

C Programming Essentials

Unit 3: Basic Data Structures

CHAPTER 8: C POINTERS AND ARRAYS

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Pointers and addresses



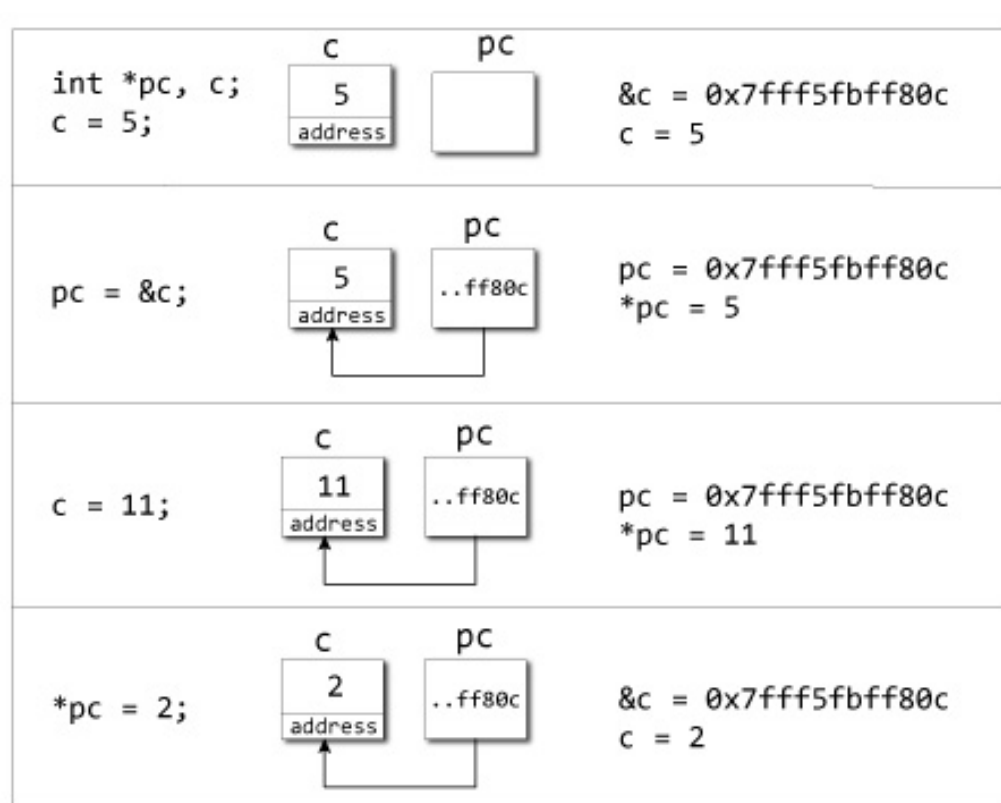
Memory addresses

- Memory is divided up into one byte pieces individually addressed.
 - minimum data you can request from the memory is 1 byte
 - Each byte has an address.
- For a 32 bit processor, addressable memory is 2^{32} bytes. To uniquely identify each of the accessible byte you need $\log_2 2^{32} = 32$ bits
- Depends on Compiler and Computer System Bus. Memory allocation may not be on a byte by byte basis. 32-bit system may allocate memory 32-bit (4 bytes) at a time. Some holes in memory allocation may happen but programmers may not observe it.

char	address
0A	0x00001234
23	0x00001235
6C	0x00001236
1D	0x00001237
'W'	0x00001238
'o'	0x00001239
'w'	0x0000123A
'\0'	0x0000123B
.	.
.	.
.	.
	0x24680975
	0x24680976
	0x24680977
	0x24680978

`*, &, ->, [], +, -, sizeof()`

Operators for Calculation of Address and Access of Object's Value



- If `c` is a variable name, we may use `pc` or `cp` as the variable name for pointer of `c` (`pc`) or `c`'s pointer (`cp`).
- In the declaration,
 - `int *pc;` // It should be read as
// **body of pc** (`*pc`) is an int, then
// `pc` is a pointer to int
- `&c` should be read as **address of c** variable.
- So, `pc = &c` means assigning `int c`'s address to the pointer `pc`.
- Pointer is a variable that hold the address of some other variable.
- `*pc` is read as body (object) of the pointer `pc`. It means the variable that the pointer `pc` pointing to.
- `*pc` is a int variable if it is on the left-hand-side. `*pc` is the value of variable if it is on the right-hand-side.
- `*pc` is used exactly like `c`.



Address of a variable

- Each variable that is declared is stored in memory. Since memory is indexed each variable has an address.
- C allows programmers to access the address of the variables.
- Unary operator & (read as 'address of') gives the address of the operand variable.
- Since constants, expressions do not have permanent storage, they do not have address.
- `&1.0`, `&(d+6)` are illegal.

```
#include <stdio.h>
#define line "\t-----\n"

main()
{
    int i = 0;
    char c = 'a';
    short s = 1;
    long l = 2;
    float f = 3.0;
    double d = 4.0;

    printf("Variable Table\n");
    printf(line);
    printf("\tVar\t\tAddress\t\tValue\n");
    printf(line);
    printf("\t%s\t\t%p\t%d\n", "i", &i, i);
    printf("\t%s\t\t%p\t%c\n", "c", &c, c);
    printf("\t%s\t\t%p\t%d\n", "s", &s, s);
    printf("\t%s\t\t%p\t%d\n", "l", &l, l);
    printf("\t%s\t\t%p\t%f\n", "f", &f, f);
    printf("\t%s\t\t%p\t%f\n", "d", &d, d);
    printf(line);
}
```



Address of a Variable

This program was compiled by gcc on Windows machine.

The output of this program:

Variable Table		
Var	Address	Value
i	0060FF0C	0
c	0060FF0B	a
s	0060FF08	1
l	0060FF04	2
f	0060FF00	3.000000
d	0060FEF8	4.000000

```
#include <stdio.h>
#define line "\t-----\n"

main()
{
    int i = 0;
    char c = 'a';
    short s = 1;
    long l = 2;
    float f = 3.0;
    double d = 4.0;

    printf("Variable Table\n");
    printf(line);
    printf("\tVar\t\tAddress\t\tValue\n");
    printf(line);
    printf("\t%s\t\t%p\t%d\n", "i", &i, i);
    printf("\t%s\t\t%p\t%c\n", "c", &c, c);
    printf("\t%s\t\t%p\t%d\n", "s", &s, s);
    printf("\t%s\t\t%p\t%d\n", "l", &l, l);
    printf("\t%s\t\t%p\t%f\n", "f", &f, f);
    printf("\t%s\t\t%p\t%f\n", "d", &d, d);
    printf(line);
}
```



Pointer to a Variable

- If variables have addresses, can these be stored in the variables and manipulated with?
- These addresses have a new data-types (derived data-types) called *pointers*.
- Pointers are declared as
`data-type *var-name;`
- In this example one pointer variable has been declared as `p` and it can store addresses of any integer variables.

```
#include <stdio.h>
#define line "\t-----\n"

main()
{
    int i = 0;
    int j = 1;
    int k = 2;
    float f = 3.0;
    int *p = 0;

    p = &k;
    printf("p = %p\t\t*p = %d\n", p, *p);

    j = *p + 5;
    printf("j = %d\t\t*p = %d\n", j, *p);

    *p = *p + i;
    printf("p = %p\t\t*p = %d\n", p, *p);

    p = &f;
    printf("p = %p\t\t*p = %d\n", p, *p);
}
```



Pointer to a variable

- The simplest operation is to get an address of a variable and store it in a pointer variable.

- Example

```
p = &k;  
p = &f;
```

- Pointer p is declared to store an address of an `int` variable.
- Assigning an address of a float variable to p, would cause a compiler warning or error. (Continuing in spite of warning, may result in disaster).

```
#include <stdio.h>  
#define line "\t-----\n"  
  
main()  
{  
    int i = 0;  
    int j = 1;  
    int k = 2;  
    float f = 3.0;  
    int *p = 0;  
  
    p = &k;  
    printf("p = %p\t\t*p = %d\n", p, *p);  
  
    j = *p + 5;  
    printf("j = %d\t\t*p = %d\n", j, *p);  
  
    *p = *p + i;  
    printf("p = %p\t\t*p = %d\n", p, *p);  
  
    p = &f;  
    printf("p = %p\t\t*p = %d\n", p, *p);  
}
```




Pointer to a variable

- The value of pointer `p` is an address of some `int` variable.
- The value of the `int` variable pointed to by `p` can be accessed by using a dereferencing operator `*`.
- Printf statement here prints the value of `p` and that of `*p`.

```
p = 0xbffffc8c      *p = 2
```

- `*p` is just like `k` here. All expressions where `k` can appear, `*p` also can.

```
j = 7              *p = 2
```

```
#include <stdio.h>
#define line "\t-----\n"

main()
{
    int i = 4;
    int j = 1;
    int k = 2;
    float f = 3.0;
    int *p = 0;

    p = &k;
    printf("p = %p\t\t*p = %d\n", p, *p);

    j = *p + 5;
    printf("j = %d\t\t*p = %d\n", j, *p);

    *p = *p + i;
    printf("p = %p\t\t*p = %d\n", p, *p);

    p = &f;
    printf("p = %p\t\t*p = %d\n", p, *p);
}
```



Pointer to a Variable

- The dereferencing operator `*p` is not only used to get the value of the variable pointed to by `p`, but also can be used to change the value of that variable.
- `&f` is address of a float type data. It is not compatible with the integer pointer `p`. So, we convert it to `(void *)`. This generic pointer type is compatible with all other pointer type.
- It is legal to use `*p` as left side of an assignment.

`P = 0xbffffc8c`

`*p = 6`

```
#include <stdio.h>
#define line "\t-----\n"

main()
{
    int i = 0;
    int j = 1;
    int k = 2;
    float f = 3.0;
    int *p = 0;

    p = &k;
    printf("p = %p\t\t*p = %d\n", p, *p);

    j = *p + 5;
    printf("j = %d\t\t*p = %d\n", j, *p);

    *p = *p + i;
    printf("p = %p\t\t*p = %d\n", p, *p);

    p = (void *) &f;
    printf("p = %p\t\t*p = %d\n", p, *p);
}
```

```
1 #include <stdio.h>
```

```
2 main(){
```

```
3     int a, *pa, **ppa;
```

```
4     int b=10;
```

```
5     int *pb;
```

```
6     pa = &a;
```

```
7     ppa = &pa;
```

```
8     a = 3;
```

```
9
```

```
10    pb=&b;
```

```
11
```

```
12    if (*pa == 3) printf("*pa==%d (a) Checked\n", a);
```

```
13    if (**ppa == 3) printf("**ppa==%d (a) Checked\n", a);
```

```
14
```

```
15    *ppa = &b; // *ppa means pa
```

```
16    printf("When *ppa updated, See how pa got updated. *pa=%d and b=%d\n", *pa, b);
```

```
17    *pa = 6; // At this moment, pa is pointint to b
```

```
18    printf("When *pa (to b) updated, See how b got updated. *pa=%d and b=%d\n", *pa, b);
```

```
19    *ppa = &a; // ppa's body (pa) is updated to a
```

```
20    **ppa = 7; // **ppa, *pa, a got updated.
```

```
21    printf("After **ppa=7 => **ppa=%d, *pa=%d, a=%d\n", **ppa, *pa, a);
```

```
22 }
```

	ppa's view	pa's view	a's view
2 nd Order Pointer	ppa	&pa	-
1 st Order Poitner	*pa	pa	&a
(object) 0 th Order Pointer	**ppa	*pa	a

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\pointer>pointer2
```

```
*pa==3 (a) Checked
```

```
**ppa==3 (a) Checked
```

```
When *ppa updated, See how pa got updated. *pa=10 and b=10
```

```
When *pa (to b) updated, See how b got updated. *pa=6 and b=6
```

```
After **ppa=7 => **ppa=7, *pa=7, a=7
```

Demo Program:
pointer2.c

LECTURE 2

Pointers and Functional Arguments

Pointers and Functions Arguments

Demo Program: swap.c

- We have seen that since a and b are ***passed by value*** to swap1 function, the values of a and b in main are not swapped.
- But in swap2, the address of a and address of b is passed. Hence pa and pb contains the addresses of a and b. Then the changes are made directly to the locations of a and b.
- The result would be

3 5
5 3

```
void swap1(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
void swap2(int *pa, int *pb) {  
    int temp = *pa;  
    *pa = *pb;  
    *pb = temp;  
}  
  
main() {  
    int a = 3, b = 5;  
  
    swap1(a, b);  
    printf("%d\t%d\n", a, b);  
    swap2(&a, &b);  
    printf("%d\t%d\n", a, b);  
}
```

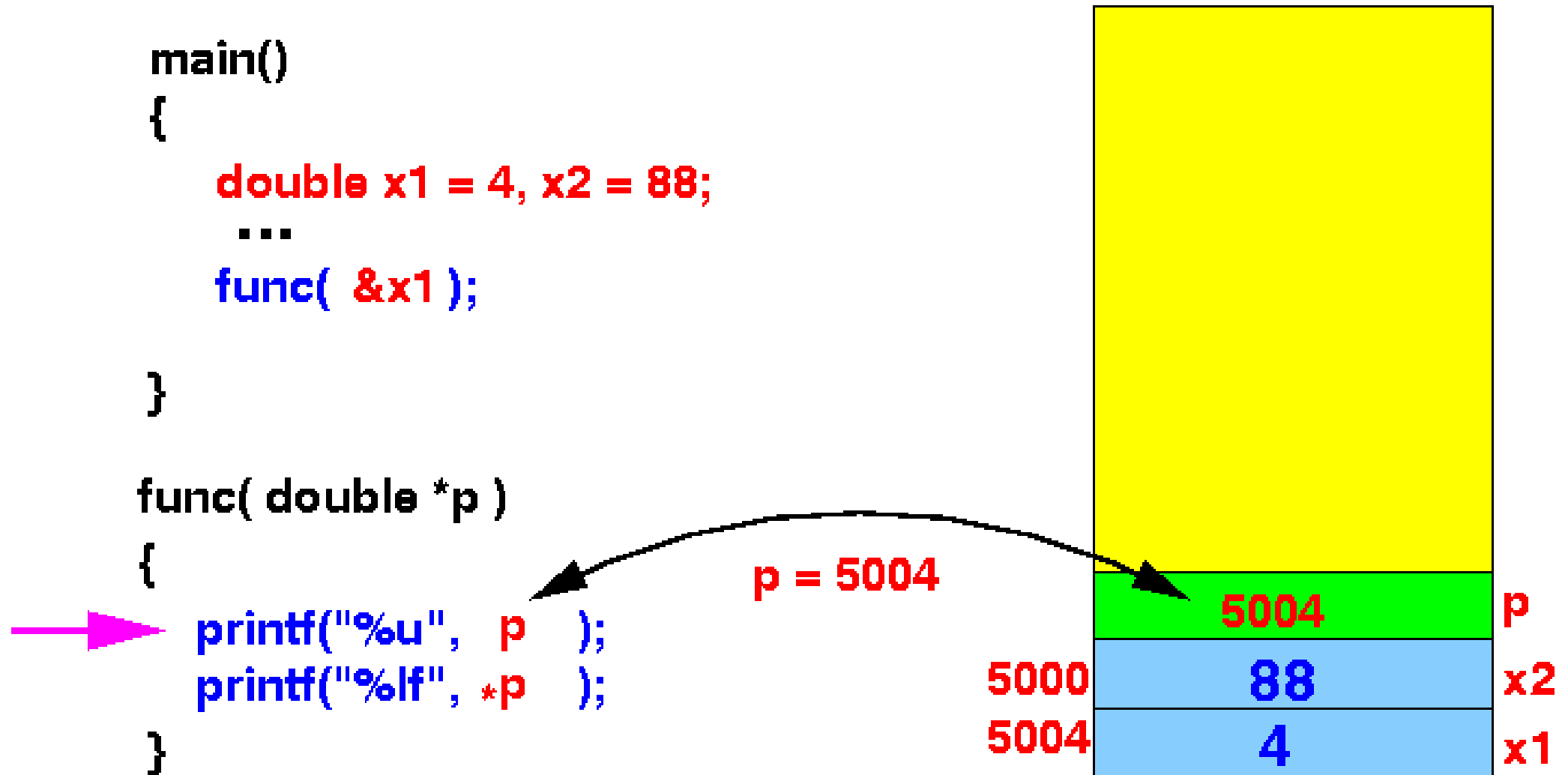


Pass by Reference

- If you want to modify the value of a variable (0th order pointer, you must pass its address to an argument of 1st order pointer).
- If you want to modify the value of a 1st order pointer, you must pass its address to an argument of 2nd order pointer.
- Pass by reference is achieved by escalating one level in pointer structure.

```
main()
{
    double x1 = 4, x2 = 88;
    ...
    func( &x1 );
}
```

```
func( double *p )
{
    printf("%u", p );
    printf("%lf", *p );
}
```



LECTURE 3

Pointers and Arrays

Pointers in C

Storage of Address

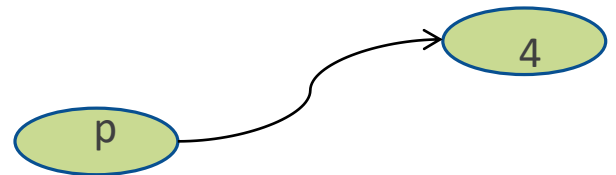
- A char pointer points to a single byte.
- An int pointer points to first of the four bytes.
- A pointer itself has an address where it is stored in the memory. Pointers are usually four bytes.

```
int *p; ⇔ int* p;
```

* is called the dereference operator. It is read as **body of** or **object of**.

- *p gives the value pointed by p

```
int i = 4;  
p = &i;
```



- & (ampersand) is called the reference operator. This & operator is overloaded (bit-wise and).
- &i returns the address of variable i

Pointers

They may point to a variable, an element in an array or an element in a struct.

```
int x = 1, y = 2, z[10];  
  
int *ip;           /* A pointer to an int */  
  
ip = &x;           /* Address of x */  
y = *ip;           /* Content of ip */  
*ip = 0;           /* Clear where ip points */  
ip = &z[0];         /* Address of first element of z */
```

Pointer Issues (I)

- For any type T, you may form a pointer type to T.
 - Pointers may reference a function or an object.
 - The value of a pointer is the address of the corresponding object or function
 - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: * dereferences a pointer, & creates a pointer (reference to)
 - **Example: (pointer to variables)**
 - ```
int i = 3;
int *j = &i;
*j = 4;
printf("i = %d\n", i); // prints i = 4
```
  - **Eexample: (pointer to a function)**
  - ```
int myfunc (int arg);           // declare a
myfunc function header.
int (*fptr)(int) = myfunc;      // declare a
pointer fptr of int (*)(int)
i = fptr(4); // same as calling myfunc(4);
```

Pointer Issues (II)

- Generic pointers: (Other pointers can be casted into or from)
 - Traditional C used (char *)
 - Standard C uses (void *) – these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use **NULL** or **0**. *It is a good idea to always initialize pointers to NULL.*



Arrays

- An array declaration has a form

```
Type array-name[constant-expression];
```

- Examples (Declaration of an Array)

```
int rollNos[100];
```

```
float marks[MAX*3];
```

- Arrays in C are indexed from 0.
- The elements of an array can be accessed by

```
array-name[int-expression]
```

1

Array Data Type

2

Array Pointer

3

Array address
calculation

4

Array Memory
Allocation (local,
global, extern,
heap)

5

Array Initialization

Design Issues in Array



Pointers and arrays

- Pointers and arrays are tightly coupled.

```
char a[] = "Hello World";
```

```
char *p = &a[0]; // a is also a pointer type when passed as argument to a function.
```

char a[12], *p = &a[0];											
*p	*(p+1)	*(p+2)	*(p+3)	*(p+4)	*(p+5)	*(p+6)	*(p+7)	*(p+8)	*(p+9)	*(p+10)	*(p+11)
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
H	e	l	l	o		W	o	r	l	d	'\0'



Array - Accessing an element

int A[6];	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
	0x1000	0x1004	0x1008	0x1012	0x1016	0x1020
	6	5	4	3	2	1

6 elements of 4 bytes each, total size = 6 x 4 bytes = 24 bytes

Read an element

```
int tmp = A[2];
```

Write to an element

```
A[3] = 5;
```




Arrays

An Array is a collection of variables of the same type that are referred to through a common name.

Declaration

```
type var_name[size]
```

e.g

```
int A[6];  
double d[15];
```



Array Initialization

- After declaration, array contains some garbage value.

- Static initialization

```
int month_days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

- Run time initialization

```
int i;  
int A[6];  
for(i = 0; i < 6; i++)  
    A[i] = 6 - i;
```



Demo Program:

polynomial.c

Polynomials

- Read the coefficients of a polynomial and print a table of values for all $x = 0.1, 0.2, \dots, 1.0$.

```
/* Evaluating a polynomial */
#include <stdio.h>
#include <math.h>

main() {
    int i, order;
    float coef[10], x, val;

    printf( "Enter the order -> " );
    scanf("%d" , &order);
    for ( i = 0; i <= order; i++ ){
        printf("a[%d] = ", i);
        scanf("%f", &(coef[i]) );
    }

    for ( x = 0; x <= 1; x += 0.1 ){
        val = coef[0];
        for ( i = 1; i <= order; i++ )
            val += coef[i] * pow(x,i);

        printf("%f\t%f",x,val);
    }
}
```

LECTURE 4

Address Calculation



Pointers and Arrays

Topic: address advancement

- An array name is a constant pointer. So if a is an array of 10 integers and pa is a pointer to int variable, then

```
Pa = &a[0];
```

```
Pa = a;
```

- Are legal and same. So are

```
B = a[5];
```

```
B = *(a+5);
```

- However since a is constant pointer following is invalid

```
A = pa;
```

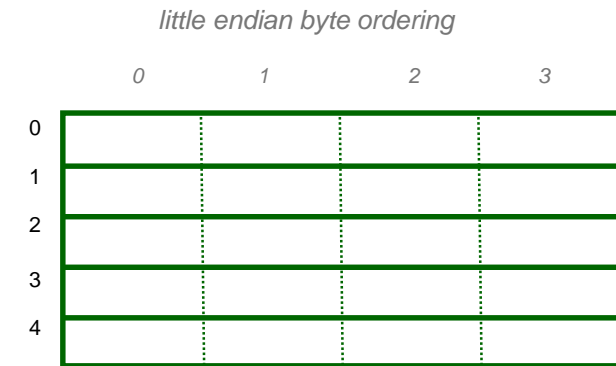


Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

- `int x[5];` // an array of 5 4-byte ints.

- All arrays begin with an index of 0



memory layout for array x

- An array identifier is equivalent to a pointer that references the first element of the array

- `int x[5], *ptr;`
`ptr = &x[0]` is equivalent to `ptr = x;`

- Pointer arithmetic and arrays:

- `int x[5];`
`x[2]` is the same as `*(x + 2)`, the compiler will assume you mean 2 objects beyond element x.



Pointers in C (and C++)

Step 1:

```
int main (int argc, argv) {  
    int  x = 4;  
    int *y = &x;  
    int *z[4] = {NULL, NULL, NULL, NULL};  
    int  a[4] = {1, 2, 3, 4};  
    ...  
}
```

Note: The compiler converts `z[1]` or `*(z+1)` to
Value at address (Address of `z` + `sizeof(int)`);

In C you would write the byte address as:
`(char *)z + sizeof(int);`

or letting the compiler do the work for you
`(int *)z + 1;`

Program Memory		Address
x	4	0x3dc
y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
z[3]	0	0x3cc
z[2]	0	0x3c8
z[1]	0	0x3c4
z[0]	0	0x3c0
a[3]	4	0x3bc
a[2]	3	0x3b8
a[1]	2	0x3b4
a[0]	1	0x3b0



Pointers Continued

Step 1:

```
int main (int argc, argv) {  
    int x = 4;  
    int *y = &x;  
    int *z[4] = {NULL, NULL, NULL, NULL};  
    int a[4] = {1, 2, 3, 4};  
}
```

Step 2: Assign addresses to array z

```
z[0] = a;           // same as &a[0];  
z[1] = a + 1;       // same as &a[1];  
z[2] = a + 2;       // same as &a[2];  
z[3] = a + 3;       // same as &a[3];
```

Program Memory		Address
x	4	0x3dc
y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
z[3]	0x3bc	0x3cc
z[2]	0x3b8	0x3c8
z[1]	0x3b4	0x3c4
z[0]	0x3b0	0x3c0
a[3]	4	0x3bc
a[2]	3	0x3b8
a[1]	2	0x3b4
a[0]	1	0x3b0



Pointers Continued

Step 1:

```
int main (int argc, argv) {  
    int x = 4;  
    int *y = &x;  
    int *z[4] = {NULL, NULL, NULL, NULL};  
    int a[4] = {1, 2, 3, 4};  
}
```

Step 2:

```
z[0] = a;  
z[1] = a + 1;  
z[2] = a + 2;  
z[3] = a + 3;
```

Step 3: No change in z's values

```
z[0] = (int *) ((char *)a);  
z[1] = (int *) ((char *)a  
                + sizeof(int));  
z[2] = (int *) ((char *)a  
                + 2 * sizeof(int));  
z[3] = (int *) ((char *)a  
                + 3 * sizeof(int));
```

Program Memory		Address
x	4	0x3dc
y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
z[3]	0x3bc	0x3cc
z[2]	0x3b8	0x3c8
z[1]	0x3b4	0x3c4
z[0]	0x3b0	0x3c0
a[3]	4	0x3bc
a[2]	3	0x3b8
a[1]	2	0x3b4
a[0]	1	0x3b0



Pointer Arithmetic

Demo Program: arithmetic.c

A simple arithmetic is allowed for the pointers.
A pointer points a variable of a given type.
When we add 1 to a pointer variable, it points to the next variable in the memory(though it may not be of the same type).
Here $p = p + 1$ would cause the value of p to be added 4 which is size of `int`.
Look at output.

```
#include <stdio.h>

void main(void){
    int i = 0;
    int j = 1;
    int k = 2;
    int *p = 0;
    int **f;
    p = &i;
    printf("p = %p\t\t*p = %d\n", p, *p);
    p = p + 1;
    printf("p = %p\t\t*p = %d\n", p, *p);
    p++;
    printf("p = %p\t\t*p = %d\n", p, *p);
    f = &p;
    printf("p = %p\t\t*p = %d\n", *f, **f);
}
```



Pointer Arithmetic

□ Variable Table

Var	Address	Value
i	0xbffffc94	0
j	0xbffffc90	1
k	0xbffffc8c	2
f	0xbffffc88	3.000000
p	0xbffffc84	(nil)

□ The output of this program is given here.

p = 0xbffffc8c	*p = 2
p = 0xbffffc90	*p = 1
p = 0xbffffc94	*p = 0
p = 0xbffffc88	*p = 1077936128 (float number in int format)

Pointer Arithmetic

- A 32-bit system has 32 bit address space.
- To store any address, 32 bits are required.
- Pointer arithmetic : $p+1$ gives the next memory location assuming cells are of the same type as the base type of p .



Pointer arithmetic: Valid operations

- $\text{pointer} + / - \text{integer} \rightarrow \text{pointer}$
- $\text{pointer} - \text{pointer} \rightarrow \text{integer}$
- $\text{pointer} \langle \text{any operator} \rangle \text{pointer} \rightarrow \text{invalid}$
 - $\text{pointer} + / - \text{pointer} \rightarrow \text{invalid}$



Pointer Arithmetic: Example

```
int *p, x = 20;  
p = &x;  
printf("p    = %p\n", p);  
printf("p+1 = %p\n", (int*)p+1);  
printf("p+1 = %p\n", (char*)p+1);  
printf("p+1 = %p\n", (float*)p+1);  
printf("p+1 = %p\n", (double*)p+1);
```

Sample output:

```
p    = 0022FF70  
p+1 = 0022FF74  
p+1 = 0022FF71  
p+1 = 0022FF74  
p+1 = 0022FF78
```

{program: pointer_arithmetic.c}

LECTURE 5

Pointer to Characters



Strings in C

No “Strings” keyword A string is an array of characters.

OR

```
char string[] = "hello world";  
char *string = "hello world";
```

A C String of Characters with Addresses											
1234:0000	1234:0001	1234:0002	1234:0003	1234:0004	1234:0005	1234:0006	1234:0007	1234:0008	1234:0009	1234:000A	1234:000B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
H	e	l	l	o		W	o	r	l	d	'\0'



Significance of NULL character '\0'

```
char string[] = "hello world";  
printf("%s", string);
```

- Compiler has to know where the string ends

'\0' denotes the end of string

- Some more characters (do \$man ascii):

'\n' = new line, '\t' = horizontal tab, '\v' = vertical tab, '\r' = carriage return
'A' = 0x41, 'a' = 0x61, '\0' = 0x00

String and Array of Characters

```
char a[] = {'A', 'B', 'C'};  
// This is an array of characters. a is a pointer to character array.  
char b[] = {'A', 'B', 'C', '\0'};  
// This is an array of characters with 0 ending and that is a string.  
// b is a pointer to string  
char c[] = "ABC";  
// This is a string. It is the same with the line above.  
// c is also a pointer to string.  
char *d = "ABC";  
// This is a string. d is a pointer to the starting character of "ABC"  
char e[256]; strcpy(e, d);  
// b, c, d, e have the same string value. b, c, d, e are all strings.  
// e is an array of 256 char elements. It is stored with a string of  
// length 3 (4 elements.)
```



Three Ways to Declare and Initialize an Array (Allocation of Memory)

```
char *str; // declare a pointer to an character
char *str = "Hello Word!"; // declare a pointer to an character and pointing it to
// a array of characters with a string ending
char str[256]; // allocate fixed-size array
char str[256] = "Hello World!" // allocate fixed-size array and redirected pointer
// to a string. No, difference from char str[] = "Hello World!";
char *str = calloc(256, sizeof(char)); // allocate memory for the character array in
// data heap
```

LECTURE 6

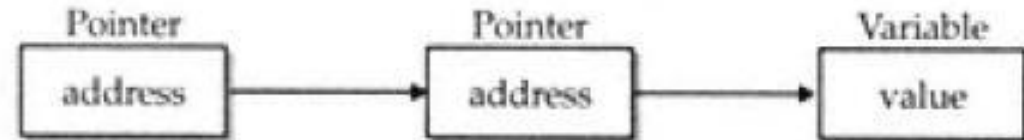
Pointer Arrays

Pointer to Pointer

- Declaration
 - Place an additional asterisk
- ```
double **newbalance;
```
- newbalance is a pointer to a float pointer.



Single Indirection



Multiple Indirection

# Review of Pointer Levels

```
int a, *pa, **ppa;
int b=10;
int *pb;
pa = &a;
ppa = &pa;
a = 3;
```

|                                        | ppa's view | pa's view | a's view |
|----------------------------------------|------------|-----------|----------|
| 2 <sup>nd</sup> Order Pointer          | ppa        | &pa       | -        |
| 1 <sup>st</sup> Order Poitner          | *ppa       | pa        | &a       |
| (object) 0 <sup>th</sup> Order Pointer | **ppa      | *pa       | a        |



# Pointer to Pointer

---

```
#include <stdio.h>

int main() {
 int x, *p, **q;
 x = 10;
 p = &x;
 q = &p;

 printf("%d %d %d\n", x, *p, **q);
 return 0;
}
```



```
int **a; int *b[2]; int c[][2];
```

(2<sup>nd</sup> level pointer, array of pointers and 2D array)

---

- a is a pointer to a pointer. (It is only a pointer.)
- b is an array of pointers. (It is 1-D array of pointers.)
- c is a 2D array. (It is 2-D array of arrays.)

Note: 2<sup>nd</sup> dimension must be declared and c must be initialized when declared. Arrays need memory allocation, pointers don't.

But, as a variable a, b, are all level 2 pointers. They are similar but not exactly the same.

Demo Program: [pointers2D.c](#)

Run: [gopointers2D.bat](#)



```

#include <stdio.h>
#include <stdlib.h>
int c[][2] = {1, 2, 3, 4}; // row-major 2D array
int **a; // pointer of pointers
int *b[2]; // array of pointers (2 row pointers)
int main(void){
 int i, j;
 a = (int **) calloc(2, sizeof(int *));
 a[0] = c[0]; a[1] = c[1];
 b[0] = c[0]; b[1] = c[1];
 printf("Using a pointer to pointers\n");
 for (i=0; i<2; i++) {
 for (j=0; j<2; j++) printf("%d ", a[i][j]);
 printf("\n");
 }
 printf("Using b 1D array of pointers\n");
 for (i=0; i<2; i++) {
 for (j=0; j<2; j++) printf("%d ", b[i][j]);
 printf("\n");
 }
 printf("Using c 2D - array\n");
 for (i=0; i<2; i++) {
 for (j=0; j<2; j++) printf("%d ", c[i][j]);
 printf("\n");
 }
 return 0;
}

```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\pointers>gopointers2D
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\pointers>gcc pointers2D.c -o pointers2D
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\pointers>pointers2D
```

```
Using a pointer to pointers
```

```
1 2
```

```
3 4
```

```
Using b 1D array of pointers
```

```
1 2
```

```
3 4
```

```
Using c 2D - array
```

```
1 2
```

```
3 4
```



# Level 2 Pointers' Usage

---

- Pointing to a 2-D array.
- Pointing to an array of pointers.
- Used as an argument to update an 1-D pointer.

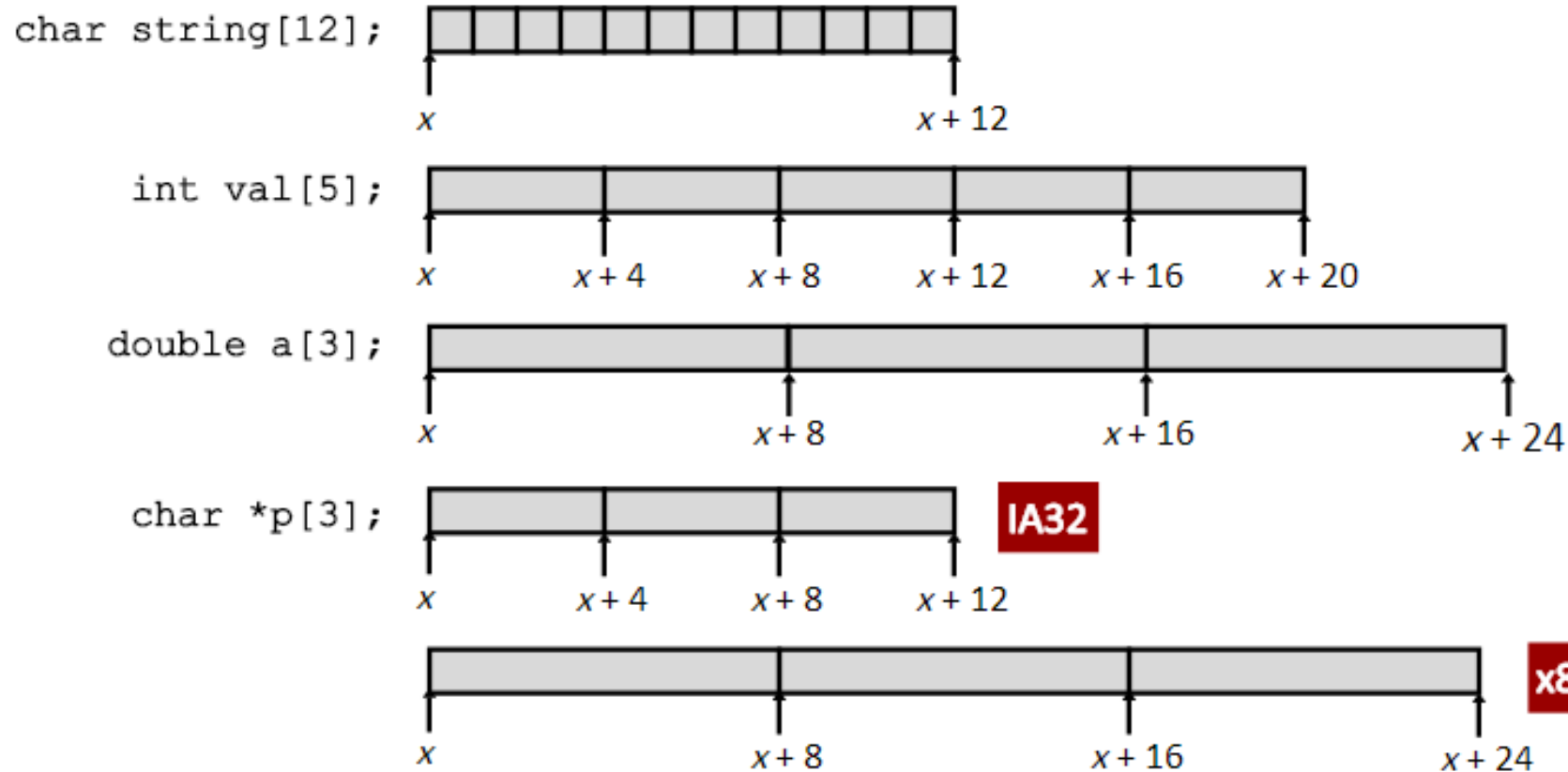


# Array Allocation

## ■ Basic Principle

$T\ A[N];$

- Array of data type  $T$  and length  $N$
- Contiguously allocated region of  $N * \text{sizeof}(T)$  bytes

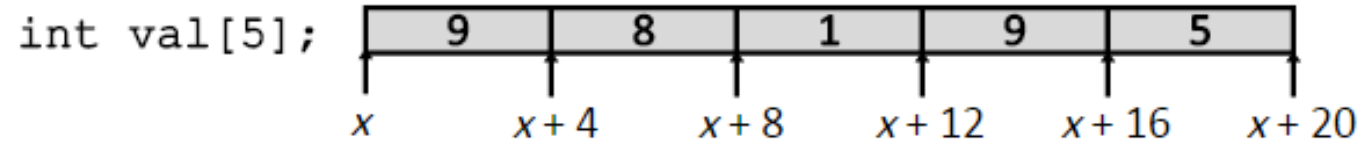




## ■ Basic Principle

$T$   $A[N]$  ;

- Array of data type  $T$  and length  $N$
- Identifier  $A$  can be used as a pointer to array element 0: Type  $T^*$



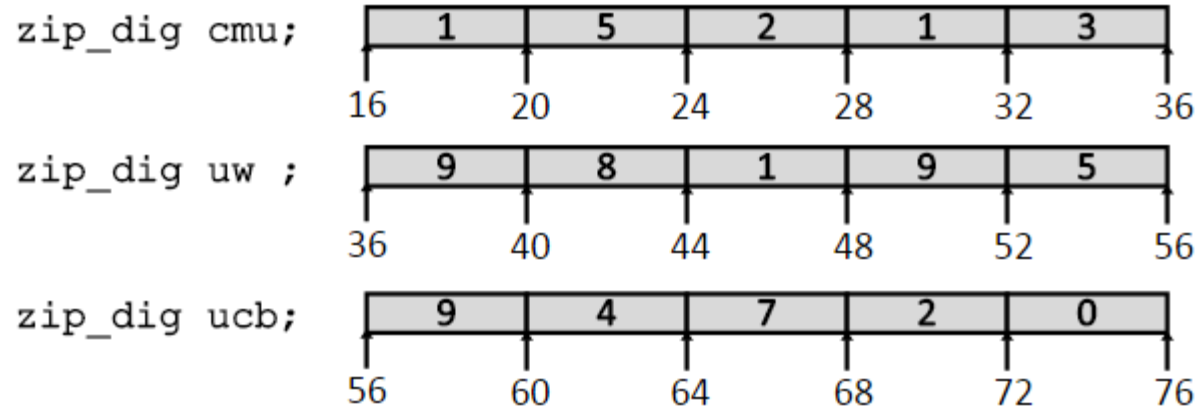
| Reference                | Type               | Value  |
|--------------------------|--------------------|--------|
| <code>val[4]</code>      | <code>int</code>   | 5      |
| <code>val</code>         | <code>int *</code> | $x$    |
| <code>val+1</code>       | <code>int *</code> | $x+4$  |
| <code>&amp;val[2]</code> | <code>int *</code> | $x+8$  |
| <code>val[5]</code>      | <code>int</code>   | ??     |
| <code>*(val+1)</code>    | <code>int</code>   | 8      |
| <code>val + i</code>     | <code>int *</code> | $x+4i$ |



## Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration “`zip_dig uw`” equivalent to “`int uw[5]`”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general



# Array Accessing Example

zip\_dig uw;     

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 8 | 1 | 9 | 5 |
|---|---|---|---|---|

36       40       44       48       52       56

```
int get_digit
(zip_dig z, int dig)
{
 return z[dig];
}
```

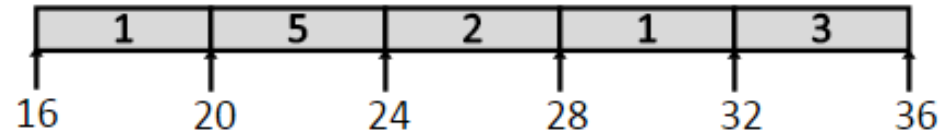
## IA32

```
%edx = z
%eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```

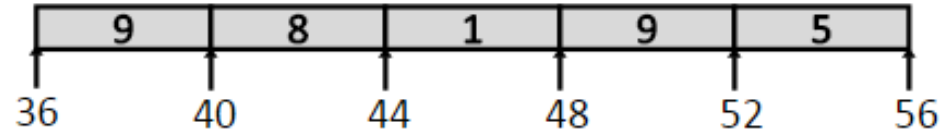


## Referencing Examples

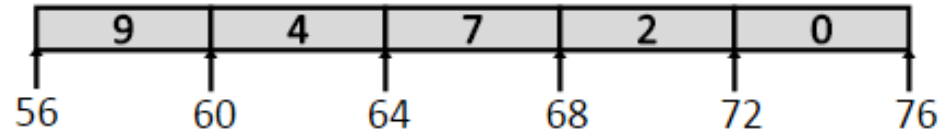
zip\_dig cmu;



zip\_dig uw ;



zip\_dig ucb;



| Reference | Address            | Value | Guaranteed? |
|-----------|--------------------|-------|-------------|
| uw[3]     | $36 + 4 * 3 = 48$  | 9     | Yes         |
| uw[6]     | $36 + 4 * 6 = 60$  | 4     | No          |
| uw[-1]    | $36 + 4 * -1 = 32$ | 3     | No          |
| cmu[15]   | $16 + 4 * 15 = 76$ | ??    | No          |

- No bound checking
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Index, \*, & are just ways to calculate reference address

LECTURE 7

# N-D Dimensional





## 2-Dimensional Arrays (Array of arrays)

---

```
int d[3][2];
```

- Access the point 1, 2 of the array:

```
d[1][2]
```

- Initialize (without loops):

```
int d[3][2] = {{1, 2}, {4, 5}, {7, 8}};
```



# More about 2-Dimensional arrays

- A Multidimensional array is stored in a row major format.
- A two dimensional case:
  - ➔ next memory element to `d[0][3]` is `d[1][0]`

|                      |                      |                      |                      |
|----------------------|----------------------|----------------------|----------------------|
| <code>d[0][0]</code> | <code>d[0][1]</code> | <code>d[0][2]</code> | <code>d[0][3]</code> |
| <code>d[1][0]</code> | <code>d[1][1]</code> | <code>d[1][2]</code> | <code>d[1][3]</code> |
| <code>d[2][0]</code> | <code>d[2][1]</code> | <code>d[2][2]</code> | <code>d[2][3]</code> |

- What about memory addresses sequence of a three dimensional array?
  - ➔ next memory element to `t[0][0][0]` is `t[0][0][1]`



# Multidimensional Arrays

---

## Syntax

```
type array_name[size1][size2]...[sizeN];
```

e.g

size of array = 3 x 6 x 4 x 8 x 4 bytes

```
int a[3][6][4][8];
```



# Demo Program

cube.c

## Go Dev C++!!!

C:\Eric\_Chou\C Course\C Programming Essentials\CDev\Ch8\ND\ND.exe

```
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
16 17 18
19 20 21
22 23 24
25 26 27
```

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 3

int cube[SIZE][SIZE][SIZE]= {
 {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}},
 {{10, 11, 12}, {13, 14, 15}, {16, 17, 18}},
 {{19, 20, 21}, {22, 23, 24}, {25, 26, 27}}
};

int main(int argc, char *argv[]) {
 int i=0;
 int j=0;
 int k=0;
 for (i=0; i<SIZE; i++) {
 for (j=0; j<SIZE; j++) {
 for (k=0; k<SIZE; k++){
 printf("%2d ", cube[i][j][k]);
 }
 printf("\n");
 }
 }

 return 0;
}
```

LECTURE 8

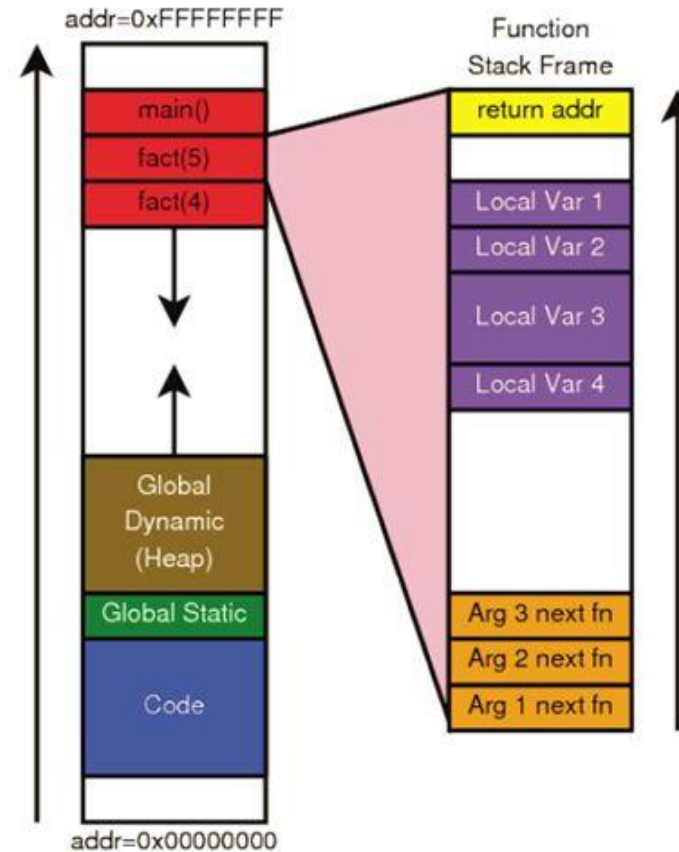
# Initialization of pointers and Arrays

# C Memory Model – where are variables & program code stored in the computer memory?

- Program code
- Function variables
  - Arguments
  - Local variables
  - Return location
- Global Variables
  - Statically allocated
  - Dynamically allocated

```
int fact (int n)
{
 return (n*fact (n-1)) ;
}

void main() { ... fact(5); ...}
```





# Dynamic Memory Allocation (Heap)

---

- To allocate memory at run time.
- malloc(), calloc()
  - both return a void\*
  - you'll need to typecast each time.

```
char *p;
p = (char *) malloc(1000); /*get 1000 byte space */
```

```
int *i;
i = (int *) malloc(1000*sizeof(int));
```



# Dynamic Memory Allocation (Heap)

---

- To free memory
- free()
  - free(ptr) frees the space allocated to the pointer ptr

```
int *i;
i = (int *)malloc(1000*sizeof(int));
.
.
.
free(i);
```





# Which memory location to use?

---

1. Global variables are always available.
  - Used like private data field in a module. The functions usually can be used as the setter, mutators for the data fields
2. Dynamic variables are allocated when programmer needs it.
  - It can be returned when it is not needed.
  - It is more flexible when memory size is an issue.
3. Local variables exist only when the function is in power.
  - These variables are usually related to the function only.



# Demo Program:

[memory.c](#)

---

## Go gcc!!!

# C Memory Model

## and The Corresponding Declared Array

```
8 int main(void){
9 int i=0;
10 int e[2] = {9, 8};
11 int *g;
12 printf("Global Int: \n");
13 printf("%-0x = %d\n", &a, a); printf("%-0x = %d\n", &b, b);
14 printf("Global Initialized Array\n");
15 printf("%-0x = %d\n", &c[0], c[0]); printf("%-0x = %d\n", &c[1], c[1]);
16 printf("Global Uninitialized:\n");
17 for (i=0; i<3; i++){
18 d[i] = i; // allocated at this time
19 printf("%-0x = %d\n", &d[i], d[i]);
20 }
21 printf("Local Initialized Array\n");
22 printf("%-0x = %d\n", &e[0], e[0]); printf("%-0x = %d\n", &e[1], e[1]);
23 printf("Global Dynamic Array:\n");
24 f = calloc(3, sizeof(int));
25 for (i=0; i<3; i++){
26 f[i] = i*2; // allocated at this time
27 printf("%-0x = %d\n", &f[i], f[i]);
28 }
29 printf("Local Dynamic Array:\n");
30 g = calloc(3, sizeof(int));
31 for (i=0; i<3; i++){
32 g[i] = i*2; // allocated at this time
33 printf("%-0x = %d\n", &g[i], g[i]);
34 }
35 return 0;
36 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int a=3; int b=5;
4 static int c[2] = {1, 2};
5 static int d[3]; // The keyword static is redundant in this case
6 int *f;
7
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\memory>memory
Global Int:
402000 = 3
402004 = 5
Global Initialized Array
402008 = 1
40200c = 2
Global Uninitialized:
405008 = 0
40500c = 1
405010 = 2
Local Initialized Array
60ff00 = 9
60ff04 = 8
Global Dynamic Array:
6d15b0 = 0
6d15b4 = 2
6d15b8 = 4
Local Dynamic Array:
6d15c8 = 0
6d15cc = 2
6d15d0 = 4
```

Command-line  
arguments

Stack

Heap

Initialized Data Segment

Uninitialized Data  
Segment

LECTURE 9

# N-Dimensional Arrays versus Pointers



# Arrays of Pointers

---

```
int *x[10];
```

- Declares an array of int pointers. Array has 10 pointers.
- Assign address to a pointer in array
- To find the value of var,

```
x[2] = &var;
```

```
int i = *x[2];
```

C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat

- `int *a[n]`, n-element array of pointers to integer
- `int (*a)[n]`, pointer to n-element array of integers

# Pointer to Pointer

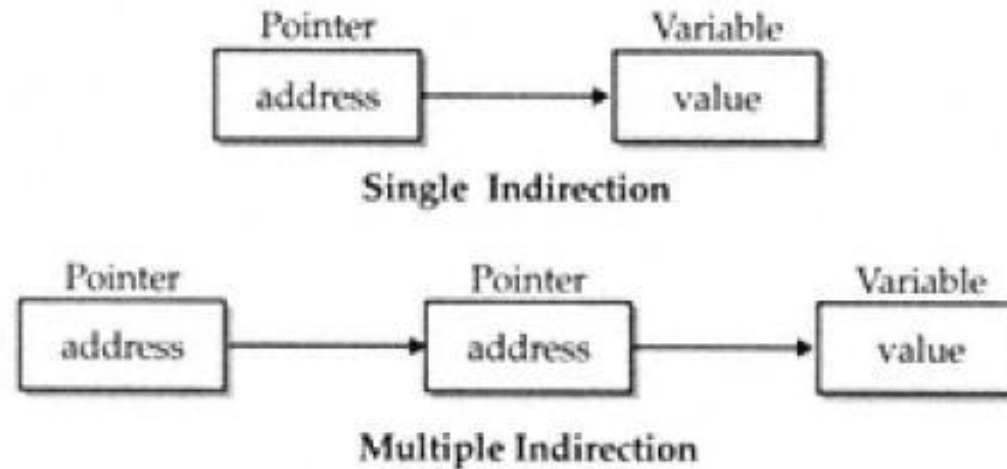
## (Level 2 Pointers, Pointer to 2D Array)

### Declaration

- Place an additional asterisk

```
double **newbalance;
```

newbalance is a pointer to a double pointer.





# Demo Program:

[pointer3.c](#)

---

## Go Dev C++!!!

```
#include <stdio.h>

int main() {
 int x, *p, **q;
 x = 10;
 p = &x;
 q = &p;

 printf("%d %d %d\n", x, *p, **q);
 return 0;
}
```



# What is the difference between the following declarations:

---

1) `int* arr1[8];`

2) `int (*arr2)[8];`

3) `int *(arr3[8]);`

•What is the general rule for understanding more complex declarations?



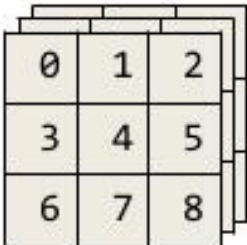
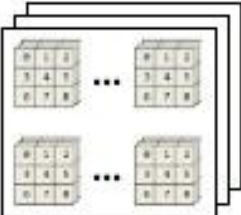
1) `int* arr[8];` // An array of int pointers. Same for `int *arr[8];` Read to the right, it is an 8-element array of int pointers.

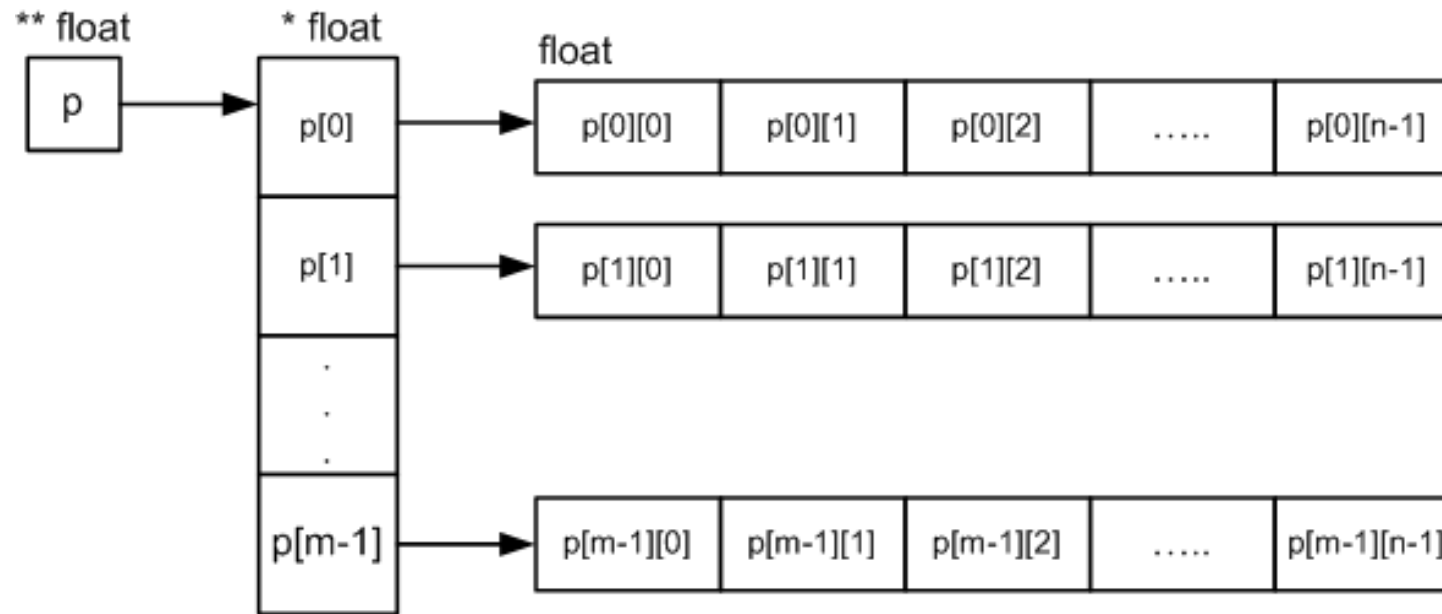
2) `int (*arr)[8];` // A pointer to an array of integers. With parentheses, arr2 is a pointer to an array of 8 int elements. Note: Parentheses takes higher priority

3) `int *(arr3[8]);` // arr3 is an 8 element array of int pointer type. Same as 1)



# What is an array?

| Dimensions | Example                                                                               | Terminology                                |
|------------|---------------------------------------------------------------------------------------|--------------------------------------------|
| 1          |    | Vector                                     |
| 2          |    | Matrix                                     |
| 3          |   | 3D Array<br>(3 <sup>rd</sup> order Tensor) |
| N          |  | ND Array                                   |



#### DYNAMIC ARRAY (2D) USING POINTER TO ARRAY OF POINTERS

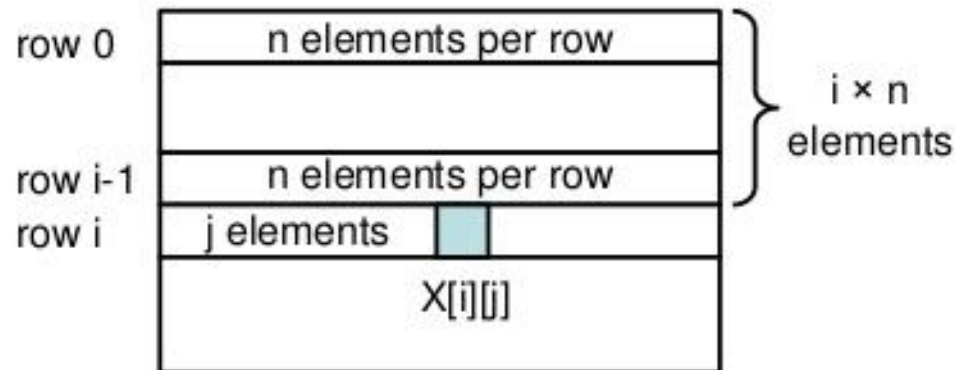
- Pointer 'p' points to an array of pointers to float
- Each element of this array points to a block of memory sufficient to store one row
- The way memory is allocated, it is possible that the memory allocated to the different rows may not be contiguous

# C 2D Arrays `type p[m][n];`

---

## Address Calculation for 2D Arrays

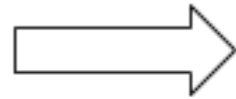
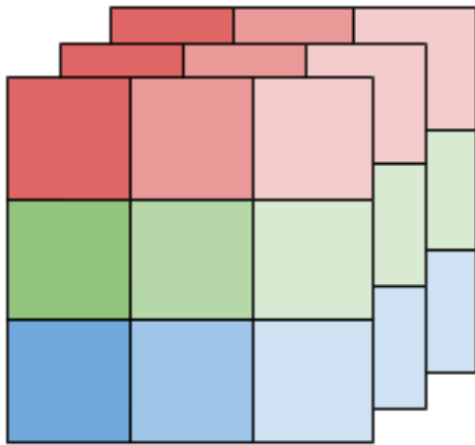
- ❖ Row-Major Order: 2D arrays are stored as rows
- ❖ Calculate Address of:  $X[i][j]$   
= Address of  $X + (i \times n + j) \times 8$  (8 bytes per element)



- ❖ Address of  $Y[i][k]$  = Address of  $Y + (i \times n + k) \times 8$
- ❖ Address of  $Z[k][j]$  = Address of  $Z + (k \times n + j) \times 8$

# 3D Array

row,col,depth



|       |       |       |
|-------|-------|-------|
| 0,0,0 | 0,1,0 | 0,2,0 |
| 1,0,0 | 1,1,0 | 1,2,0 |
| 2,0,0 | 2,1,0 | 2,2,0 |

|       |       |       |
|-------|-------|-------|
| 0,0,1 | 0,1,1 | 0,2,1 |
| 1,0,1 | 1,1,1 | 1,2,1 |
| 2,0,1 | 2,1,1 | 2,2,1 |

|       |       |       |
|-------|-------|-------|
| 0,0,2 | 0,1,2 | 0,2,2 |
| 1,0,2 | 1,1,2 | 1,2,2 |
| 2,0,2 | 2,1,2 | 2,2,2 |



LECTURE 10

# Console Argument Vector and Counter



# Main Function Structure (I)

No System Exit Code Return and No Arguments

---

```
void main(void){
 /* put your program here */
}
```



# Main Function Structure (II)

With System Exit Code Return and No Arguments

---

```
int main(void){
 /* put your program here */
 return 0;
}

// EXIT_SUCCESS and EXIT_FAILURE
```



# Main Function Structure (III)

With System Exit Code Return and Arguments

---

```
int main(int argc, char *argv[]){
 /* put your program here */
 return 0;
}
```





# The main( ) function

---

- So far, we have been defining the main() function to receive no arguments.
- Actually, the main( ) function can receive two arguments.
- To do that, the main( ) function should be defined as below.
- This is how arguments can be passed in at the command line.

```
int main(int argc, char *argv[]) { ... }
```



# Command Line Arguments

---

- From the command prompt, we can start running a program by typing its name and pressing ENTER.
- To pass arguments, we type in the program's name followed by some arguments, then press ENTER.
- Below is an example from the Windows command prompt.

% myprog

% myprog 5 23

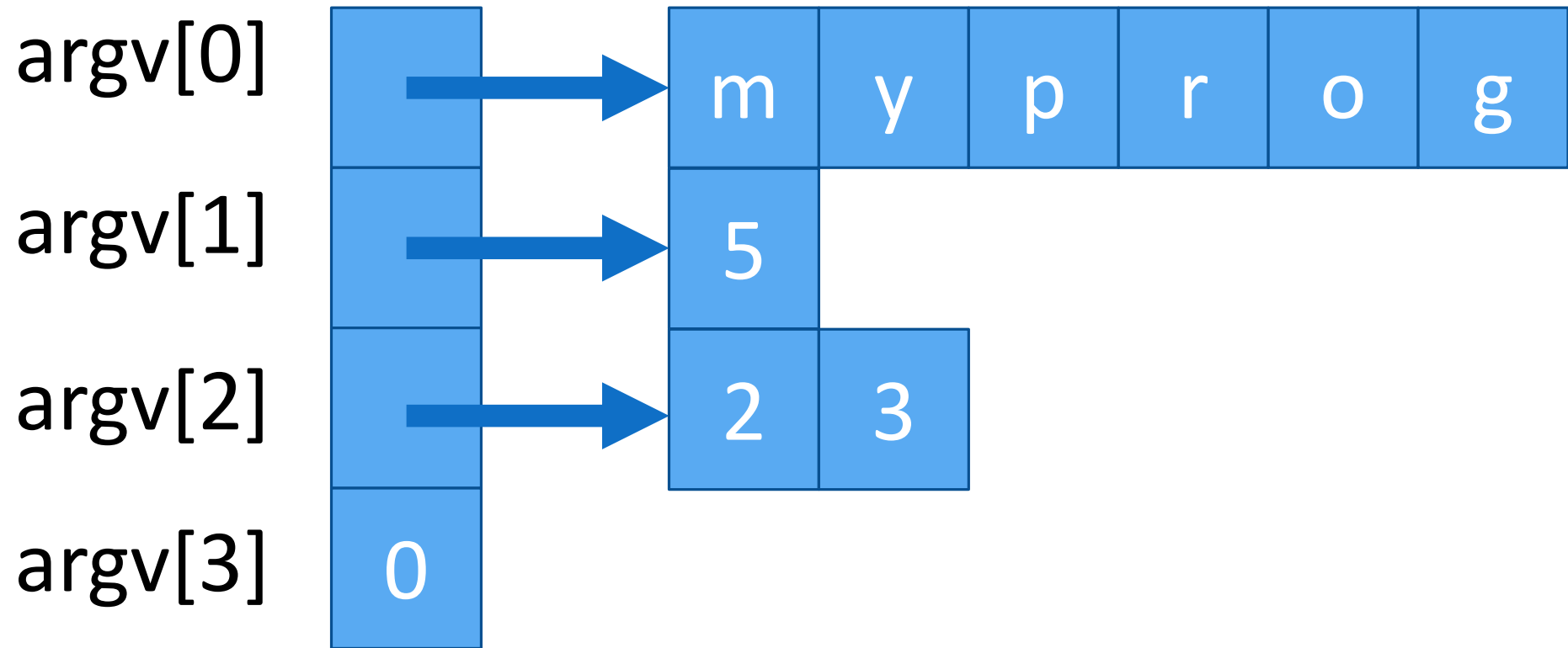


# argc and argv

---

- When the program is called from the command line with arguments, `argc` will contain the number of arguments.
- When the user types in arguments, the user separates each argument with a space.
- **argv** is an array of character strings.
- **argv[1]** is the character string containing the first argument, **argv[2]** the second, etc.
- Note: **argv[0]** contains the character string that is the name of the executable/binary.

argc = 3





# Demo Program

myprog.c

---

```
int main(int argc, char *argv[]) {
 int i;
 printf();
 printf("\n");
 return 0;
}
```



# Functions to Convert the Argument Strings to int and float

---

- Use this library file `#include <string.h>`
- Functions to convert argument strings to int, float or long data types:
  - `atoi(str);` // convert the str string to integer
  - `atof(str);` // convert the str string to float number
  - `atol(str);` // convert the str string to long integer



# Demo Program:

## argument.c

# Go gcc!!!

goargument.bat

```
1 gcc argument.c -o argument
2 argument 5 23
```

argument.c:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
 int i;
 for (i = 1; i < argc; i++)
 printf("%s ", argv[i]);
 printf("\n");
 if (argc >= 3){
 printf("First Argument = %d\n", atoi(argv[1]));
 printf("Second Argument = %8.4f\n", atof(argv[2]));
 }
 return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\argument>goargument
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\argument>gcc argument.c -o argument
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch8\argument>argument 5 23
5 23
First Argument = 5
Second Argument = 23.0000
```

LECTURE 11

# Pointer to Function





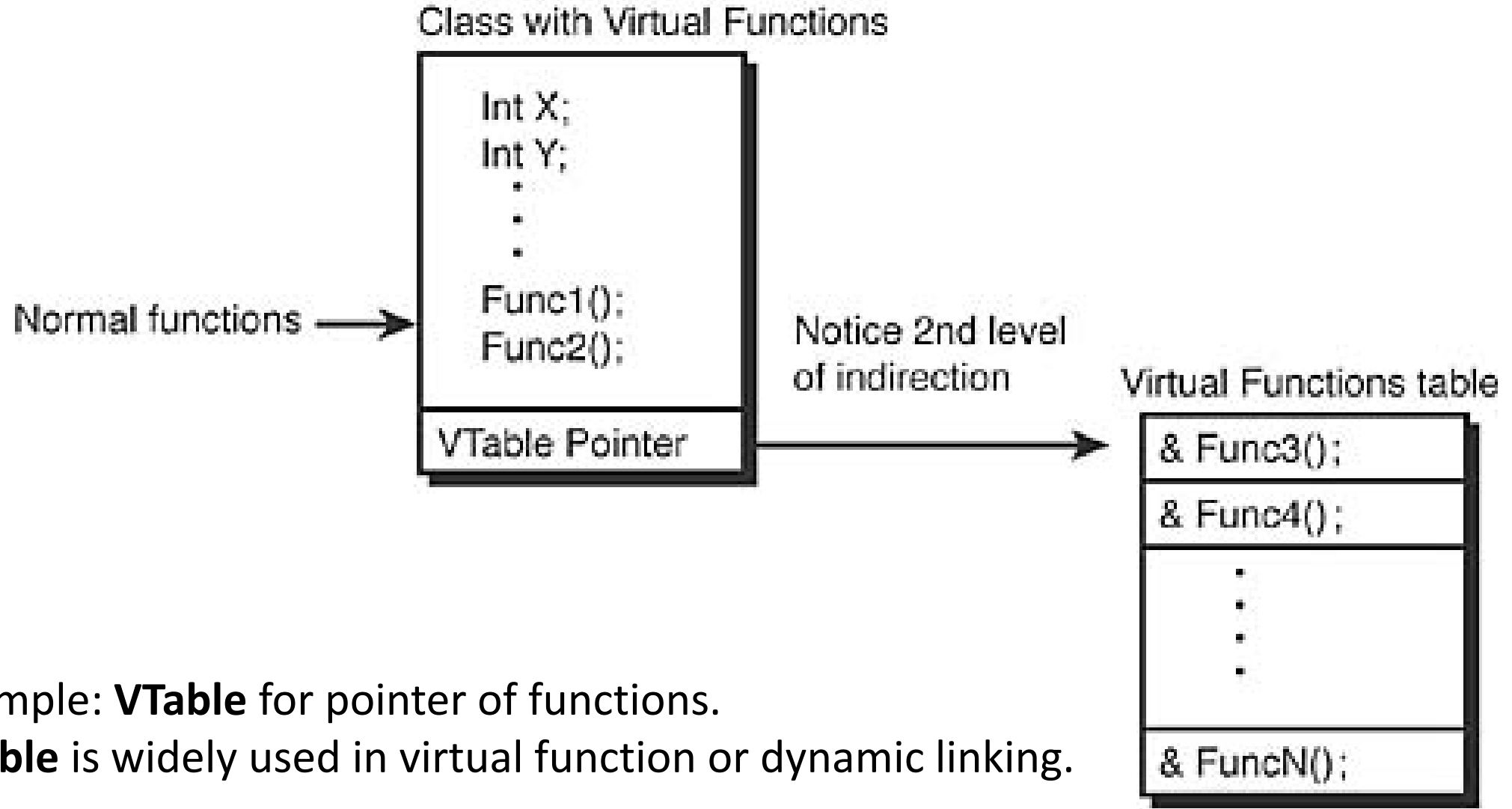
# Pointer to function

---

Pointer to function provide the programmer the flexibility to pick a proper function to execute under different special needs:

- Different functions,
- Different number of parameters, and
- Different parameter types.

A good example is the integrator or differentiator function which finds integral or derivatives of another function. The function can be passed as an argument to the integrator or differentiator.



Example: **VTable** for pointer of functions.

**VTable** is widely used in virtual function or dynamic linking.



# Pointer to Function Type

---

## Syntax:

```
int (*f)(); // (*f): f is a pointer of int function type.
```

## Example:

```
int square(int x) { return x * x; }
int (*f) ();
int main(int argc, char **argv){
 f = □
 printf("%d\n", f(3)); // output 9
 return 0;
}
```



# Pointer to a Function as an Argument to Another Function

---

## Example:

```
int run(int (*f)(int), x){
 return f(x);
} // the result of f(x) will be returned. f can be any function of one parameter
```



# Demo Program:

## funcPointer.c

# Go Dev C++!!!

```
#include <stdio.h>
int square(int x){
 return x * x;
}

int doubled(int x){
 return 2 * x;
}

int run(int (*f)(int), int x){

 return f(x);
}

int main(int argc, char** argv) {
 printf("square(3)=%d\n", run(&square, 3));
 printf("double(3)=%d\n", run(&doubled, 3));
 return 0;
}
```

C:\Eric\_Chou\C Course\C Programming Essentials\CDev\Ch8\funcPointer\func...

```
square(3)=9
double(3)=6

Process exited after 0.01172 seconds with return value 0
Press any key to continue . . .
```



# Application of Pointer to Functions

---

- Integration and Differentiation Operations (Mathematical Solvers)
- Functional Table (VTable)
- Interrupt Handler
- Event Handler (One common use is to implement a callback function.)
- Discrete-Time simulation
- plugins and extensions - the pointers to functions provided by plugins or library extensions are gathered by a standard function **GetProcAddress**, **dlsym** or similar, which take the function identifier as name and return a function pointer. Absolutely vital for APIs like OpenGL.

LECTURE 12

# Complicated Declaration of Arrays and Pointers



# C Pointers And Array Reference

---

- C pointers and arrays

```
int *a == int a[]
```

```
int **a == int *a[]
```

- BUT equivalences don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension
  - otherwise it allocates a pointer

```
int **a, int *a[] pointer to pointer to int
```

```
int *a[n], n-element array of row pointers
```

```
int a[n][m], 2-d array
```





# C Pointers And Array Reference

---

- Compiler has to be able to tell the size of the things to which you point

- So the following aren't valid:

```
int a[][] bad
int (*a) [] bad
```

- C declaration rule: **read right as far as you can (subject to parentheses), then left, then out a level and repeat**

`int *a[n]` , n-element array of pointers to integer

`int (*a)[n]` , pointer to n-element array of integers



# C Pointers And Array Reference

```
int n;
int *a; /* pointer to integer */
int b[10]; /* array of 10 integers */
```

Array Reference in integer Type (address calculation in the unit of integer (4 bytes))

```
1. a = b; /* make a point to the initial element of b */
2. n = a[3]; /* The third integer array element
3. n = *(a+3); /* The body of a pointer + 3 integer away location
4. n = b[3]; /* equivalent to previous line */
5. n = *(b+3); /* equivalent to previous line */
```

## DESIGN & IMPLEMENTATION

### Pointers and arrays

Many C programs use pointers instead of subscripts to iterate over the elements of arrays. Before the development of modern optimizing compilers, pointer-based array traversal often served to eliminate redundant address calculations, thereby leading to faster code. With modern compilers, however, the opposite may be true: redundant address calculations can be identified as common subexpressions, and certain other code improvements are easier for indices than they are for pointers. In particular, as we shall see in [Chapter 16](#), pointers make it significantly more difficult for the code improver to determine when two l-values may be *aliases* for one other.

Today the use of pointer arithmetic is mainly a matter of personal taste: some C programmers consider pointer-based algorithms to be more elegant than their array-based counterparts; others simply find them harder to read. Certainly the fact that arrays are passed as pointers makes it natural to write subroutines in the pointer style.



# C Pointers to Functions

---

```
int (*function)(int);
```

```
int (*function[3])(int); // 3 element array of function pointers
```

```
int a[3]; // a is a 3-element array
```

```
int *a[3]; // a is a 3-element array of pointers
```

```
int (*a[3])(int); // a is a 3-element array of pointers to functions
```



# Demo Program:

[arrayFunc.c](#)

---

## Go Dev C++!!!

```
#include <stdio.h>
#include <stdlib.h>

int (*function[3])(int); // 3 element array of function pointers

int f1(int x){
 return x;
}
int f2(int y){
 return y;
}
int f3(int z){
 return z;
}

/* run this program using the console pauser or add your own getch, system("pause") or input loop */

int main(int argc, char *argv[]) {
 function[0]=&f1;
 function[1]=&f2;
 function[2]=&f3;
 int i=0;
 for (i=0; i<3; i++){
 printf("function%d=%d\n", i, i*2, function[i](i*2));
 }

 return 0;
}
```