CS100
Introduction to Programming

Lecture 6. Structures

Structures

- A **structure** is an aggregate of values, in which components are distinct and may possibly have different data types.

- For example, a **record** about a book in a library may contain:

```
char title[40];
char author[20];
float value;
int libcode;
```

Setting up a Structure Template

- A **structure template** is the master plan that describes how a structure is put together. To set up a structure template, e.g.

```
struct book {        /* template of book */
    char title[40];
    char author[20];
    float value;
    int libcode;
};
```

  - struct: the reserved keyword to introduce a structure
  - book: an optional tag name which follows the keyword “struct” to name the structure declared.
  - title, author, value and libcode: the members of the structure book.

- The above declaration declares a template, not a variable. No memory space is allocated.

Structures – Example

```
/* book.c -- one-book inventory */
#include <stdio.h>
struct book {
   char title[40];
   char author[20];
   float value;
   int libcode;
};
int main(void)
{
   struct book bookRec;
   printf(“Please enter the book title\n”);
   gets(bookRec.title);
   printf(“Now enter the author.\n”);
   gets(bookRec.author);
   printf(“Now enter the value.\n”);
   scanf(“%f”, &bookRec.value);
   printf(“%s by %s: $%.2f\n”, bookRec.title,
            bookRec.author, bookRec.value);
   return 0;
}
```

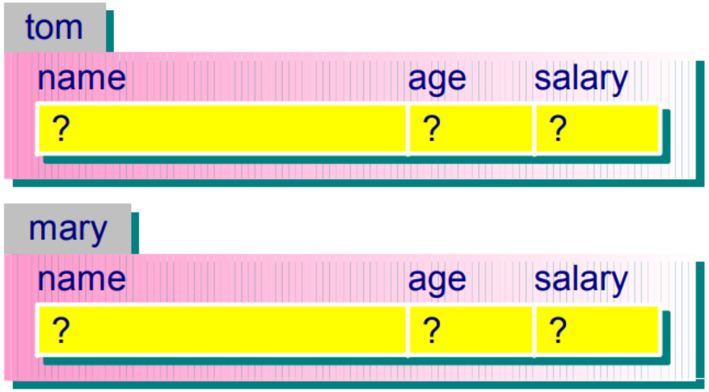
Output:

```
Please enter the book title:
The C Programming Language
Please enter the author:
K & R
Please enter the value:
63.65
The C Programming Language by K & R: $63.65
```

Defining a Structure Variable

- **With tag/name**: separate the definition of structure template from the definition of structure variable.

```
struct person {
   char name[20];
   int age;
   float salary;
};
struct person tom, mary;
```



- **Without tag/name**: combine the definition of structure template with that of structure variable.

```
struct {           /* no tag */
   char name[20];
   int age;
   float salary;
} tom, mary;
```

Structure Initialization

- Syntax for initializing structures is **similar to** that for initializing arrays.

- When there are insufficient values assigned to all members of a structure, the remaining members are assigned **zero** by default.

- Initialization of structure variables can only be performed with constant values or constant expressions which deliver values of the required types.

```
struct person{
   char name[20];
   int id;
   int tel;
};
struct person student = {“John”, 123, 20684863};
printf(“%s %d %d\n”, student.name, student.id, student.tel);
```

Output:

```
John 123 20684863
```

Structure Assignment and Access

Structure Assignment

- The values in one structure can be assigned to another:

```
struct person newMember;
newMember = student;
```

Accessing Structure Members

- Notation required to reference the members of a structure is

  structureVariableName.memberName

  as shown in the previous example

- The "**.**" is a member access operator known as the **member operator**.

Arrays of Structures

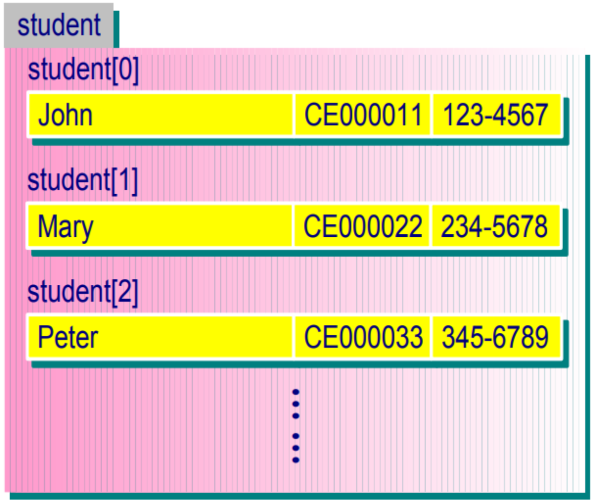
- A structure variable can be seen as a **record**
  - e.g. the structure variable **student** in the previous example is a student record with the information of a student’s name, address, id, etc.

- When student variables of the same type are grouped together, we have a **database** of that structure type.

- One can create a database by defining an **array** of certain structure type.

Arrays of Structures – Example

```
/* Define a database with up to 10 student records */
struct person {
   char name[20], id[20], tel[20];
};
person student[3] = {
   {“John”, “CE000011”, “123-4567”},
   {“Mary”, “CE000022”, “234-5678”},
   {“Peter”, “CE000033”, “345-6789”},
};
//struct keyword could be removed
//in many existing compilers

int main(void)
{
   int i;
   for (i=0; i < 3; i++) {
      printf(“Name: %s, ID: %s, Tel: %s.\n”,
         student[i].name, student[i].id, student[i].tel);
   }
}
```


student

student[0]

John	CE000011	123-4567

student[1]

Mary	CE000022	234-5678

student[2]

Peter	CE000033	345-6789

Output:

```
Name: John, ID: CE000011, Tel: 123-4567.
Name: Mary, ID: CE000022, Tel: 234-5678.
Name: Peter, ID: CE000033, Tel: 345-6789.
```

Nested Structures

- **A structure can also be included in other structures.**

- For example, to keep track of the course history of a student, one can use a structure (**without any nested structures**) like

```
struct student {
    char        name[40];
    char        id[20];
    char        tel[20];
    int         CS100Yr;        /* the year when CS100 is taken */
    int         CS100Sr;        /* the semester when CS100 is taken */
    char        CS100Grade;     /* the grade obtained for CS100 */
    int         CS102Yr;        /* the year when CS102 is taken */
    int         CS102Sr;        /* the semester when CS102 is taken */
    char        CS102Grade;     /* the grade obtained for CS102 */
}

student student[1000];
```

Nested Structures

- Alternatively, student can be defined in a more elegant manner, **using nested structures**, as

```
struct person  {
     char    name[40];
     char    id[20];
     char    tel[20];
};
struct course  {
     int      year, semester;
     char    grade;
};
struct student  {
     person studentInfo;
     course CS100, CS102;
};
student student[1000];
```

- **student** denotes the complete array (database)

```
student student[3] = {
      {{“John”, “CE000011”, “123-4567”},
            {2016, 1, ‘B’}, {2017, 1, ‘A’}},
      {{“Mary”, “CE000022”, “234-5678”},
            {2016, 1, ‘A’}, {2017, 1, ‘A’}},
      {{“Peter”, “CE000033”, “345-6789”},
            {2016, 1, ‘C’}, {2017, 1, ‘B’}},
};

/* To print individual elements of the new student array */
int i;
for (i=0; i <= 2; i++) {
      printf(“Name: %s, ID: %s, Tel: %s\n”,
            student[i].studentInfo.name,
            student[i].studentInfo.id,
            student[i].studentInfo.tel);
      printf(“CS100 in year %d semester %d : %c\n”,
             student[i].CS100.year,
             student[i].CS100.semester,
             student[i].CS100.grade);
      printf(“CS102 in year %d semester %d : %c\n”,
             student[i].CS102.year,
             student[i].CS102.semester,
             student[i].CS102.grade);
}
```

- **student[i]** denotes the (i+1)th record
- **student[i].studentInfo** denotes the personal information in the (i+1)th record
- **student[i].studentInfo.name** denotes the student’s name in this record
- **student[i].studentInfo.name[j]** denotes a single character value

Pointers to Structures

- Pointers are flexible and powerful in C. They can be used to point to structures.

```
/* The structure members can be accessed in 3 different ways,
   using pointers or not. */
struct person {
   char name[40], id[20], tel[20];
};
person student = {“John”, “CE000011”, “123-4567”};
person *ptr;
...
printf(“%s %s %s\n”, student.name, student.id, student.tel);
ptr = &student;
printf(“%s %s %s %s\n”, (*ptr).name, (*ptr).id, (*ptr).tel);
/* Why is the round brackets around *ptr needed? */
printf(“%s %s %s\n”, ptr->name, ptr->id, ptr->tel);
...
```

Pointers to Structures

- The operator **->** is called the **structure pointer operator**, which is reserved for a pointer pointing to a structure. Less typing is needed if one compares **ptr->tel** to **(*ptr).tel**
- **3 reasons for using pointers to structures:**
  - Pointers to structures are easier to manipulate than structures themselves;
  - In older C implementation, a structure is passed as an argument to a function using pointer to structure;
  - Many advanced data structures require pointers to structures.

Pointers to Structures: Example

```
#include <stdio.h>
struct book {
   char title[40];
   char author[20];
   float value;
   int libcode;
};

int main(void)
{
   book bookRec = {
      “The C Programming Language”, “K&R”, 63.65, 123
   };
   book *ptr;
   ptr = &bookRec;
   printf(“The book %s (%d) by %s: $%.2f.\n”, ptr->title,
         ptr->libcode, ptr->author, ptr->value);
   return 0;
}
```

Output:

The book The C Programming Language (123) by K&R: $63.65.

Dynamic Structure Construction

- **Dynamic allocation and content copy**
  - When structures need to be created dynamically

```
person *pMember =
        (struct person *) malloc(sizeof(person));
```

  - Copy structure contents

```
memcpy(pMember1, pMember2, sizeof(person));
```

- **Accessing Structure Members by pointers**

Notation required to reference the members of a structure is

structureVariableName->memberName

Dynamic Structure Construction

- Example

```
struct person{
   char name[20];
   int id;
   int tel;
};

person *pstudent = (persion *)malloc(sizeof(person));

if(pstudent!=NULL)
{
     printf(“%s %d %d\n”,
       pstudent->name, pstudent->id, pstudent->tel);
}

free(pstudent);
```

Dynamic Array of Structures

- **Dynamic array of structure allocation**

```
int student_num=0;
… //get student number

person *pstudents = (struct person *)
          malloc(sizeof(person)*student_num);

for(int i=0;i<student_num;i++)
{
      scanf(“name of student %d”, i+1, pstudents[i].name);
      //scanf(“name of student %d”, i+1, (pstudents +i)->name);
      …
}
… //do something else
free(pStudents);
```

Functions and Structures

- **Four ways to pass structure information to a function:**
  - Passing structure members as arguments using call-by-value, call-by-pointer or call-by-reference;
  - Passing structures as arguments;
  - Passing pointers/references to structures as arguments;
  - Passing by returning structure/pointer to structure.

Passing Structure Members as Argument

```
#include <stdio.h>
float sum(float, float);
struct account {
   char bank[32];
   float current;
   float saving;
};

int main(void)
{
   account john = {“Bank of China”, 1000.43, 4000.87};
   printf(“The account has a total of %.2f.\n”,
      sum(john.current, john.saving));   // pass by value
   return 0;
}

float sum(float x, float y)
{
   return (x + y);
}
```

Output:

The account has a total of 5001.30.

- **Pass by value**
- **struct members** are used as arguments

Passing Structure as Argument

```
#include <stdio.h>
struct account {
   char bank[32];
   float current;
   float saving;
};
float sum(account); // argument is a structure, ignoring
                    // the argument name

int main(void)
{
   account john = {“Bank of China”, 1000.43, 4000.87};
   printf(“The account has a total of %.2f.\n”,
      sum(john));    // pass by value
   return 0;
}

float sum(account money)
{
   return (money.current + money.saving);
}
```

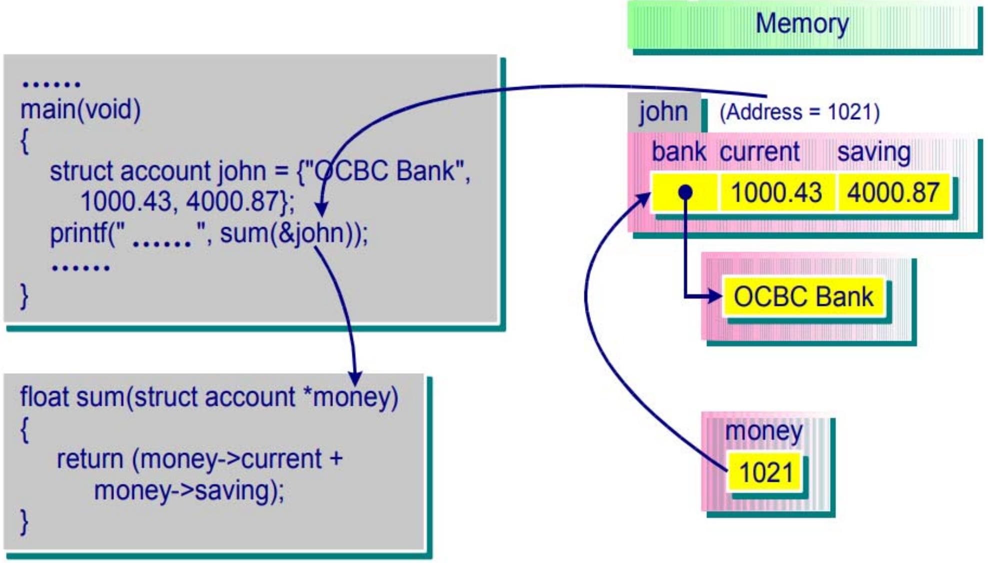
Output:

The account has a total of 5001.30.

- **Pass by value**
- **struct account money** is used as parameter

Passing Structure Address as Argument

```
#include <stdio.h>
struct account {
   char bank[20];
   float current;
   float saving;
};
float sum(account*);

int main(void)
{
   struct account john = {“OCBC Bank”, 1000.43, 4000.87};
   printf(“The account has a total of %.2f.\n”,
      sum(&john));   // pass by reference
   return 0;
}

float sum(account *money)
{
   return (money->current + money->saving);
}
```



```
......
main(void)
{
   struct account john = {"OCBC Bank",
      1000.43, 4000.87};
   printf(" ...... ", sum(&john));
   ......
}
```

```
float sum(struct account *money)
{
   return (money->current +
      money->saving);
}
```

- **Pass by pointer**
- **account *money** is used as parameter

Passing Structure Reference as Argument

```
#include <stdio.h>
struct account {
   char bank[20];
   float current;
   float saving;
};
float sum(account &);

int main(void)
{
   struct account john = {“OCBC Bank”, 1000.43, 4000.87};
   printf(“The account has a total of %.2f.\n”,
      sum(john)); // pass by reference
   return 0;
}

float sum(account &money)
{
   return (money.current + money.saving);
}
```

- **Pass by reference**
- **account *money** is used as parameter

Returning a Structure in Function

```
#include <stdio.h>
struct name {char first_name[20], last_name[20];};
int main(void) {
   name my_name;
   my_name = get_name();
   printf(“Your name is %s %s\n”,
         my_name.first_name, my_name.last_name);
   return 0;
}
name get_name(void) {
   name new_name;
   printf(“Enter first name: ”);
   gets(new_name.first_name);
   printf(“Enter last name: ”);
   gets(new_name.last_name);
   return new_name;
}
```

Output:

```
Enter first name: Li
Enter last name: Min
Your name is Li Min.
```

- When is it better to use structures?
- When is it better to use pointers to structures?
- How to pass an array of structures into a function?

Returning a Structure in Function

- **Sometimes it is not good to return a structure**
  - Use pointer or reference as return

```
struct name {char first_name[20], last_name[20];};
int main(void) {
   name my_name;
   get_name(&my_name);
   printf(“Your name is %s %s\n”,
         my_name.first_name, my_name.last_name);
   return 0;
}
void get_name(name *name_ret) {
   printf(“Enter first name: ”);
   gets(name_ret->first_name);
   printf(“Enter last name: ”);
   gets(name_ret->last_name);
}
```

The typedef Construct

- **typedef** provides an elegant way in structure declaration. For example, having

```
struct date { int day, month, year; };
```

one can define a new data type **Date** as

```
typedef struct date Date;
```

- Variables can be defined either as

```
date today, yesterday;
```

or

```
Date today, yesterday;
```

- When ***typedef*** is used, structure name is redundant, thus:

```
typedef struct {
        int day, month, year;
} Date;
Date today, yesterday;
```

The typedef Construct: Example

```
#include <stdio.h>
#define CARRIER 1
#define SUBMARINE 2
typdef struct {
    int shipClass;
    char *name;
    int speed, crew;
} warShip;

void printShipReport(warShip);

int main(void)
{
    warShip ship[10];
    int i;
    ship[0].shipClass = CARRIER;
    ship[0].name = “Liaoning”;
    ship[0].speed = 29;
    ship[0].crew = 3000;
```

```
    ship[1].shipClass = SUBMARINE;
    ship[1].name = “Changzheng-6”;
    ship[1].speed = 24;
    ship[1].crew = 140;
    for (i=0; i < 2; i++)
        printShipReport(ship[i]);
    return 0;
}

void printShipReport(warShip ship)
{
    if (ship.shipClass == CARRIER)
        print(“Carrier:\n”);
    else
        print(“Submarine:\n”);
    printf(“\tname = %s\n”, ship.name);
    printf(“\tspeed = %d\n”, ship.speed);
    printf(“\tcrew = %d\n”, ship.crew);
}
```

Size of a Structure

- The size of the structure is the summation of all member sizes

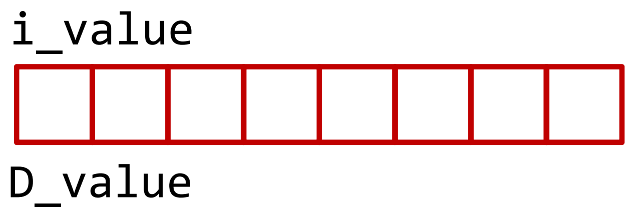
```
struct account {
   char bank[20];
   float current;
   float saving;
};

printf(“the size of account = %d\n”,sizeof(account));
```

```
sizeof(account) = sizeof(bank) + sizeof(current) sizeof(saving)
       = sizeof(char)*20 + sizeof(float) + sizeof(saving) = 28
```

Union

- A special data type to store different data types in the same memory location

```
struct index_data {
   int i_value[2];
   double d_value;
};
```

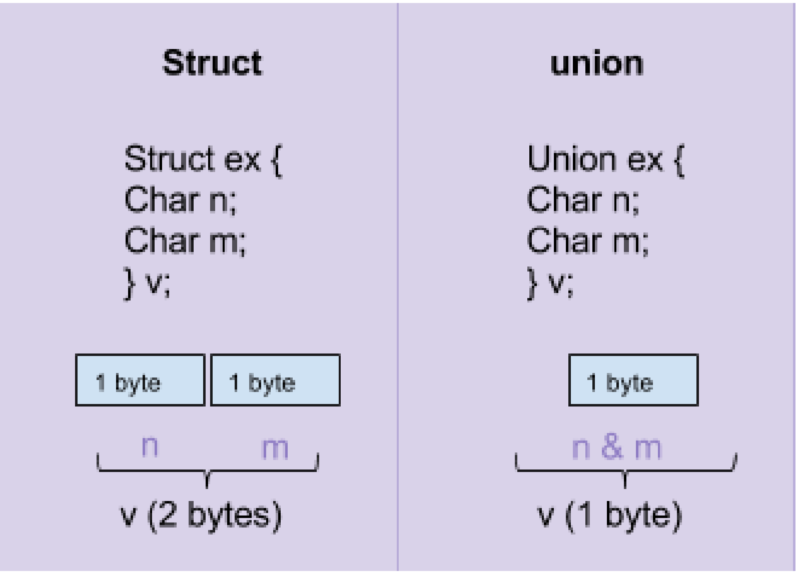


```
printf(“the size of index_data = %d\n”,sizeof(index_data));
```

```
sizeof(index_data) = max{sizeof(i_value), sizeof(d_value)} = 8
```

Difference between Structure and Union

- Consecutive memory v.s. overlapped (shared) memory



```
Struct ex {
Char n;
Char m;
} v;
```

```
Union ex {
Char n;
Char m;
} v;
```

Structure in a Union

- **Structure in a union is consecutive**
  - Share the memory with other union member

```
union vertex{
   struct{
       float x,y,z;
   };
   float data[3];
};

vertex v;
v.x=10.0f; v.y=5.2f; v.z=-6.8f;
printf(“vertex coordinate : %f, %f, %f\n”,
           v.data[0], v.data[1], v.data[2]);
```

An Example of Editing A Student List

- **Creating an array of student list**

```
struct student_info {
    char name[20];
    int id;
    float score;
};

int student_num=0;
scanf(“Please input the number of students: %d”,&student_num);

student_info *student_array=
         (student_info *)malloc(sizeof(student_info)*student_num);

printf(“Please input student info.\n\n”);
for(int i=0;i<student_num;i++)
{
     printf(“Inputting student %d...\n”, i+1);
     scanf(“student name: %s”, student_array[i].name);
     scanf(“student id: %d”, &student_array[i].id);
     scanf(“student score: %f”, &student_array[i].score);
}
```

An Example of Editing A Student List

- **Inserting some student information**

```
int student_num_insert=0;
scanf(“How many students you want to insert: %d”,&student_num_insert);

student_info *student_insert_array=
        (student_info *)malloc(sizeof(student_info)*student_num_insert);

printf(“Please insert student info.\n\n”);
for(int i=0;i<student_num_insert;i++)
{
     printf(“Inputting inserted student %d...\n”, i+1);
     scanf(“student name: %s”, student_insert_array[i].name);
     scanf(“student id: %d”, & student_insert_array[i].id);
     scanf(“student score: %f”, & student_insert_array[i].score);
}
```

An Example of Editing A Student List

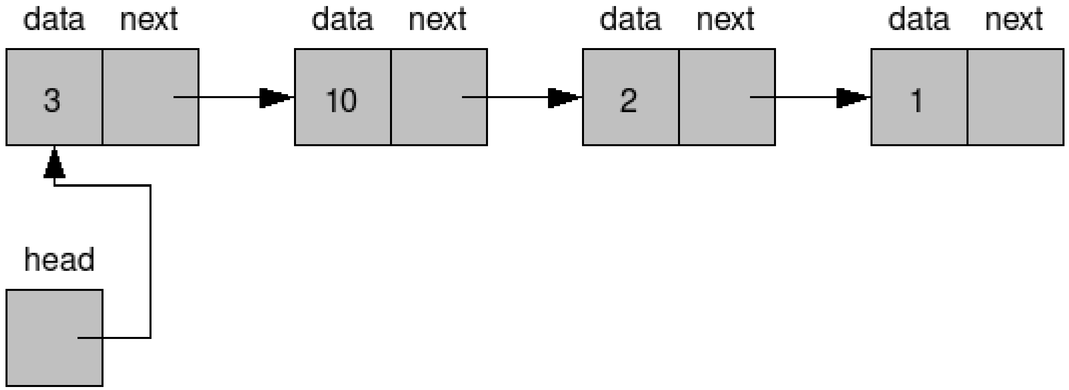
- **Inserting some student information**

```
int insert_index=0;
scanf(“Where do you want to insert: %d”,&insert_index);

student_info *student_array_new =
  (student_info *)malloc(sizeof(student_info)*student_num+student_num_insert);

for(int i=0;i<insert_index;i++)
{
      memcpy(&student_array_new[i], &student_array[i],
                              sizeof(student_info)); //any better way? Efficiency?
}
for(int i=insert_index;i<insert_index+student_num_insert;i++)
{
      memcpy(&student_array_new[i], &student_insert_array[i-insert_index],
                              sizeof(student_info));
}
for(int i=insert_index+student_num_insert;i<student_num+student_num_insert;i++)
{
      memcpy(&student_array_new[i], &student_array[i-student_num_insert],
                              sizeof(student_info));
}
```

Constructing a Linked List

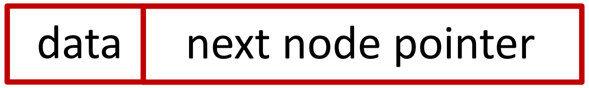
- **Problem with dynamic array**
  - Inserting even one item requires a lot of operations

- **Better design and algorithms?**
  - Linked list: items are linked by pointers

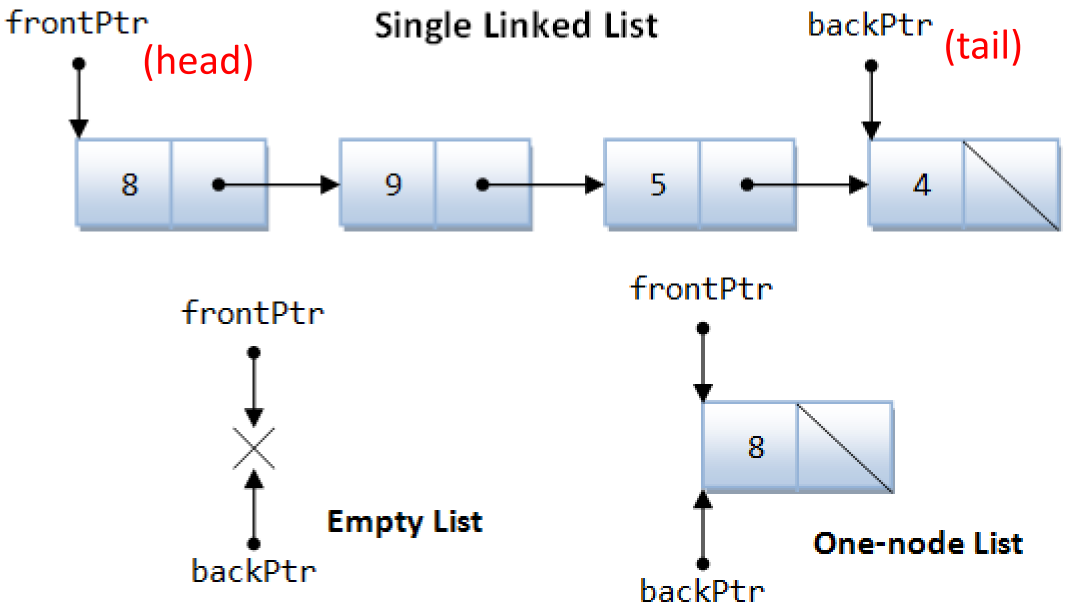


Constructing a Linked List

- **Types of linked lists**
  - Single linked list



List node

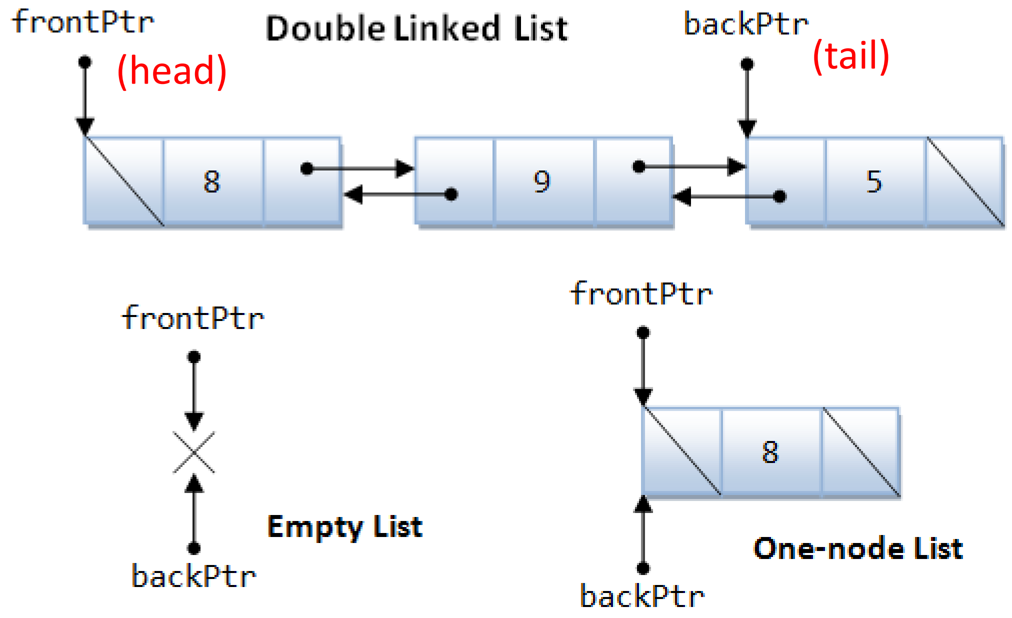


Constructing a Linked List

- **Types of linked lists**
  - Double linked list

data	next node pointer	next node pointer

List node



Constructing a Linked List

- **A node in a linked list**
  - Implementation with structures

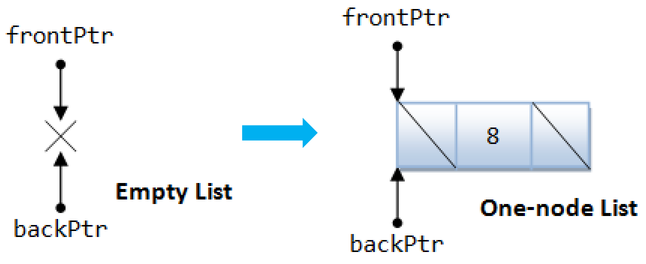
```
struct data_info{
   char name[20];
   int id;
   float score;
};
```

```
struct data_info_node {
    data_info data;
    data_info_node *prev;
    data_info_node *next;
};
```

Constructing a Linked List

- **Add one student information into a linked list**

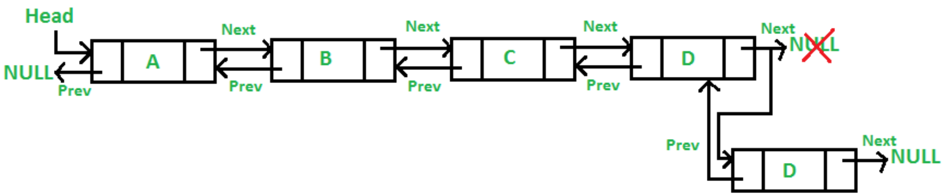
```
data_info_node *p_head = NULL;
data_info_node *p_tail = NULL;
//adding first student item

data_info_node *p_temp = (data_info_node *)malloc(sizeof(data_info_node));
printf("Inputting student info...\n");
scanf("student name: %s", p_temp->data.name);
scanf("student id: %d", & p_temp->data.id);
scanf("student score: %f", & p_temp->data.score);

p_head = p_temp;
p_tail = p_head;
p_head->prev=NULL;
p_head->next=NULL;
```



Constructing a Linked List

- **Keep adding student information**

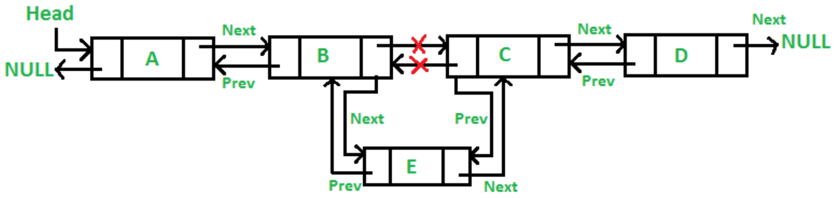
```
While(1) {
     char whether_to_add=‘y’;
     scanf(“Are you willing to add student info? (y/n):%c”, &whether_to_add);

     if(whether_to_add==‘n’）
          break;

     data_info_node *p_temp =
                   (data_info_node*)malloc(sizeof(data_info_node));
     printf(“Inputting student info...\n”);
     scanf(“student name: %s”, p_temp->data.name);
     scanf(“student id: %d”, & p_temp->data.id);
     scanf(“student score: %

     p_tail->next = p_temp;
     p_temp->prev = p_tail;
     p_temp->next = NULL;
     p_tail=p_temp;
}
```



Constructing a Linked List

- **Inserting student information**

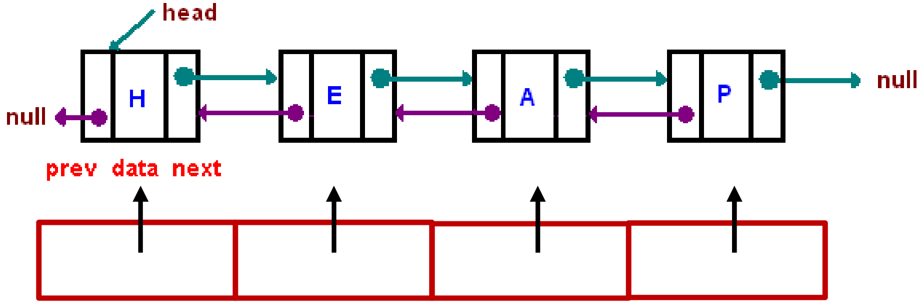
```
int insert_index=0;
scanf("Where do you want to insert: %d",&insert_index);

//locating the inserting point based on the insertion index
data_info_node * p_insert=p_head;
for(int i=0;i<insert_index;i++)
         p_insert=p_insert->next;

data_info_node *p_temp =
                   (data_info_node*)malloc(sizeof(data_info_node));
printf("Inputting student info...\n");
scanf("student name: %s", p_temp->data.name);
scanf("student id: %d", & p_temp->data.id);
scanf("student score: %f", & p_temp->data.score);

p_temp->prev=p_insert;
p_temp->next=p_insert->next;
p_temp->next->prev=p_temp;
p_insert->next = p_temp;
```



Combining Array and Linked List

- **Looking again the pros and cons of array and linked list**
  - Array
    - Pros: continuous, random access
    - Cons: difficult for dynamic insertion/deletion
  - List
    - Pros: Easy for dynamic insertion/deletion
    - Cons: hard to access randomly

- **Compromise**
  - An array of node pointers of linked list



An array of node address

Combining Array and Linked List

- **Constructing a hybrid structure**

```
//determining the number of students in the linked list
int student_num=0;
data_info_node * p_scan=p_head;
while(p_scan->next!=NULL){
        student_num++;
        p_scan=p_insert->next;
}

data_info_node **node_array=
             (data_info_node **)malloc(sizeof(data_info_node *)*student_num);
p_scan=p_head;
for(int i=0;i< student_num;i++){
           node_array[i]=p_scan;
           p_scan=p_insert->next;
}
```

Combining Array and Linked List

- **Constructing a hybrid structure**
  - Access list item
    - Usually when inserting/deletion is done
    - Use the node array

```
printf(“student name: %s”, node_array[i].data.name);
printf(“student id: %d”, &node_array[i].data.id);
printf(“student score: %f”, &node_array[i].data.score);
```

  - Insertion/deletion
    - When inserting/deleting student item(s), operate on the linked list until no insertion/deletion will be done
    - Update the node array

Files based on Structures

- **Writing to/reading from a binary file**
  - Writing to a binary file

```
FILE* file=fopen(“E:\\data”, “wb”);
if(file==NULL)
        return;
…
fclose(file);
```

  - Reading from a binary file

```
FILE* file=fopen(“E:\\data”, “rb”);
if(file==NULL)
        return;
…
fclose(file);
```

Files based on Structures

- **Writing a structure to a file**
  - Create and write to a binary file

```
struct student_info {
   char name[20];
   int id;
   float score;
};

student_info student;
... //some operations on student info

FILE* file=fopen(“E:\\data”, “wb”);
if(file==NULL)
     return;

if(fwrite(file,&student,sizeof(student_info))
          !=sizeof(student_info))
     printf(“error in writing the student info.\n”);

fclose(file);
```

Files based on Structures

- **Writing an array of structures to a file**

```
struct student_info {
   char name[20];
   int id;
   float score;
};

student_info *student_array=
     (student_info *)malloc(sizeof(student_info)*student_num);
... //some operations on student info array

FILE* file=fopen(“E:\\data”, “wb”);
if(file==NULL)
     return;
if(fwrite(file,student_array,sizeof(student_info)*student_num)
        !=sizeof(student_info) *student_num)
     printf(“error in writing the student info.\n”);
fclose(file);

free(student_array);
```

Files based on Structures

- **Writing an list of structures to a file**

```
data_info_node *p_scan=p_head;

FILE* file=fopen(“E:\\data”, “wb”);
if(file==NULL)
     return;

while(p_scan->next!=NULL)
{
     if(fwrite(file,&p_scan->data,sizeof(student_info))
         !=sizeof(student_info))
           printf(“error in writing the student info.\n”);
     p_scan=p_scan->next;
}

fclose(file);
```

Files based on Structures

- **Reading an array of structures from a file**

```
struct student_info {
   char name[20];
   int id;
   float score;
};

student_info *student_array=
     (student_info *)malloc(sizeof(student_info)*student_num);

FILE* file=fopen(“E:\\data”, “rb”);
if(file==NULL)
     return;
if(fread(file,student_array,sizeof(student_info)*student_num)
        !=sizeof(student_info) *student_num)
     printf(“error in reading the student info.\n”);
fclose(file);
...//some operations on the read student info
free(student_array);
```

Files based on Structures

- **Reading a list of structures from a file**

```
data_info_node *p_scan=p_tail;

FILE* file=fopen(“E:\\data”, “rb”);
if(file==NULL)
     return;
while(!feof(file))
{
     data_info_node *node=
            (data_info_node *)malloc(sizeof(data_info_node));
     if(fread(file,&node->data,sizeof(student_info))
         !=sizeof(student_info))
           printf(“error in reading the student info.\n”);
     p_scan->next=node;
     node->prev=p_scan;
     node->next=NULL;
     p_tail=node;
     p_scan=p_tail;
}
fclose(file);
```

File with a Header

- **What is a file header**
  - – A region at the beginning of each file
  - – Specify the information of the data stored

- **Implementing a file header**
  - – Usually be defined with structures

```
struct file_header {
    int record_num;
    char ower_name[32];
};
```

File with a Header

- **Writing a list of structures with a file header**

```
data_info_node *p_scan=p_head;

FILE* file=fopen(“E:\\data”, “wb”);
if(file==NULL)
     return;

//writing file header
int student_num=0;
data_info_node * p_scan=p_head;
while(p_scan->next!=NULL){
      student_num++;
      p_scan=p_insert->next;
}

file_header header;
header.record_num=student_num;
strcpy(header.ower_name,“Li Min”);

fwrite(file,&header,sizeof(file_header)); //write the file header
... //write the data trunk

fclose(file);
```

File with a Header

- **Reading a list of structures with a file header**

```
data_info_node *p_scan=p_head;

FILE* file=fopen(“E:\\data”, “rb”);
if(file==NULL)
     return;

//read the file header
file_header header;
if(fread(file,&header,sizeof(file_header))!=sizeof(file_header))
{
     printf(“unable to read file header\n”);
     return;
}

student_info *student_array=
     (student_info *)malloc(sizeof(student_info)*header.record_num);

fread(file, student_array, sizeof(student_info)*header.record_num);
... //use the data

free(student_array);
fclose(file);
```