Computer Programming II (CSE2035)
Enum, Structure, and Union

Slide by Prof. Joo Ho Lee

Instructor: Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

Seoul, Korea

Enumerated Types (Enum)

- An enumerated type is a special data type whose variables take specific symbols as values.
- Symbols must be used only once during the definition.
- Defined symbols are mapped to integer values. Useful when you want to give meanings to constants (other programmers is apt to understand the context).

```
enum state { absent, present, late, trip };
int main(void) {
    enum state s = absent; // Starts from 0 in default
    enum { mint = 3, mintchoco, chocomint = 7, choco } flavor;
    flavor = mint;
    ...
```

Question: what numbers are mapped to mintchoco and choco?

Enum against Macro

- Usages are similar.
- An enumerated type is a real type and its variable follows scoping rules.
- You don't have to type initial values and `#define` macro commands when you use enumerated types

```
#define EAST 0
#define WEST 1
#define SOUTH 2
#define NORTH 3

int direction1 = EAST;

enum Direction {East, West, South, North};

enum Direction direction2 = East;
```

Mistake in Using Enum

- **You can misuse a symbol from another enumerated type**
  - Bad practice

```
#include <stdio.h>

enum icecream {mint, mintchoco, choco, chocomint};
enum fruit {apple=3, pineapple, kiwi, peach};

int main(void){
    enum icecream a;
    a = chocomint;
    if (a == chocomint)
        printf("You know the taste.\n");

    // gcc doesn't raise any warning or error (while g++ does).
    a = apple;
    if (a == apple)
        printf("You know the taste.\n");

    return 0;
}
```

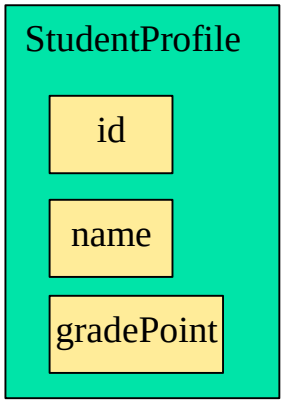
Structure (Struct)

- **"struct" keyword declares a set of data as a struct type.**
  - Each element is called a ***field***
- **We use '.' operator and '->' operator to access the fields.**

```
struct StudentProfile{
    int id;
    char name[26];
    double gradePoint;
};
```

Example struct definition

```
int main(void){
    struct StudentProfile chulsoo;
    struct StudentProfile *ps;

    strcpy(chulsoo.name, "Gang Chulsoo");
    chulsoo.id = 20081310;
    chulsoo.gradePoint = 4.3;

    ps = &chulsoo;
    printf("Student name is %s\n", ps->name);

    return 0;
}
```



Struct diagram

Typedef

- In C, you need to type "`struct A`" to declare its variable.
  - `struct StudentProfile chulsoo;`
- To shorten it, typedef expression gives another name to the existing type (good for improving code readability)
  - `typedef struct StudentProfile StudentProfile;`
  - `StudentProfile chulsoo;`
- You can also use `typedef` in several different ways, to make the code shorter
- In C++ (when you compile your code with g++), you don't need to use `typedef` command (a long story behind this)

Copying Structs

- **Variables defined in the same struct type can be copied with the assignment operator "="**

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

typedef struct{
    int id;
    char name[26];
    double gradePoint;
} StudentProfile;

void printProfile (StudentProfile x){
    printf("%d: %s (%.1f) \n", x.id, x.name, x.gradePoint);
}

int main(void){
    StudentProfile chulsoo, avatar;
    strcpy ( chulsoo.name, "Gang Chulsoo" );
    chulsoo.id = 20081310;
    chulsoo.gradePoint = 4.3;
    avatar = chulsoo;
    printProfile ( avatar );
    return 0;
}
```

This also invokes struct-to-struct copy

Call By Reference

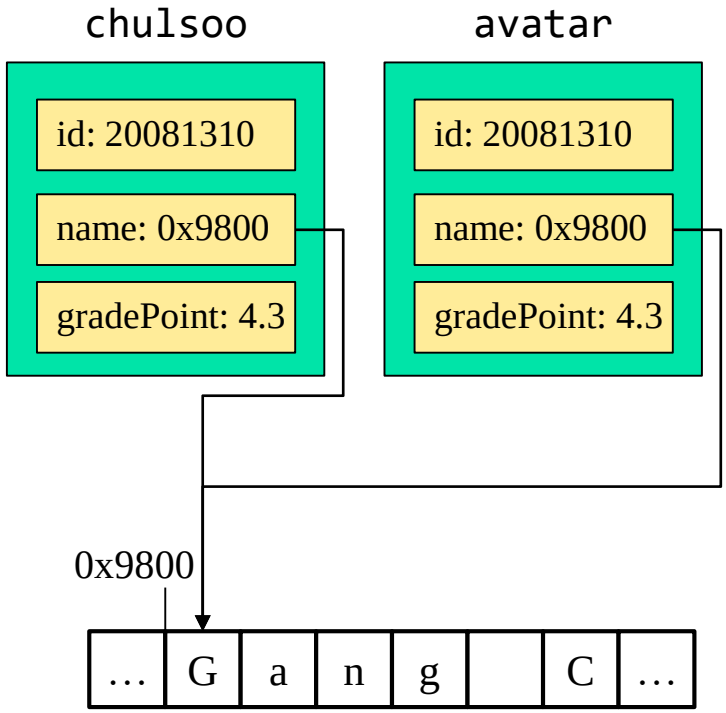
- **Call-by-reference is be more effective in passing structs**

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

typedef struct{
    int id;
    char name[26];
    double gradePoint;
} StudentProfile;

void printProfile (StudentProfile *x){
    printf("%d: %s (%.1f) \n", x->id, x->name, x->gradePoint);
}

int main(void){
    StudentProfile chulsoo, avatar;
    strcpy ( chulsoo.name, "Gang Chulsoo" );
    chulsoo.id = 20081310;
    chulsoo.gradePoint = 4.3;
    avatar = chulsoo;
    printProfile ( &avatar );
    return 0;
}
```

Struct with Pointer Field

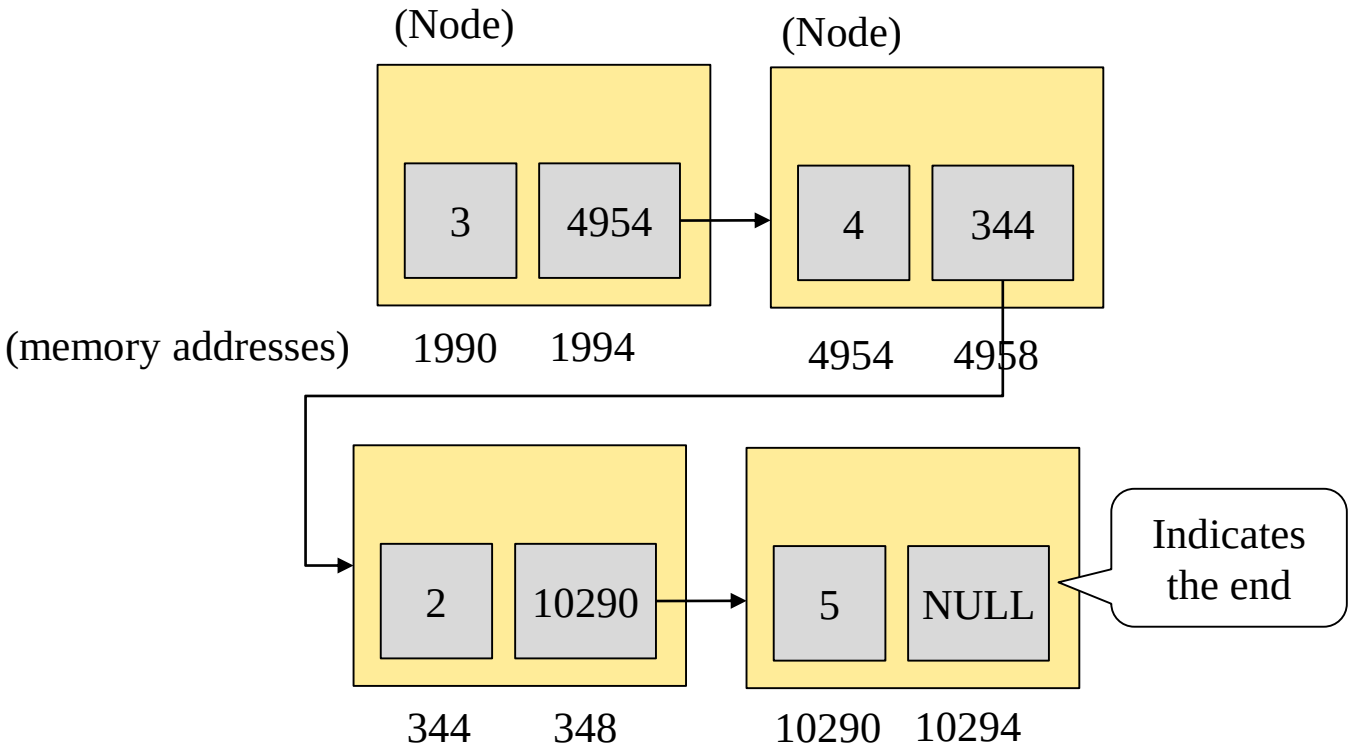
- **Be careful when you copy a struct that contains pointers as its fields (will be discussed deeply in the following weeks)**

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

typedef struct{
    int id;
    char *name;
    double gradePoint;
} StudentProfile;

int main(void){
    StudentProfile chulsoo, avatar;
    chulsoo.name = (char *) malloc(26);
    strcpy ( chulsoo.name, "Gang Chulsoo" );
    chulsoo.id = 20081310;
    chulsoo.gradePoint = 4.3;
    avatar = chulsoo;
    return 0;
}
```



Data Structure: Linked List

- **Different from an array, a linked list is a data structure that is not consecutively placed.**
- **Each "node" (a basic component that contain a value) links to the next node.**

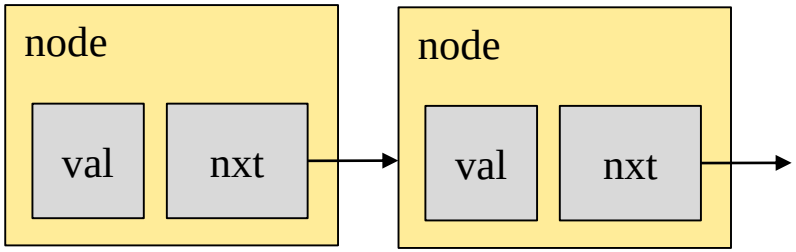


Node Definition with struct type

- **A struct type can include itself as a field (*recursive* data type)**
- **Note that such field must be included as a *pointer***
  - Think what would happen in the memory if "`nxt`" is not a pointer

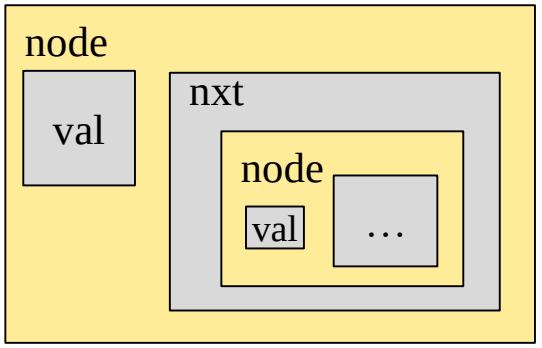
(Correct way)

```
struct Node {
    int val;
    struct Node *nxt;
};
```



(This doesn't compile)

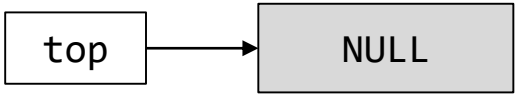
```
struct Node {
    int val;
    struct Node nxt;
};
```



Constructing List with Nodes

- **Think how nodes are generated and appended in the loop.**

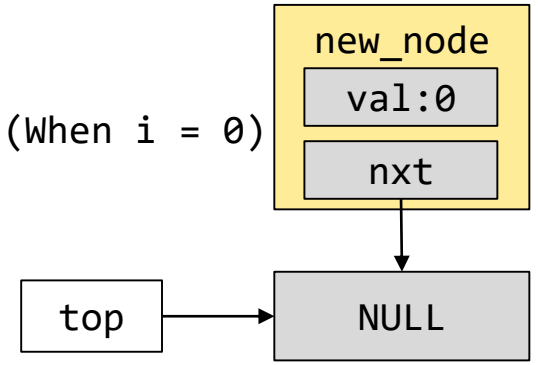
```c
typedef struct Node {
    int val;
    struct Node *nxt;
} Node;

int main(void) {
    int length = 4;
    Node *top = NULL;
    for (int i = 0; i < length; i++) {
        Node *new_node = (Node *) malloc(sizeof(Node));
        new_node->val = i;
        new_node->nxt = top;
        top = new_node;
    }

    Node *cur_node = top;
    while(cur_node != NULL) {
        printf("%d ", cur_node->val);
        cur_node = cur_node->nxt;
    }

    return 0;
}
```



Constructing List with Nodes

- **Think how nodes are generated and appended in the loop.**

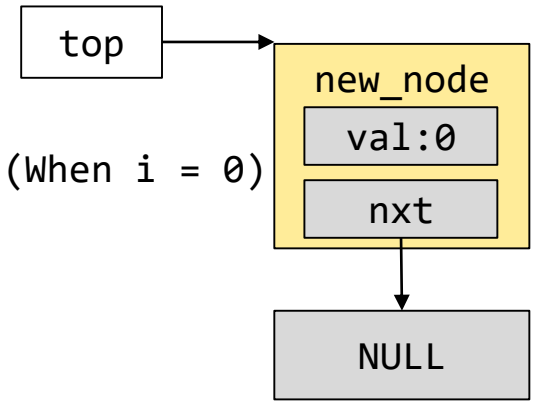
```c
typedef struct Node {
    int val;
    struct Node *nxt;
} Node;

int main(void) {
    int length = 4;
    Node *top = NULL;
    for (int i = 0; i < length; i++) {
        Node *new_node = (Node *) malloc(sizeof(Node));
        new_node->val = i;
        new_node->nxt = top;
        top = new_node;
    }

    Node *cur_node = top;
    while(cur_node != NULL) {
        printf("%d ", cur_node->val);
        cur_node = cur_node->nxt;
    }

    return 0;
}
```



Constructing List with Nodes

- **Think how nodes are generated and appended in the loop.**

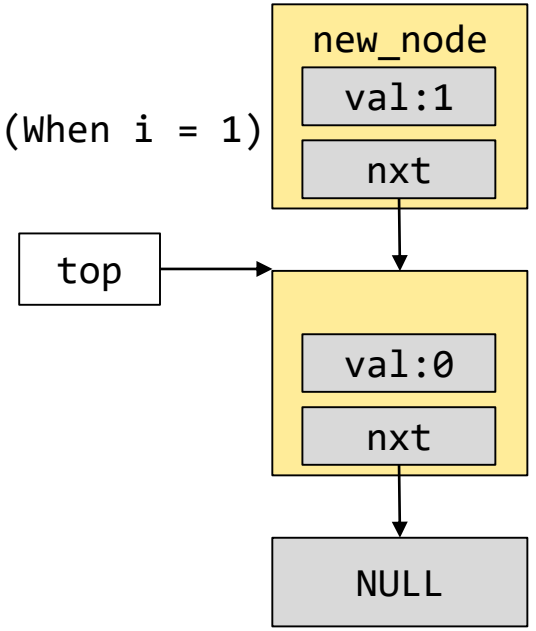
```c
typedef struct Node {
    int val;
    struct Node *nxt;
} Node;

int main(void) {
    int length = 4;
    Node *top = NULL;
    for (int i = 0; i < length; i++) {
        Node *new_node = (Node *) malloc(sizeof(Node));
        new_node->val = i;
        new_node->nxt = top;
        top = new_node;
    }

    Node *cur_node = top;
    while(cur_node != NULL) {
        printf("%d ", cur_node->val);
        cur_node = cur_node->nxt;
    }

    return 0;
}
```



Constructing List with Nodes

- **Think how nodes are generated and appended in the loop.**

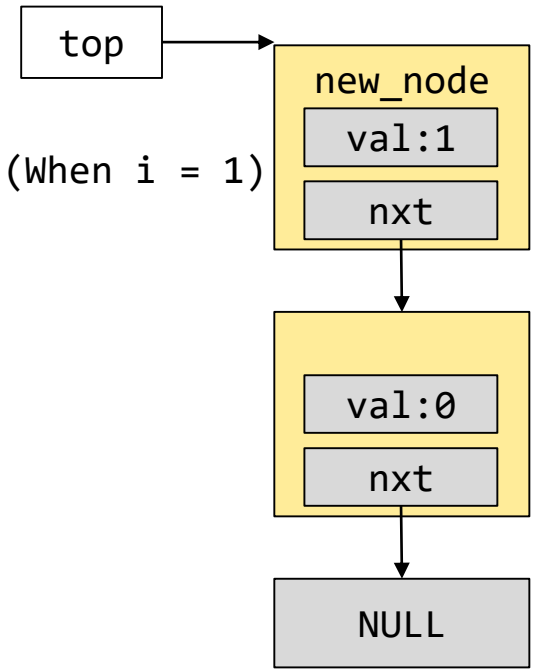
```
typedef struct Node {
    int val;
    struct Node *nxt;
} Node;

int main(void) {
    int length = 4;
    Node *top = NULL;
    for (int i = 0; i < length; i++) {
        Node *new_node = (Node *) malloc(sizeof(Node));
        new_node->val = i;
        new_node->nxt = top;
        top = new_node;
    }

    Node *cur_node = top;
    while(cur_node != NULL) {
        printf("%d ", cur_node->val);
        cur_node = cur_node->nxt;
    }

    return 0;
}
```



Constructing List with Nodes

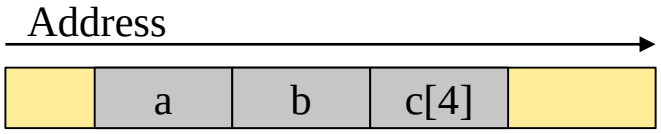
- **Think how nodes are generated and appended in the loop.**

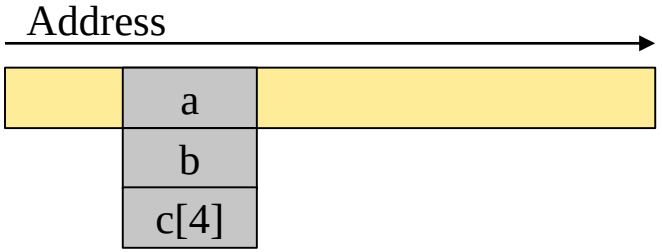
```c
typedef struct Node {
    int val;
    struct Node *nxt;
} Node;

int main(void) {
    int length = 4;
    Node *top = NULL;
    for (int i = 0; i < length; i++) {
        Node *new_node = (Node *) malloc(sizeof(Node));
        new_node->val = i;
        new_node->nxt = top;
        top = new_node;
    }

    Node *cur_node = top;
    while(cur_node != NULL) {
        printf("%d ", cur_node->val);
        cur_node = cur_node->nxt;
    }

    return 0;
}
```



Union

- A union is a special data type that defines multiple fields for the same data chunk.
- Only one member can be accessed at a time.
- By using different fields in union, the value is interpreted as the type of the accessed field.
- It is useful when we manipulate its binary representation.






```
struct A {
    int a;
    float b;
    char c[4];
};
```

```
union B {
    int a;
    float b;
    char c[4];
};
```

Example – Extracting a Binary Representation of a Float Value

- **Q. What would be the result of printing "temp.a"?**

```c
#include <stdio.h>

typedef union B{
    int a;
    float b;
    char c[4];
} B;

int main(){
    B temp;
    temp.c[0] = 'A'; // ASCII code: 0x41
    temp.c[1] = 'B'; // ASCII code: 0x42
    temp.c[2] = 'C'; // ASCII code: 0x43
    temp.c[3] = 'D'; // ASCII code: 0x44
    printf("0x%x\n", temp.a);
    printf("%f\n", temp.b);
    return 0;
}
```