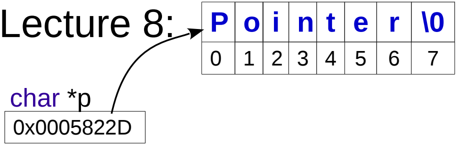
C Programming

Lecture 8:



P	o	i	n	t	e	r	\0
0	1	2	3	4	5	6	7

char *p

0x0005822D

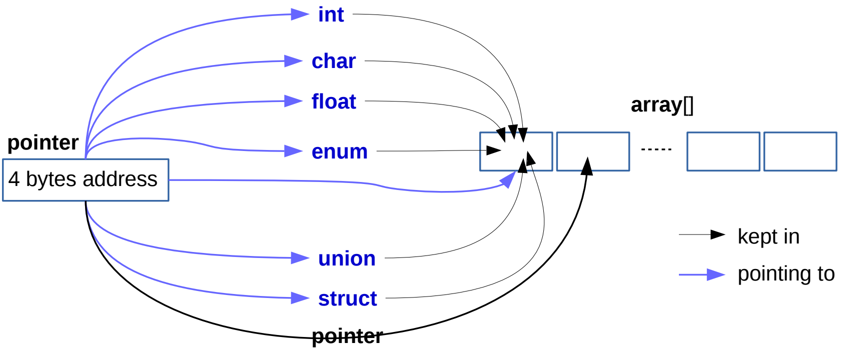
Lecturer: *Dr*. Wan-Lei Zhao

Autumn Semester 2022

Email: wlzhao@xmu.edu.cn,

Outline

1. Pointer to Primitive Type Variables
2. Pointer to Array
3. Pointer to struct Variables
4. Dynamic Memory Allocation
5. List Structure
6. Pointer to Function



- Pointer essentially is the address of a variable
- Any types of variable has an address
- Array has address too
- It is allowed to have pointer array (array of addresses)

Grammar for pointer definition

dataType *pointVariableName

- Pointer is a variable too
- A variable keeps address of other variable(s)
- “*” followed by variable name of the pointer

```
int main()
{
    int *pt; //pointer points to an integer variable
}
```

Pointer initialization

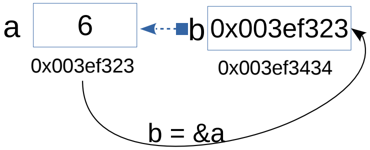
- Pointer is a variable too
- A variable keeps address of other variable(s)
- "*" followed by variable name of the pointer

```
#include <string.h>
#include <stdio.h>
int main()
{
    short *pt = NULL;//points to an integer variable
    float a = 3.1;
    float *fpt = &a;
    printf("Size␣of␣pt:␣%d\n", sizeof(pt));
    printf("Size␣of␣fpt:␣%d\n", sizeof(fpt));
    printf("Size␣of␣short:␣%d\n", sizeof(short));
}
```

- "&" is an operator (something new!)
- "&a" extracts the address of variable **a**
- Address of variable **a** (4 bytes number) is then assigned to "fpt"

Pointer in its nature

```
#include <string.h>
#include <stdio.h>
int main()
{
    int a = 6;
    int *b = &a;
    ....
```



- "&a" extracts the address of variable **a**
- Address of variable **a** (4 bytes number) is then assigned to "fpt"

Visit variable by its pointer (1)

```c
#include <string.h>
#include <stdio.h>
int main()
{
    short a = 4;
    short *pa= &a;
    float b = 3.1;
    float *pb = &b;
    printf("a = %d\n", a);
    printf("b = %f\n", b);
    printf("*pa = %d\n", *pa);
    printf("*pb = %f\n", *pb);
    printf("pa = %ld\n", pa);
    printf("pb = %ld\n", pb);
    return 0;
}
```

[Output:]

```
??
??
??
??
??
??
```

- "*pa" takes the value from the address kept by pa

Visit variable by its pointer (2)

```c
#include <string.h>
#include <stdio.h>
int main()
{
    short a = 4;
    short *pa= &a;
    float b = 3.1;
    float *pb = &b;
    printf("a = %d\n", a);
    printf("b = %f\n", b);
    printf("*pa = %d\n", *pa);
    printf("*pb = %f\n", *pb);
    printf("pa = %ld\n", pa);
    printf("pb = %ld\n", pb);
    return 0;
}
```

[Output:]

```
4
3.1
4
3.1
0439082323
0439082336
```

- "*pa" takes the value from the address kept by pa

Visit variable by its pointer (3)

```c
#include <string.h>
#include <stdio.h>
int main()
{
    float a = 4.5;
    float b = 3.1;
    float *p = &a;
    printf("p␣=␣%x\n", p);
    p = &b;
    printf("*p␣=␣%f\n", *p);
    printf("p␣=␣%x\n", p);
    *p = 7.2;
    p  = &a;
    *p = 5 .3;
    printf("a␣=␣%f\n", a);
    printf("b␣=␣%f\n", b);
    return 0;
}
```

[Output:]

```
?
?
?
?
?
```

- "*pa" takes the value from the address kept by pa

Revisit: swap values of *a* and *b* (1)

```c
#include <stdio.h>
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
    return ;
}
int main()
{
   int a = 3;
   int b = 5;
   printf("a=%d,b=%d\n",a,b);
   swap(a, b);
   printf("a=%d,b=%d\n",a,b);
   return 0;
}
```

```c
#include <stdio.h>
int a, b;
void swap()
{
   int tmp = a;
   a = b;
   b = tmp;
   return ;
}
int main()
{
   a = 3;
   b = 5;
   printf("a=%d,b=%d\n",a,b);
   swap(a, b);
   printf("a=%d,b=%d\n",a,b);
   return 0;
}
```

Revisit: swap values of *a* and *b* (2)

```c
#include <stdio.h>
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
    return ;
}
int main()
{
  int a = 3;
  int b = 5;
  printf("a=%d,b=%d\n",a,b);
 swap(a, b);
  printf("a=%d,b=%d\n",a,b);
  return 0;
}
```

```c
#include <stdio.h>
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
    return ;
}
int main()
{
  int a = 3;
  int b = 5;
  printf("a=%d,b=%d\n",a,b);
 swap(&a, &b);
  printf("a=%d,b=%d\n",a,b);
  return 0;
}
```

Revisit: swap values of *a* and *b* (3)

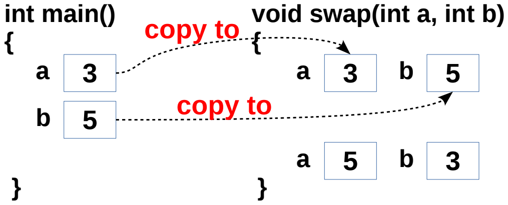


Figure: What happens for swap(int a, int b).

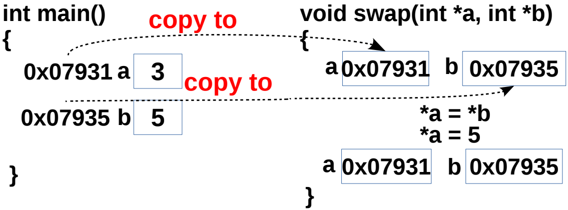


Figure: What happens for swap(int *a, int *b).

Revisit: swap values of *a* and *b* (4)

```c
#include <stdio.h>
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
    return ;
}
int main()
{
  int a = 3;
  int b = 5;
  printf("a=%d,b=%d\n",a,b);
 swap(&a, &b);
  printf("a=%d,b=%d\n",a,b);
  return 0;
}
```

```c
#include <stdio.h>
void swap(adr a, adr b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
    return ;
}
int main()
{
  int a = 3;
  int b = 5;
  printf("a=%d,b=%d\n",a,b);
 swap(&a, &b);
  printf("a=%d,b=%d\n",a,b);
  return 0;
}
```

- Given adr is an address type

Summary over Pointer to Variables (1)

- Pointer is a variable or constant
- It keeps the address of a variable
- One is allowed to do operation on a variable by its address

```c
#include <stdio.h>
int main()
{
    int a = 3, *p;
    int b = 1;
    p = &a;
    printf("a=%d\n”, *p);
    p = &b;
    printf("b=%d\n”, *p);
}
```

```c
void incr(int *a)
{
    *a = *a + 1;
}
int main()
{
 int a = 4, *b = &a;
 printf(”%d\n”, *b);
 incr(&a);
 printf(”%d\n”, a);
 printf(”%d\n”, *b);
 return 0;
}
```

Summary over Pointer to Variables (2)

- Pointer is a variable or constant
- It keeps the address of a variable
- One is allowed to do operation on a variable by its address

```c
#include <stdio.h>
int main()
{
    int a = 3, *p;
    int b = 1;
    *p = a;
    p  = b;
    p  = &c;
}
```

```c
#include <stdio.h>
int main()
{
    int a = 3, *p;
    int b = 1;
    float c = 2.2;
    p   = &a;
    printf("%d", *p);
    *p  = b;
    printf("%d", *p);
    printf("%d", a);
}
```

Summary over Pointer to Variables (3)

```c
void incr(int *a)
{
    *a = *a + 1;
}
int main()
{
  int a = 4, *b = &a;
  printf("%d\n", *b);
  incr(&a);
  printf("%d\n", a);
  printf("%d\n", *b);
  return 0;
}
```

```c
void incr(int *a)
{
    a = a + 4;
}
int main()
{
  int a = 4, *b = &a;
  printf("%d\n", *b);
  incr(&a);
  printf("%d\n", a);
  printf("%d\n", *b);
  return 0;
}
```

- 'incr(int* a)' on the right, increases the address number of **a**
- It points to another memory cell
- **a** inside 'incr(int *a)' is a local variable
- It has no effect on input variable

Explained

```c
void incr(adr a)
{
    *a = *a + 1;
}
int main()
{
 int a = 4, *b = &a;
 printf("%d\n", *b);
 incr(&a);
 printf("%d\n", a);
 printf("%d\n", *b);
 return 0;
}
```

```c
void incr(adr a)
{
    a = a + 4;
}
int main()
{
 int a = 4, *b = &a;
 printf("%d\n", *b);
 incr(&a);
 printf("%d\n", a);
 printf("%d\n", *b);
 return 0;
}
```

- Given adr is an address type
- Keep the principle that parameter "transfer by value" in C
- **a** inside 'incr(adr a)' is a local variable
- It has no effect on input variable

A Revisit about "scanf(·,·)"

```c
int main()
{
  int a = 0;
  printf("Input␣value␣for␣a:␣");
  scanf("%d", &a); //<-- pay attention to here
  return 0;
}
```

- Now we should be clear why we put "&" before **a**
- By this way, we tell **scanf**(·) to put the user input value to which memory address
- Is it possible if we do something as following?

```c
int main()
{
  int a = 0;
  printf("Input␣value␣for␣a:␣");
  scanf("%d", a); //<--ask yourself whether this is valid??
  return 0;
}
```

Outline

1. Pointer to Primitive Type Variables
2. Pointer to Array
3. Pointer to struct Variables
4. Dynamic Memory Allocation
5. List Structure
6. Pointer to Function

An Overview: Pointer to Array (1)

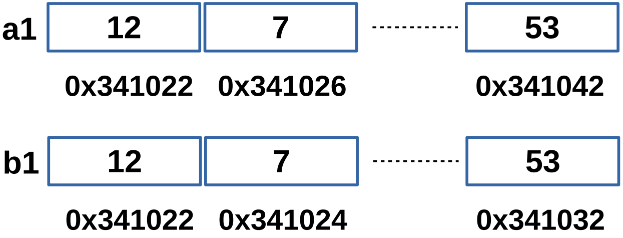
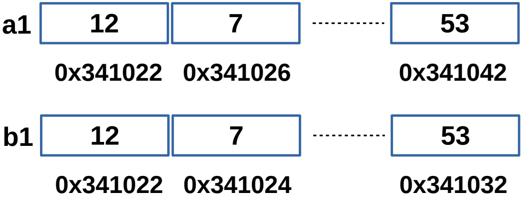


Figure: Two typical arrays of int type.

- Array is a continuous memory block
- It has a starting address
- It has a length
- It has a name

An Overview: Pointer to Array (2)



a1 | 12 | 7 | .... | 53

0x341022 0x341026 0x341042

b1 | 12 | 7 | .... | 53

0x341022 0x341024 0x341032

Figure: Two typical arrays of int type

- Unlike primitive type variable
- The name of an array is also the starting address of an array

```
int main()
{
    int a[5]={4, 5, 7, 11, 13, 17};
    int *p = a;
    p = &a[0];
    return 0;
}
```

Definition and initializaiton (1)

```
int *p;
int a1[10];
p = a1;
p = &a1[0];
```

- Definition of array pointer is the same as variable pointer
- Above two ways are valid
- '**p**' keeps the address of starting address of **a1**
- Now think about what "p = p+2' means here??

Definition and initializaiton (2)

```
#include <stdio.h>
int main()
{
    int a1[4] = {31, 1, 11, 4};
    int i = 0, *p = a1;
    for(i=0;i<4; i++,p++)
    {
        printf("%d␣", *p);
    }
    return 0;
}
```

- ‘**p**’ visits element in array a1 one by one
- ‘***p**’ takes the value according to the address in ‘**p**’

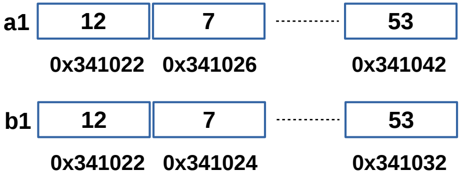
Definition and initializaiton (3)

```c
#include <stdio.h>
int main()
{
    int a1[4]={31, 1, 11, 4};
    int i = 0, *p = a1;
    for(i=0;i<4; i++,p++)
    {
        printf("%d␣", *p);
    }
    return 0;
}
```

```c
#include <stdio.h>
int main()
{
    int a1[4]={31, 1, 11, 4};
    int i = 0, *p = a1;
    for(i = 0; i < 4; i++)
    {
        printf("%d␣", a1[i]);
    }
    return 0;
}
```

- ‘**p**’ visits element in array a1 one by one
- ‘***p**’ takes the value according to the address in ‘**p**’

Definition and initializaiton (3)



a1 12 7 53

0x341022 0x341026 0x341042

b1 12 7 53

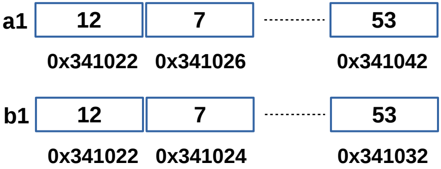
0x341022 0x341024 0x341032

Figure: Two typical arrays of int type

```c
int main()
{
    int a1[6] = {12, 7, 7, 11, 13, 53};
    short b1[6] = {12, 7, 7, 11, 13, 53};
    int *pa = &a1;
    short *pb = &b1;
    return 0;
}
```

- Like pointer to variable, different types of array need different types of pointer

Operations on Pointer of Array (1)



a1 12 7 53
0x341022 0x341026 0x341042
b1 12 7 53
0x341022 0x341024 0x341032

Figure: Two typical arrays of int type

```
int main()
{
   int a1[6]={12, 7, 17, 11, 13, 53};
   short b1[6]={12, 7, 17, 11, 13, 53};
   int *pa = &a1;
   short *pb = &b1;
   pa++; pb++;
   printf("%d\n", *pa);
   printf("%d\n", *pb);
   return 0;
}
```

[Output]
?
?

Operations on Pointer of Array (2)

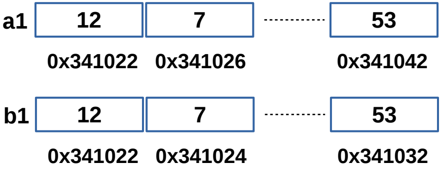


Figure: Two typical arrays of int type

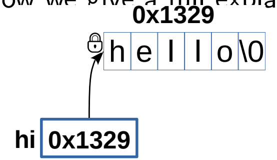
```c
int main()
{
    int a1[5] = {12, 7, 17, 11, 13, 53};
    short b1[5] = {12, 7, 17, 11, 13, 53};
    int *pa = &a1;
    short *pb = &b1;
    pa++; pb++;
    printf("%d\n", *pa);
    printf("%d\n", *pb);
    return 0;
}
```

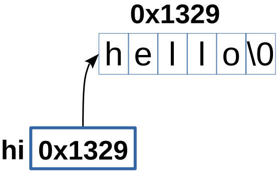
[Output]

```
7
7
```

Pointer to String

- Think about following example
- We saw it many times
- Now we give a full explanation over it





```c
#include <stdio.h>
int main()
{
    char *hi = "hello";
    hi[1] = 'a'; //<--- illegal
    printf("%s\n", hi);
    return 0;
}
```

```c
#include <stdio.h>
int main()
{
    char hi[] = "hello";
    hi[1] = 'a'; //<---legal
    printf("%s\n", hi);
    return 0;
}
```

Array of chars, String and Pointer of String

- Since pointer points to the first address of an array
- "str1" is defined as constant array of chars, and pointed by pointer str
- Definitions about "str2" and "str3" are equivalent
- Definition about "str4" is different from above three

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char *str1 = "hello"; //<--str1[0] = 'a' will be illegal
    char str2[10] = "hello";
    char str3[10] = {'h', 'e' , 'l' ,'l', 'o', '\0'};
    char str4[10] = {'h', 'e' , 'l' ,'l', 'o'}; //<--it is different
    printf("%s\n", str1);
    printf("%s\n", str2);
    printf("%s\n", str3);
    printf("%s\n", str4);
    return 0;
}
```

Example of Pointer to Array (1)

- Given **str1**=”abserds” and **str2**=”xxxxx”
- You are required to copy the contents of one string to another

Example of Pointer to Array (2)

- Given **str1**=”absERds” and **str2**=”xxxxx”
- You are required to copy the contents of one string to another
  1. Define pointers (p1 and p2) for **str1** and **str2**
  2. Pointing to the start of each
  3. Assign value of p1 to p2
  4. Repeat **Step 3** until the end of **str1**
  5. Assign '\0' to the end of **str2**

Example of Pointer to Array (3)

```c
#include <stdio.h>
int main()
{
  char *str1="hello world!";
  char str2[16];
  char *p1 = str1;
  char *p2 = str2;
  while(p1 != '\0')
  {
      *p2 = *p1;
      p1++; p2++;
  }
  printf("%s\n", str1);
  printf("%s\n", str2);
  return 0;
}
```

- There is a bug, please tell me:)

Example of Pointer to Array (4)

```c
#include <stdio.h>
int main()
{
   char *str1="hello world!";
   char str2[16];
   char *p1 = str1;
   char *p2 = str2;
   while(*p1 != '\0')
   {
        *p2 = *p1;
        p1++; p2++;
   }
   *p2='\0'; //<--indicate the end of the string
   printf("%s\n", str1);
   printf("%s\n", str2);
   return 0;
}
```

- Be careful all the time

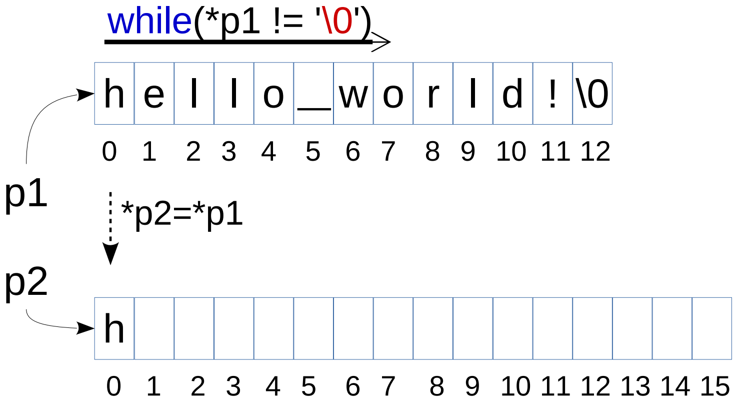
Example of Pointer to Array (5)

```c
#include <stdio.h>
void strCopy(char *p1, char *p2)
{
  while(*p1 != '\0')
  {
      *p2 = *p1;
      p1++; p2++;
  }
  *p2='\0';
}

int main()
{
  char *str1 = "hello world!";
  char str2[16];
  strCopy(?, ?);
  printf("%s\n", str1);
  printf("%s\n", str2);
  return 0;
}
```

Example of Pointer to Array (6)

```c
#include <stdio.h>
void strCopy(char *p1, char *p2)
{
  while(*p1 != '\0')
  {
      *p2 = *p1;
      p1++; p2++;
  }
  *p2='\0';
}

int main()
{
  char *str1="hello␣world!";
  char str2[16];
  strCopy(str1, str2);
  printf("%s\n", str1);
  printf("%s\n", str2);
  return 0;
}
```

Example of Pointer to Array (7)



while(*p1 != '\0')

h	e	l	l	o	_	w	o	r	l	d	!	\0
0	1	2	3	4	5	6	7	8	9	10	11	12

p1

*p2=*p1

p2

h															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- The while loop stop at '\0'
- '\0' will not be copied in the loop

Popular functions for string operation (1)

1. **strlen**(str1); length of str1, ‘\0’ is not counted
2. **strcpy**(str1, str2); copy str2 to str1
3. **strcmp**(str1, str2); compare two strings
4. **strcat**(str1, str2); concantenate two strings
5. **strncpy**(str1, str2, n); copy first n chars of str2 to str1

Popular functions for string operation (2)

2. **strcpy**(str1, str2); copy str2 to str1
4. **strcat**(str1, str2); concantenate two strings

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char *str1="hello", *str2 = "world";
    char hi[32];
    strcpy(hi, str1);
    strcat(hi, "␣");
    strcat(hi, str2);
    printf("%s\n", hi);
    return 0;
}
```

Popular functions for string operation (3)

3. **strcmp**(str1, str2); compare two strings

```
#include <stdio.h>
#include <string.h>
int main()
{
    char *str1="hello", *str2 = "hi", *str3="hello";
    if(strcmp(str1, str2) == -1)
    {
        printf("str1␣<␣str2!\n");
    }else if(strcmp(str1, str2) == 1) {
        printf("str1␣>␣str2!\n");
    }
    if(strcmp(str1, str3) == 0)
    {
        printf("They␣are␣equal!\n");
    }else{
        printf("They␣are␣inequal!\n");
    }
    return 0;
}
```

Outline

1. Pointer to Primitive Type Variables
2. Pointer to Array
3. Pointer to struct Variables
4. Dynamic Memory Allocation
5. List Structure
6. Pointer to Function

Pointer to struct Type Variable (1)

- The declaration of pointer to struct type is similar as pointer to primitive type and array

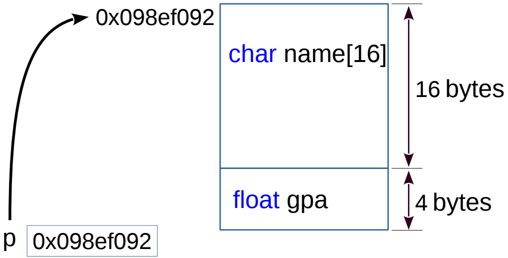
```c
struct STD {
  char name[16];
  float gpa;
};
int main()
{
    struct STD std1 = {"Peter", 3.8};
    struct STD *p = &std1;
    printf("Name:␣%s\n", (*p).name);
    printf("GPA:␣%f\n", (*p).gpa);
    return 0;
}
```

Pointer to struct Type Variable (1)

- The declaration of pointer to struct type is similar as pointer to primitive type and array

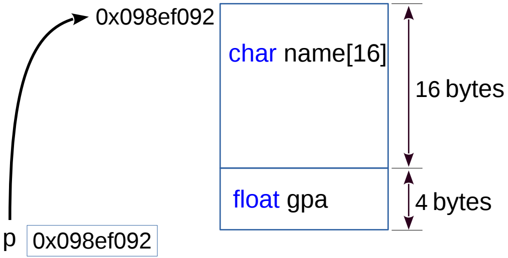
```c
struct STD {
  char name[16];
  float gpa;
};
int main()
{
    struct STD std1 = {"Peter", 3.8};
    struct STD *p = &std1;
    printf("Name:␣%s\n", (*p).name);
    printf("GPA:␣%f\n", (*p).gpa);
    return 0;
}
```

Pointer to struct Type Variable (2): explained



- Pointer keeps the starting address of the struct type variable
- sizeof(p) = ?

Pointer to struct Type Variable (3): explained



- Pointer keeps the starting address of the struct type variable
- sizeof(p) = ?
- Notice that the address is only 4 bytes (32 bits system)

Pointer to struct Type Variable

```c
struct STD {
  char name[16];
  float gpa;
};
typedef struct STD STDT;
int main()
{
 STDT std1 = {"Peter", 3.8};
 struct STD *p = &std1;
 printf("%s\n", (*p).name);
 printf("%f\n", (*p).gpa);
 return 0;
}
```

```c
struct STD {
  char name[16];
  float gpa;
};
typedef struct STD STDT;
int main()
{
 STDT std1 = {"Peter", 3.8};
 struct STD *p = &std1;
 printf("%s\n", p->name);
 printf("%f\n", p->gpa);
 return 0;
}
```

- typedef denotes “struct STD” as “STDT”
- “p->” is equivalent to “(*p).”

Comparison Study over Pointers

```c
#include <stdio.h>
struct STD {
  char name[16];
  float gpa;
};
int main()
{
    struct STD std1 = {"Peter", 3.8};
    struct STD *p = &std1;
    int *q;
    char *r;
    printf("size␣of␣STD:␣%d\n", sizeof(struct STD));
    printf("size␣of␣p:␣%d\n", sizeof(p));
    printf("size␣of␣q:␣%d\n", sizeof(q));
    printf("size␣of␣r:␣%d\n", sizeof(r));
    return 0;
}
```

- The size is the same for different kinds of pointers
- Why??

Outline

1. Pointer to Primitive Type Variables
2. Pointer to Array
3. Pointer to struct Variables
4. Dynamic Memory Allocation
5. List Structure
6. Pointer to Function

Static and Dynamic Memory Allocation (1)

- Recall what the variables we learned so far
  1. Primitive type variables
  2. Primitive type arrays
  3. Composite type variables
  4. Composite type arrays

```c
struct STD {
  char name[16];
  float gpa;
};
typedef STD STDT;
int main()
{
    int a, a1[10];
    STDT b, b1[10];
}
```

- The memory cells for a, a1, b and b1 are allocated when your code is loaded into memory
- It is done before the code is executed

Static and Dynamic Memory Allocation (2)

- In some cases, we are not sure how long is the array we need before run it
- We have two options for this case
  1. Apply for a very long array, i.e., 65,536
  2. Apply the memory cells in the runtime
- The second way is called dynamic memory allocation

Dynamic Memory Allocation: grammar (1)

```
int *p = (int*)malloc(sizeof(int)*10);
```

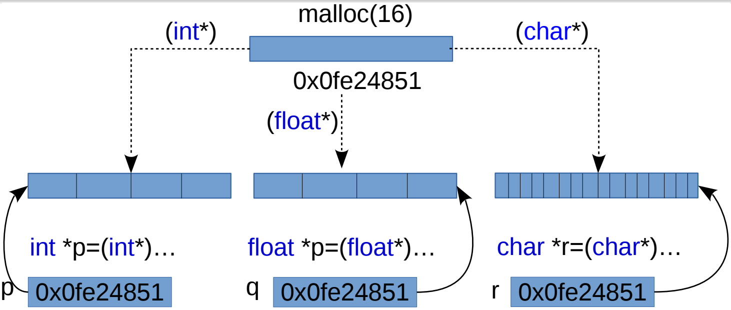
1. Apply a block of memory sized of 10*sizeof(int)=??
2. Function "**malloc**(·)" returns the starting address of this memory
3. Convert this starting address to an int type pointer
4. Assign this starting address to **p**

Dynamic Memory Allocation: grammar (2)

```
int *p = (int*)malloc(sizeof(int)*10);
```

1. Function "**malloc**(·)" sends the applicaiton to OS
2. When the application is approved, a block of memory is returned
3. OS extracts memory from **Heap**
4. Once it is allocated, you can operate it as an array

Dynamic Memory Allocation: explained



```
#include <stdlib.h>
int main()
{
    void *x = malloc(16);
    int *p = (int*)x;
    float *q = (float*)x;
    char *r = (char*)x;
}
```

- We just show it is possible
- It is NOT suggested in practice

Dynamic Memory Allocation: example

```c
#include <stdlib.h>
int main()
{
    int i = 0, *a1 = (int*)malloc(5*sizeof(int));
    for(i = 0; i < 5; i++)
    {
        a1[i] = i+1;
    }
    free(a1);//<--release the memory pointing by a1
    return 0;
}
```

1. Function “**malloc**(·)” returns the starting address of this block of memory
2. Once it is allocated, you can operate it as an array
3. Always remember to release it by calling free(·)

Dynamic Memory Allocation: memory leakage (1)

- Different from static memory allocation
- You are required to release the dynamically allocated memory on your own
- If you fail to do that, memory leakage occurs (90%) C bugs arise from this

```c
#include <stdlib.h>
int main()
{
    int i = 0, *a1 = (int*)malloc(5*sizeof(int));
    for(i = 0; i < 5; i++)
    {
        a1[i] = i+1;
    }
    free(a1);//<-- very important here
    return 0;
}
```

Dynamic Memory Allocation: memory leakage (2)

```c
1  #include <stdlib.h>
2  int main()
3  {
4      int i = 0, *a1 = (int*)malloc(5*sizeof(int));
5      for(i = 0; i < 5; i++)
6      {
7          a1[i] = i+1;
8      }
9      free(a1);
10     a1[2] = 3;//<---- illegal memory access
11     return 0;
12 }
```

- You are not allowed to use memory that has been released
- Above code (line 10) causes illegal memory access exception

Dynamic Memory Allocation: memory leakage (3)

```c
1  #include <stdlib.h>
2  int main()
3  {
4      int i = 0, *a1 = (int*)malloc(5*sizeof(int));
5      for(i = 0; i < 5; i++)
6      {
7          a1[i] = i+1;
8      }
9      a1 = (int*)malloc(15*sizeof(int));//<-something wrong here
10     free(a1);
11     return 0;
12 }
```

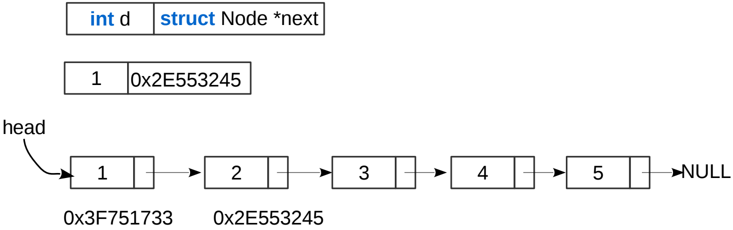
,

- You are not allowed to use memory that has been released
- We lose the pointer to one block of memory (at line 9)
- Memory leaks (ghost memory cells)

Outline

1. Pointer to Primitive Type Variables
2. Pointer to Array
3. Pointer to struct Variables
4. Dynamic Memory Allocation
5. List Structure
6. Pointer to Function

Overview of List Structure



int d | struct Node *next

1 | 0x2E553245

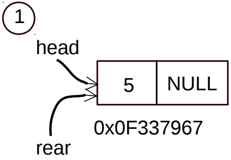
head

1 → 2 → 3 → 4 → 5 → NULL

0x3F751733 0x2E553245

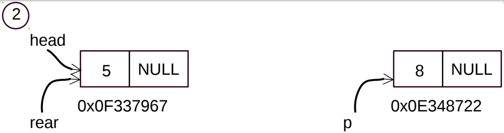
```
struct Node {
  int a;
  struct Node *next;
};
typedef Node TNode;
```

Build List—Step 1



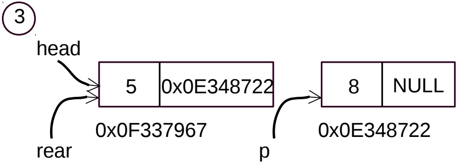
```
struct Node {
  int a;
  struct Node *next;
};
typedef Node TNode;
int main()
{
  TNode *head = NULL, *rear = NULL;
  TNode *p = (TNode*)malloc(sizeof(TNode));
  p->a = 5; p->next = NULL;
  head = p; rear = p;
}
```

Build List—Step 2



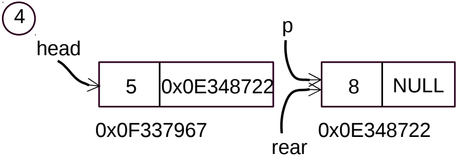
```c
struct Node {
  int a;
  struct Node *next;
};
typedef Node TNode;
TNode *buidList()
{
  TNode *head = NULL, *rear = NULL;
  TNode *p = (TNode*)malloc(sizeof(TNode));
  p->a = 5; p->next = NULL;
  head = p; rear = p;
  p = (TNode*)malloc(sizeof(TNode));
  return head;
}
```

Build List—Step 3



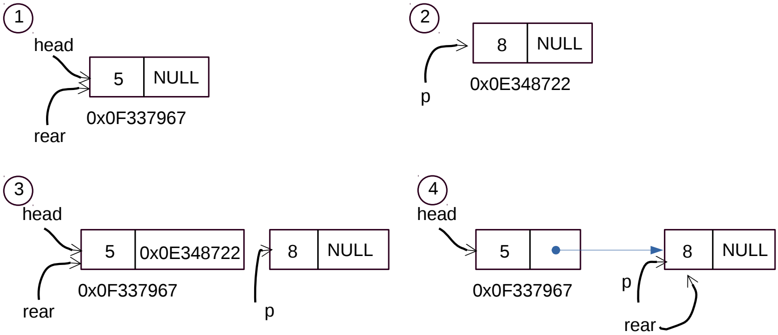
```
TNode *buidList()
{
  TNode *head = NULL, *rear = NULL;
  TNode *p = (TNode*)malloc(sizeof(TNode));
  p->a = 5; p->next = NULL;
  head = p; rear = p;
  p = (TNode*)malloc(sizeof(TNode));
  rear->next = p;
  return head;
}
```

Build List—Step 4

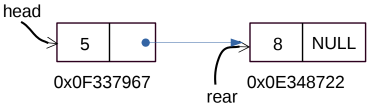


```
TNode *buidList()
{
  TNode *head = NULL, *rear = NULL;
  TNode *p = (TNode*)malloc(sizeof(TNode));
  p->a = 5; p->next = NULL;
  head = p; rear = p;
  p = (TNode*)malloc(sizeof(TNode));
  rear->next = p;
  rear = p;
  return head;
}
```

Build List—Summary

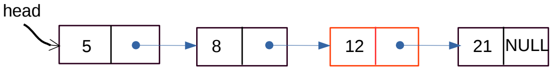


Print List



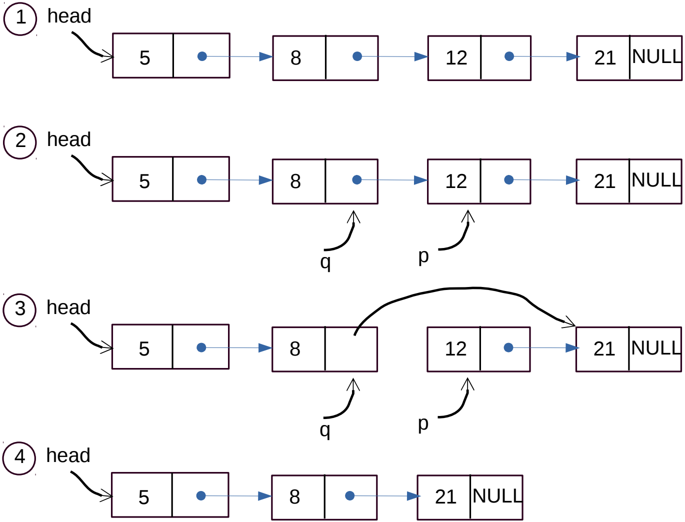
```c
int printList(TNode *head)
{
  TNode *p = head;
  int i = 0;
  while(p != NULL)
  {
      printf("%3d\n", p->a);
      p = p->next;
      i++;
  }
  return i;
}
```

Delete Node from List

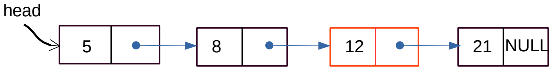


- We want to delete the node in which a equals to 12

Delete Node from List–Steps



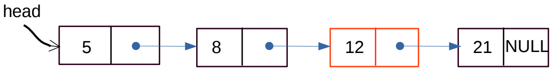
Delete Node from List—Procedure



- We want to delete the node in which a equals to 12

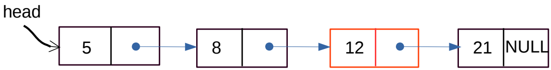
1. Find the node, whose a equals to 12
2. Given it is p, the node before it is q
    1. q− >next = p− >next;
    2. p− >next = NULL;
    3. free(p);

Delete Node from List—Codes



```
void deleteNode(int val, TNode *head)
{
    TNode *p = head, *q = head;
    //filling the codes here
}
```

Delete Node from List—The answer



```c
void deleteNode(int val, TNode *head)
{
    TNode *p = head, *q = head;
    while(p != NUL && p->a != val)
    {
        q = p;
        p = p->next;
    }
    if(p != NULL && p->a == val)
    {
        q->next = p->next;
        p->next = NULL;
        free(p);
    }
}
```

Why condition “p ! = NUL” first???

What are the differences between Array and List

	Array	List
Structure	linear	linear
Memory	continous block	chain of blocks
Visit	subscript	linear scan
Insert/delete	element shifting	direct operation

Outline

1. Pointer to Primitive Type Variables
2. Pointer to Array
3. Pointer to struct Variables
4. Dynamic Memory Allocation
5. List Structure
6. Pointer to Function

An Overview: Motivation (1)

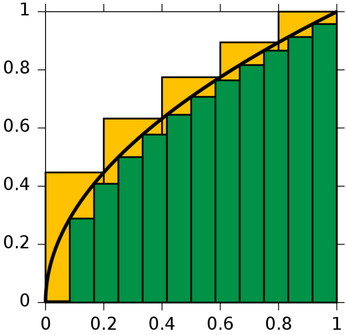


Figure: Numerical integral of $\sqrt{x}$

- Given two functions to perform the numerical integral
- $f(x) = \sqrt{x}$, $g(x) = cos(x)$
- $\int_a^b f(x)dx = ?$, $\int_a^b g(x)dx = ?$

An Overview: Motivation (2)

- Define dx=0.05, given a and b
- We can calculate integral of $\sqrt{x}$ when $x \in [a, b]$

```c
#include <math.h>
#include <stdio.h>
float intSqrt(float dx, float a, float b){
   float s = 0, x = a;
   while(x < b){
      s += sqrt(x)*dx;
      x += dx;
   }
   return s;
}
float intCos(float dx, float a, float b){
   float s = 0, x = a;
   while(x < b){
      s += cos(x)*dx;
      x += dx;
   }
   return s;
}
```

An Overview: Motivation (3)

- Define dx=0.05, given a and b
- We can calculate integral of $\sqrt{x}$ when $x \in [a, b]$

```c
void main()
{
    float a = 1.0, b = 5.0, dx = 0.05, s = 0;
    char funcName[8] = "";
    scanf("%s", &funcName);
    if(strcmp(funcName, "sqrt") == 0){
        s = intSqrt(dx, a, b);
    }else if(strcmp(funcName, "sin") == 0){
        s = intSin(dx, a, b);
    }else if(strcmp(funcName, "cos") == 0){
        s = intCos(dx, a, b);
    }
    printf("Integral is: %f\n", s);
}
```

An Overview: Motivation (4)

- Define dx=0.05, given a and b
- We can calculate integral of $\sqrt{x}$ $x \in [a, b]$

```
void main()
{
  float (*fun_ptr)(float dx, float a, float b);
  float a = 1.0, b = 5.0, dx = 0.05, s = 0;
  char funcName[8] = "";
  scanf("%s", &funcName);
  if(strcmp(funcName, "sqrt") == 0){
      func_ptr = &intSqrt;
  }else if(strcmp(funcName, "sin") == 0){
      func_ptr = &intSin;
  }else if(strcmp(funcName, "cos") == 0){
      func_ptr = &intCos;
  }
  s = (*func_ptr)(dx, a, b);
  printf("Integral␣is:␣%f\n", s);
}
```

Function Pointer: the declaration (1)

type0 (*function_pointer_name)(type1 p1, type2 p2);

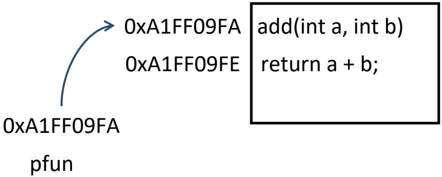
- Given a function in the same form

type0 fun1(type1 p1, type2 p2);

*function_pointer_name = &fun1;

```c
#include <stdio.h>
int add(int a, int b){
    return a+b;
}
int main(){
    int (*pfun)(int a, int b) = NULL;
    int a = 5, b = 8, r = 0;
    pfun = &add;
    r = pfun(a, b);
    printf("r␣=␣%d\n", r);
    return 0;
 }
```

Function Pointer: the declaration (2)



```c
#include <stdio.h>

int add(int a, int b){
    return a+b;
}
int main(){
    int (*pfun)(int a, int b) = NULL;
    int a = 5, b = 8, r = 0;
    pfun = &add;
    r = pfun(a, b);
    printf("r = %d\n", r);
    return 0;
 }
```

Function Pointer: the declaration (3)

```c
#include <stdio.h>

int add(int a, int b){
    return a+b;
}
int main()
{
    int (*pfun)(int a, int b) = NULL;
    int a = 5, b = 8, r = 0;
    pfun = &add;
    r = pfun(a, b);
    printf("size␣of␣pointer:␣%d\n", sizeof(pfun));
    printf("r␣=␣%d\n", r);
    return 0;
 }
```