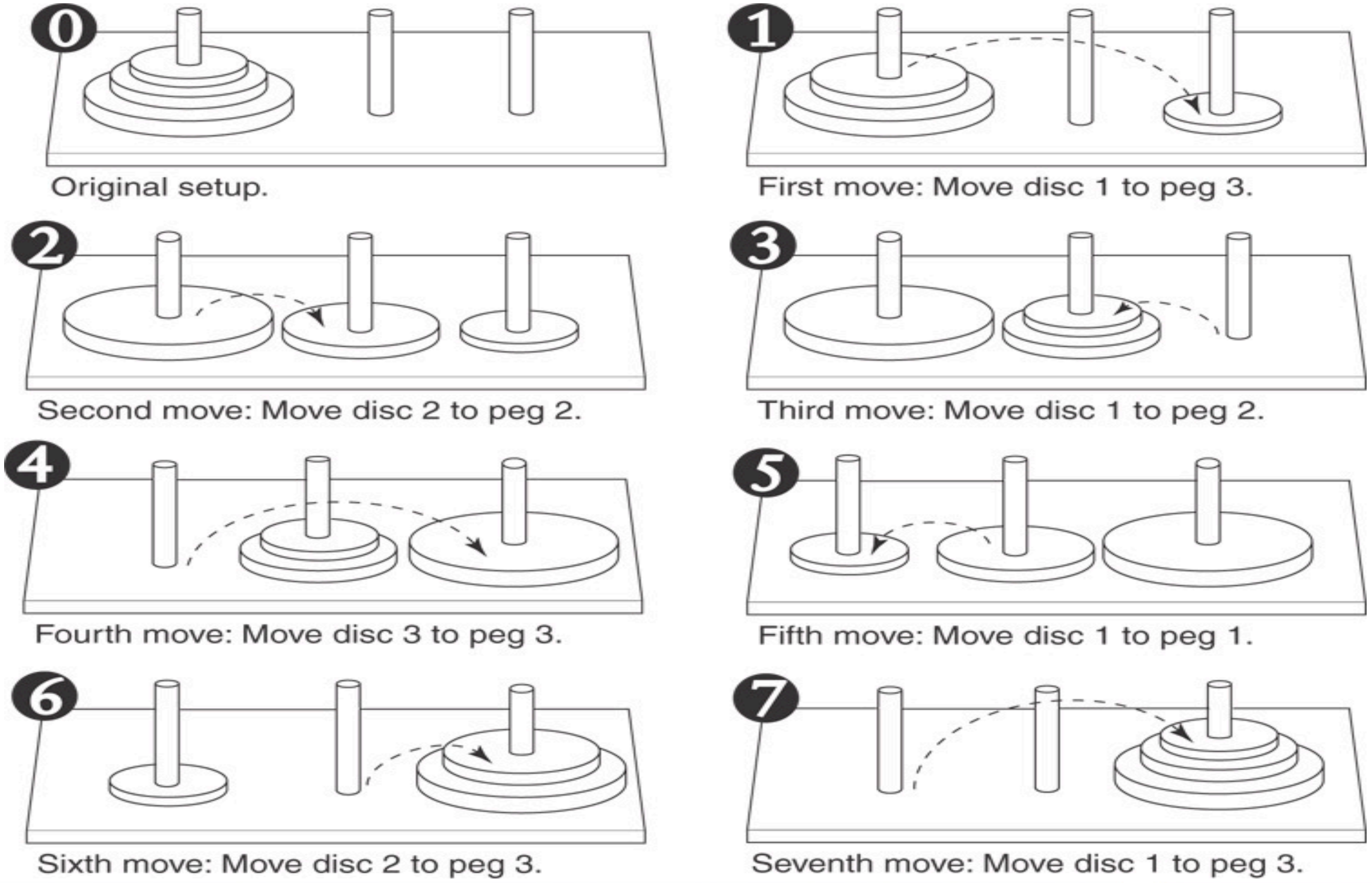
- **Fibonacci numbers**
  - Example

```
int fib(int n){
   if (n <= 0)
         return 0;
   else if (n == 1)
         return 1;
   else
         return fib(n – 1) + fib(n – 2);
}
```

- **The Towers of Hanoi**
  - The game uses three pegs A, B, C and a set of discs on peg A.
  - The goal is to move the discs from peg A to peg C by satisfying the following rules:
    - Only one disc may be moved at a time.
    - A disc cannot be placed on top of a smaller disc.
    - A peg can be used as temporary peg while moving disc.



- The Towers of Hanoi - **Moving Three Discs**



- The Towers of Hanoi - **Solution**
  - Suppose
    - We know how to move n-1 discs
  - Principle
    - To move n discs from peg A to peg C:
      - Move n–1 discs from peg A to peg B, using peg C as a temporary peg.
      - Move the remaining disc from the peg A to peg C.
      - Move n–1 discs from peg B to peg C, using peg A as a temporary peg.

- The Towers of Hanoi - **Algorithm**

```
Hanoi(n, A, B, C) {        //move n discs from peg A to peg C
        if (n == 1)        // base case
                move the disc from peg A to peg C;
        else {
                Hanoi(n-1, A, C, B);
                move the big disc from peg A to peg C;
                Hanoi(n-1, B, A, C);
        }
}
```

- The Towers of Hanoi - **Program**

```
void Hanoi(int n, char A, char B, char C){
    if (n == 1)
        cout << "move the disc from peg " << A
             << " to peg " << C << endl;
    else {
        Hanoi(n-1, A, C, B);
        cout << "move the disc from peg " << A
             << " to peg " << C << endl;
        Hanoi(n-1, B, A, C);
    }
}
```

SUMMARY

- Reminder of C/C++
- Structures
- Pointers
- Dynamic memory allocation
- Recursion

Enjoy the Course…!