Week 6

XMUT-NWEN 241 - 2024 T2

Systems Programming

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Admin

- Assignment 2
  - Due date is 20 Oct

Content

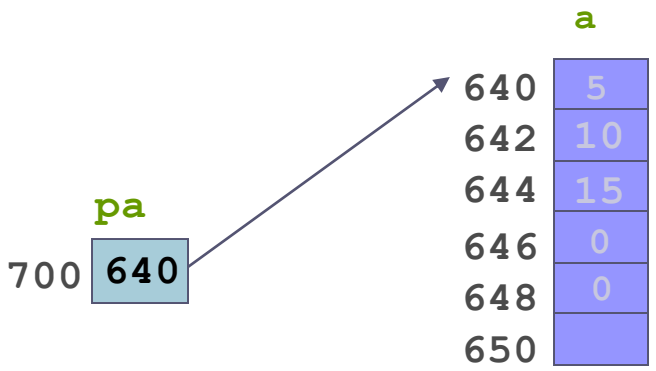
- Pointers and Strings
- Pointer to Pointer
- Pointers and 2-D arrays
- Pointers and Passing Function Parameters

Pointers

Pointer Arithmetic

Assume short is 2 bytes, and pointer variable (address size) is 4 bytes.

```
short a[5]={5, 10, 15};
short *pa;
pa = a;
int i = 5
```



Variable	Address	Value
a	640	5
	642	10
	644	15
	…	…
pa	700	640

Questions:

Expression	Value	Note
pa+1	642	640+1*2
pa+3	646	640+3*2
pa+i	650	640+i*2
*pa+1	6	5+1
*(pa+1)	10	a[1]=pa[1]=*(a+1)
pa[2]	15	644

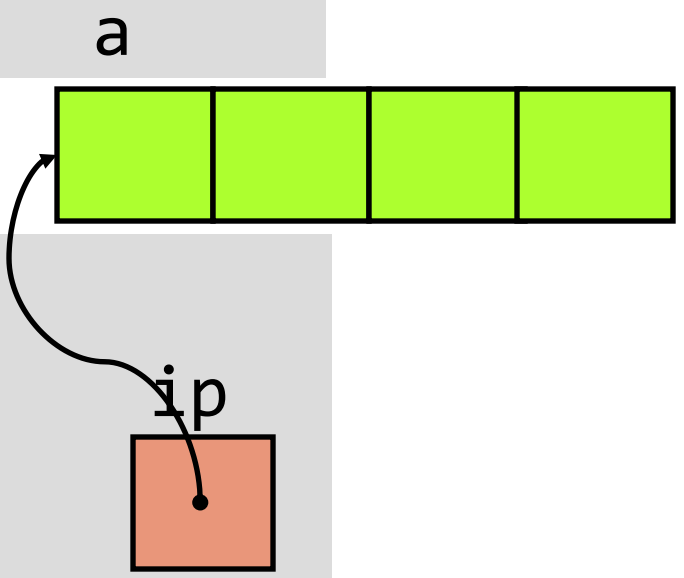
Traversing Arrays Using Pointers

The usual way to iterate over arrays:

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int i = 0; i < len; i++) {
      /* Do something about a[i] */
}
```

Using pointers:

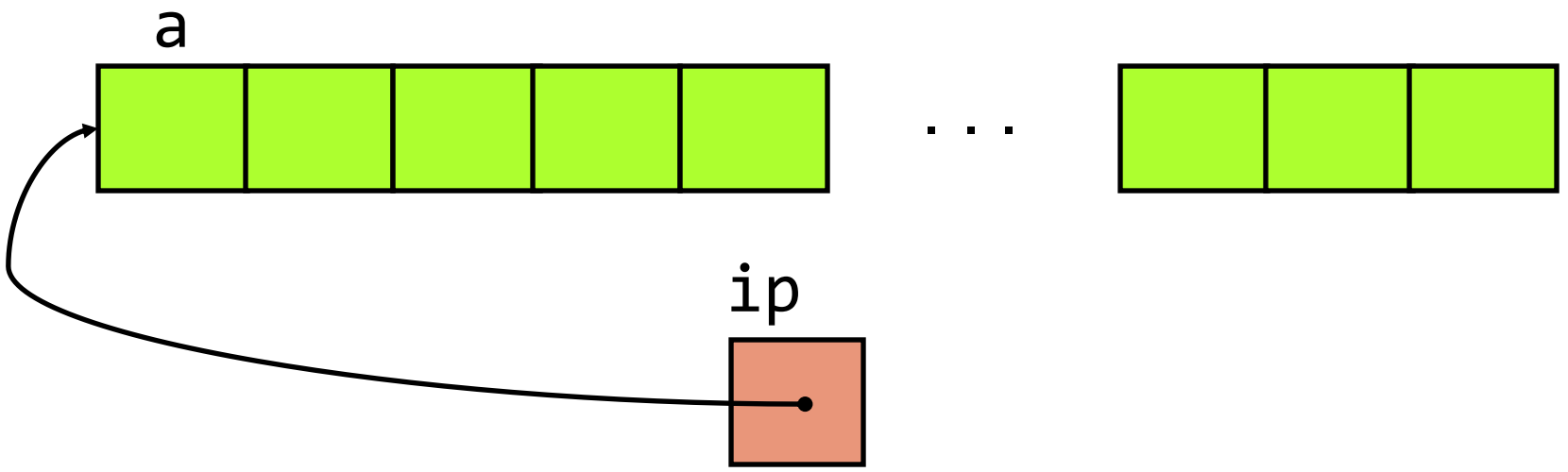
```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```



Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

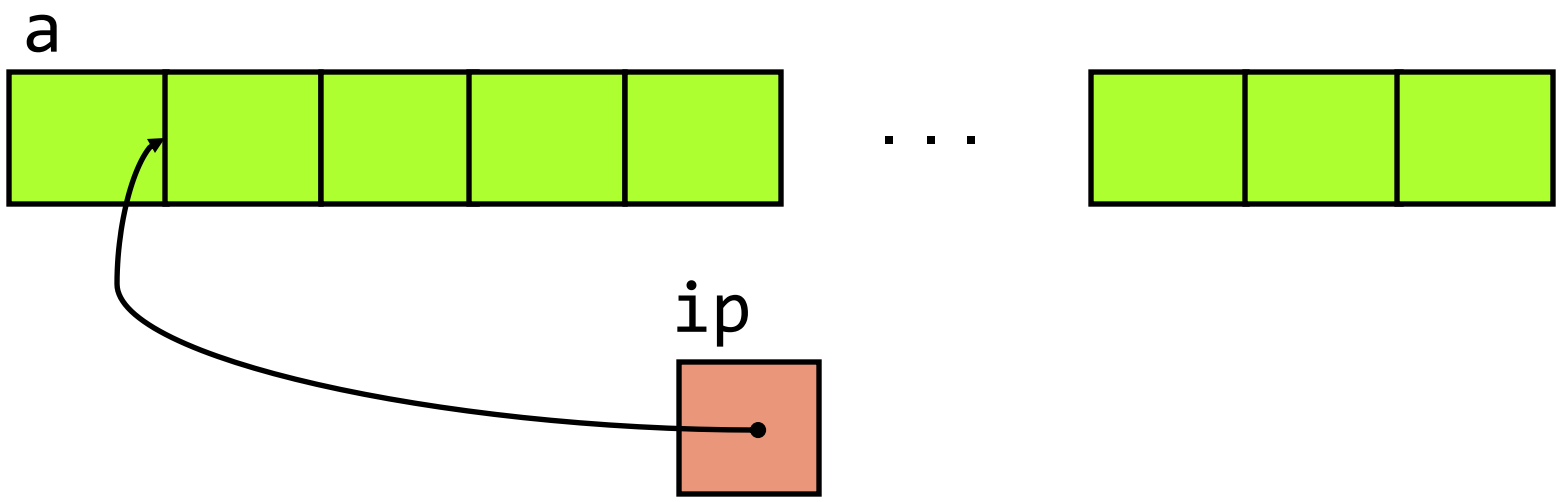
1st iteration:



Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

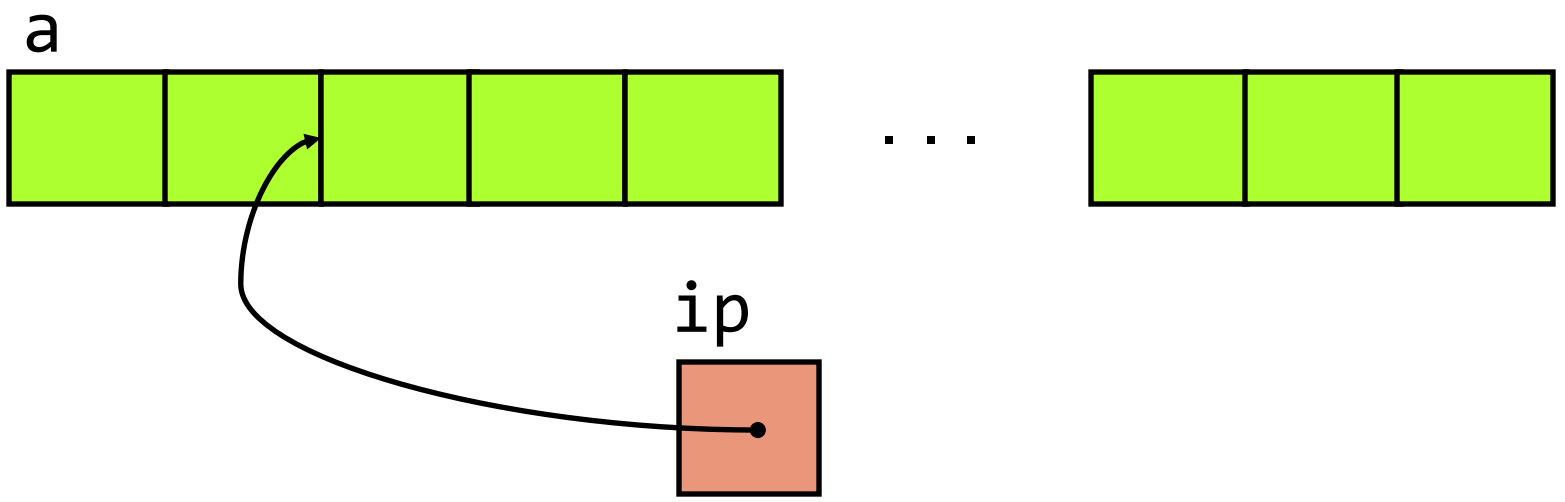
2nd iteration:



Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

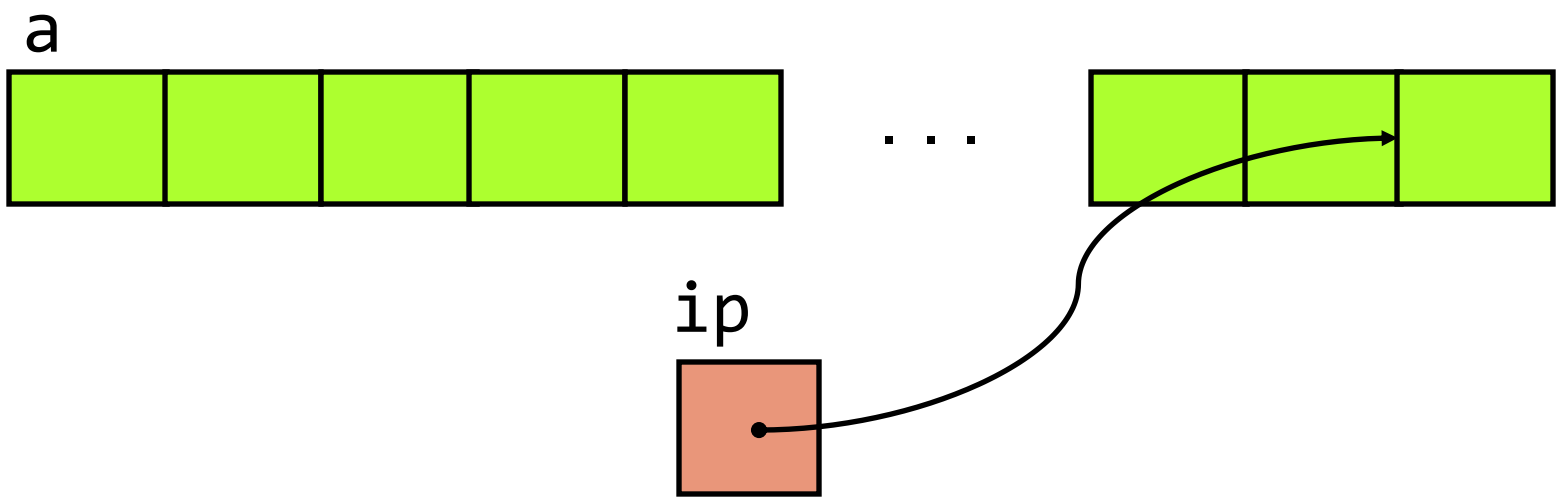
3rd iteration:



Traversing Arrays Using Pointers

```
int a[] = { ... };
int len = sizeof(a)/sizeof(int);
for(int *ip = a; ip < a + len; ip++) {
      /* Do something about *ip */
}
```

(len-1)th iteration:



A Note on Operator Precedence

Slight correction:

These only refer to *prefix* ++ and --

Postfix ++ and -- has level 1 precedence, i.e., the same as (), [], -> and .

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Increment and Indirection Together

- Suppose

```
int *ip;
int i;
```

- What does `i = *ip++` mean?
  - Since postfix ++ has higher precedence than *, the RHS expression evaluates to `*(ip++)` which means

```
i = *ip; ip = ip + 1;
```

Increment and Indirection Together

- Suppose

```
int *ip;
int i;
```

- What does `i = *++ip` mean?
  - Both prefix ++ and * have same precedence, so associativity (right to left) is applied on RHS yielding `*(++ip)` which means

```
ip = ip + 1; i = *ip;
```

Increment and Indirection Together

- Suppose

```
int *ip;
int i;
```

- How to increment the value of whatever `ip` points to?

```
(*ip)++;
```

Pointer Types

- Pointer variables are generally of the same size, but it is **inappropriate** to assign an address of one type of pointer variable to a different type of pointer variable
- Example:

```
int V = 101;
float *P = &V; /* generally results in a warning */
```

  - Warning rather than error because C will allow you to do this (it is appropriate in certain situations)

Casting Pointers

- When assigning a memory address of a variable of one type to a pointer that points to another type, it is best to use the cast operator to indicate the cast is intentional (this will remove the warning).
- Example:

```
int V = 101;
float *P = (float *) &V;
/* Casts int address to float * */
```

  - Removes warning, but is still unsafe to do this !!! You must know what you are doing when casting pointers!

General (void) Pointer

- A `void *` is considered to be a **general pointer**, it can point to any type of pointer variable
- No cast is needed to assign an address to a `void *` or from a `void *` to another pointer type
- Example:

```
int V = 101;
void *G = &V; /* No warning */
float *P = G; /* No warning, still unsafe */
printf("%d", *G); /* Compiler Error */
printf("%d", *((int *)G)); /* No warning, and safe */
```

- Certain library functions return `void *` results

Strings and Pointers

- Recall:
  - A string in C is an array of chars terminated by the null character
  - We can use a pointer to point to an array
- **A char pointer can be used to point to a string**

```
char str[] = "Hello, world";
char *vstr = str;
char *lstr = "Hello, world";
```

`vstr` points to a string variable

`lstr` points to a string literal

```
vstr[0] = 'h';
lstr[0] = 'h';
```

Allowed since `vstr` points to a string variable

Not allowed since `lstr` points to a string literal

Strings ♥ Pointers

```
int strlen (char *s){
  int n;
  for(n=0; *s!='\0'; s++){
    n++;
  }
  return n;
}
```

```
int strcmp(char *s, char *t){
  int i;
  for(i=0;s[i]==t[i];i++){
    if(s[i] == '\0'){
      break;
    }
  }
  return s[i] - t[i];
}
```

```
int strcmp(char *s, char *t)
{
  for(;*s == *t; s++,t++){
    if (*s == '\0'){
       break;
    }
  }
  return *s - *t;
}
```

```
void strcpy(char *s, char *t){
  int i = 0;
  while((s[i]=t[i]) != '\0'){
    i++;
  }
}
```

```
void strcpy(char *s, char *t)
{
  while((*s=*t) != '\0') {
    s++; t++;
  }
}
```

```
void strcpy(char *s, char *t)
{
  while((*s++=*t++) != '\0');
}
```

Notice in the second `strcmp()` and second and third `strcpy()`, the use of pointers to iterate through the strings

The conciseness of the last `strcmp()` and `strcpy()` make them hard to understand

Pointer to Pointer

- A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

- Example:

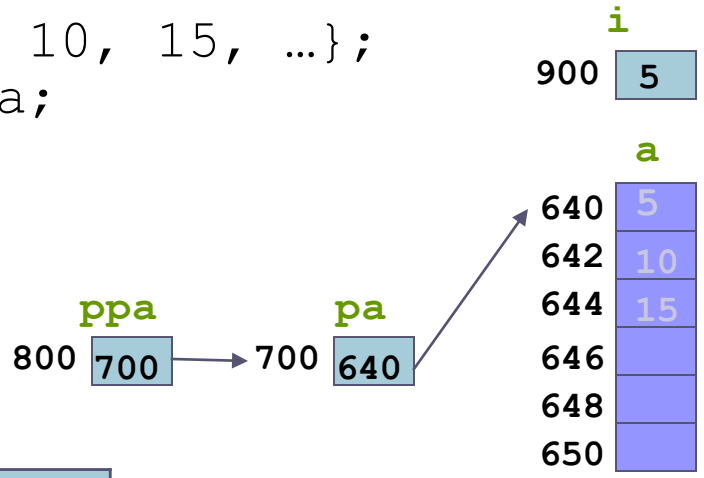
```
int V = 101;
int *P = &V;  /* P points to int V */
int **Q = &P; /* Q points to int pointer P */

printf("%d %d %d\n", V, *P, **Q);
  /* prints 101 3 times */
```

Pointer arithmetic

Assume short is 2 bytes, and pointer variable (address size) is 4 bytes.

```
short a[10]={5, 10, 15, …};
short *pa, **ppa;
int i=5;
pa = a;
ppa = &pa;
```



Variable	Address	Value
a	640	5
	642	10
	644	15
	…	…
pa	700	640
ppa	800	700
i	900	5

Questions:

Expression	Value	Note
pa+1	642	640+1*2
pa+3	646	640+3*2
pa+i	650	640+i*2
*pa+1	6	5+1
*(pa+1)	10	a[1]=pa[1]=*(a+1)
pa[2]	15	644
*ppa	640	Value of pa
*ppa+1	642	pa+1
*(*ppa+1)	10	*(pa+1)
*(ppa+1)	invalid	*(704)

Pointer to an array vs array of pointers

- Array of pointers `int *ptr[3];`
  - An array of the pointer variables

- Pointer to an array
  - `ptr` that points to the 0th component of the array.
  - Pointer can point to whole array rather than just a single component of the array.

```
int a[3] = {3, 4, 5 };
int *ptr = a;
```

```
int (* ptr)[3] = NULL;
```

  - Since square brackets have higher priority than indirection (*), it is crucial to wrap the indirection operator and pointer name inside round brackets.

Pointers and 2-D arrays

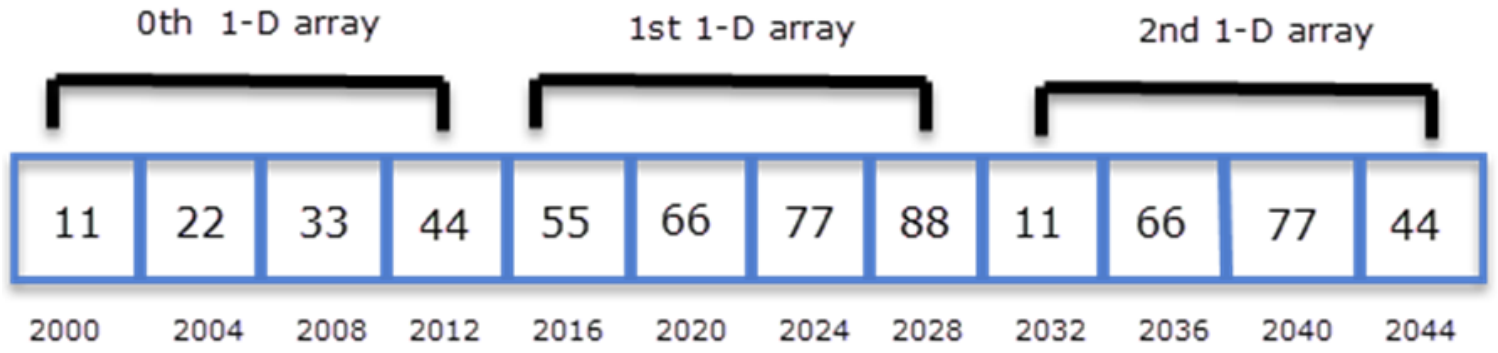
```
int arr[3][4] = {{11,22,33,44},
                 {55,66,77,88},
                 {11,66,77,44}};
```

	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44

- Computer memory is linear and there are no rows and cols.
- A 2-D array is actually a 1-D array
- So `arr` is an array of 3 elements where each element is a 1-D array of 4 integers.

```
int (*p)[4];
```

- Here `p` is a pointer that can point to an array of 4 integers. In this case, the type or base type of `p` is a pointer to an array of 4 integers.

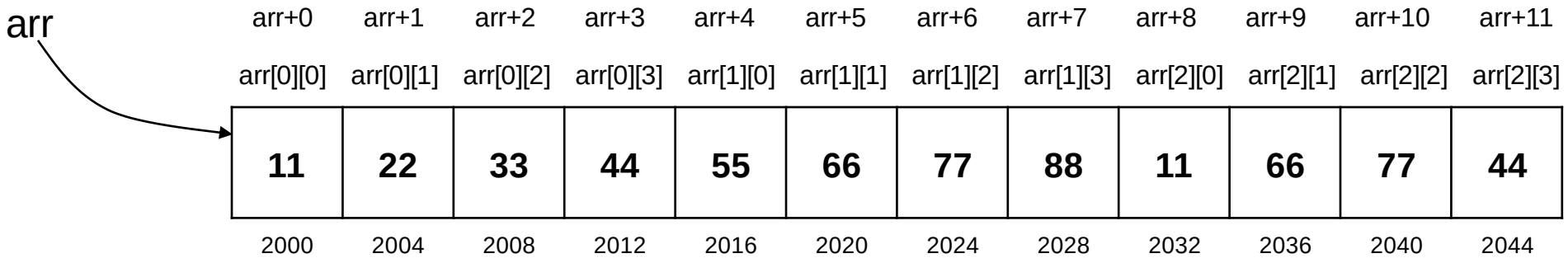


Single Pointer and 2-D arrays

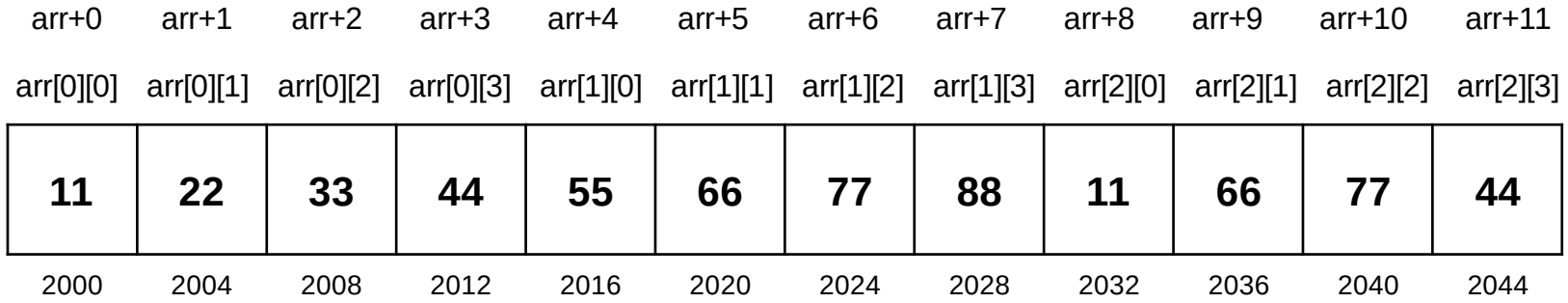
```
int arr[3][4] = {{11,22,33,44},
                 {55,66,77,88},
                 {11,66,77,44}};
```

	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44

- Computer memory is linear and there are no rows and cols.
- Each element can be accessed by a pointer



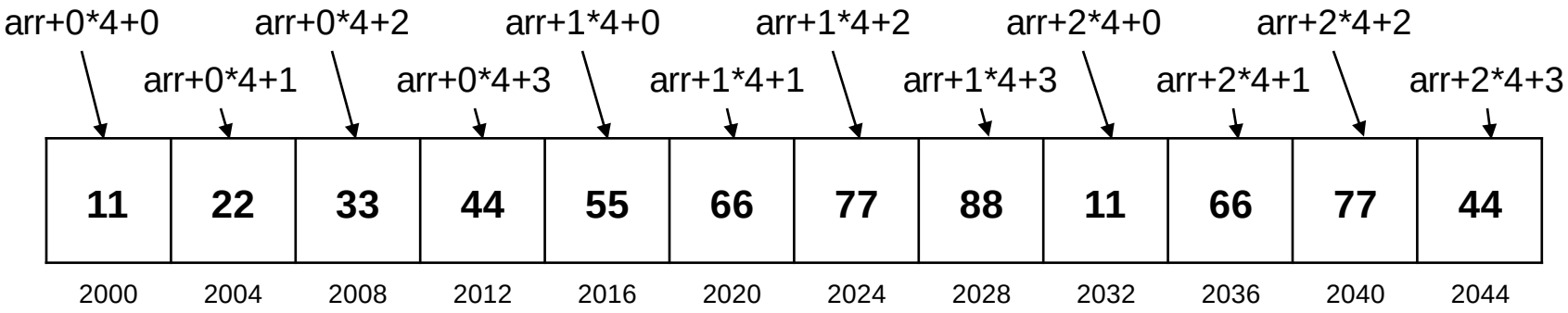
Traversing 2-D Array Using single Pointer



	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44

arr + row * number of columns + column

```
for (i = 0; i < rows; i++){
   for (j = 0; j < cols; j++){
      printf("%d ", *((arr + i * cols) + j) );
   }
}
```



Recap: Usage of Pointers

1) Provide an alternative means of accessing information stored in arrays

2) Provide an alternative (and more efficient) means of passing parameters to functions

3) Enable dynamic data structures, that are built up from blocks of memory allocated from the heap at run time

Passing Function Parameters

- Recall:

Function definition

```
int add ( int a, int b )
{
   return a + b;
}
```

Formal parameters

Actual function call

```
int i = 1, j = 2;
int k = add(i, j);
```

Actual parameters

Call by Value

```
int add ( int a, int b )
{
   return a + b;
}
```

Formal parameters

```
int i = 1, j = 2;
int k = add(i, j);
```

Actual parameters

- The values of **actual parameters** (`i, j`) are copied to **formal parameters** (`a, b`)
  - Actual and formal parameters are separate entities
- What happens thereafter to formal parameters has no effect on actual parameters
  - Any changes on `a, b` will not be transferred back to `i, j`

Example

```
void swap( int a, int b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
int i = 5;
int j = 10;
swap( i, j );
printf("%d %d", i, j);
```

Output

```
5 10
```

- After call to `swap()`, `i` and `j` values remain unchanged
- During execution of `swap()`, the copies of `i` and `j` are swapped inside the function, but not `i` and `j` themselves!

The Solution: Call by Reference

- Pass a copy of the address to the function
- **Both formal and actual parameters refer to the same address**
- This can be done using pointers as function parameters

```
void swap( int *a, int *b )
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

```
int i = 5;
int j = 10;
swap( &i, &j );
printf("%d %d", i, j);
```

Output

```
10 5
```

Example: Passing 2D Arrays to Functions (1)

```
#include <stdio.h>
void view_array(int m, int n, int a[][3]){
    int i, j;
    for (i = 0; i < m; i++){
      for (j = 0; j < n; j++){
        printf("%d ", a[i][j]);
      }
    }
}

int main(){
    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;

    view_array(m, n, arr);

return 0;
}
```

- Passing an **entire array** to a function
  - When passing an array as an argument to a function, it is passed by its memory address (starting address of the memory area) and not its value(call-by-address)!
  - Because a function accesses the original array values, we must be very careful that we do not inadvertently (accidentally) change values in an array within a function.

Example: Passing 2D Arrays to Functions (2)

```
#include <stdio.h>
void view_array(int m, int n, int (*a)[3]){
    int i, j;
    for (i = 0; i < m; i++){
      for (j = 0; j < n; j++){
        printf("%d ", a[i][j]);
      }
    }
}

int main(){
    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;

    view_array(m, n, arr);

return 0;
}
```

- What kind of pointer is passed down to the function?
  - The answer is, a pointer to the array's first element. And, since the first element of a multidimensional array is another array, what gets passed to the function is a *pointer to an array*.
  - `int (*a)[3]`
- Using index
  - `A[row][col]`

Example: Single Pointer and 2-D arrays

```
#include <stdio.h>
void view_array(int m, int n, int *a){
    int i, j;
    for (i = 0; i < m; i++){
      for (j = 0; j < n; j++){
        printf("%d ", *((a + i * n) + j));
      }
    }
}

int main(){
    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;

    view_array(m, n, (int *)arr);

return 0;
}
```

- We must typecast the 2D array when passing to function
- Using single pointer
  - `*(a + row * number of columns + col)`

Function Returning a Pointer

Function Returning a Pointer

- Functions can return a pointer
- **Make sure that returned pointer points to a valid memory location**

```
float *find_max(float A[], int N)
{
   int i;
   float *the_max = &(A[0]);
   for (i = 1; i < N; i++)
      if (A[i] > *the_max) the_max = &(A[i]);
   return the_max;
}

int main(void)
{
   float scores[5] = {10.0, 8.0, 5.5, 2.0, 4.1};
   float *max_score;

   max_score = find_max(scores, 5);
   printf("%.1f\n",*max_score);
   return 0;
}
```

Pointer to function

Pointer to function

- Function code is stored in memory
  - Functions also occupy memory locations therefore every function has an address just like variables
  - Just like ordinary variables, the address of a function refers to its starting address

- C does not require that pointers only point to data, it is possible to have **pointers to functions**

Defining a function pointer

- Declaration:

```
return_type (*name)(param_types);
```

- Examples

```
int (*f)(int, float);
```

Pointer to a function that takes an `int` and `float` arguments, resp., and returns an `int`

```
int *(*f)(int, float);
```

Pointer to a function that takes an `int` and `float` arguments, resp., and returns a *pointer* to `int`

Using a function pointer

```
int F1(int i, float f)
{
        return i/f;
}

int main(void)
{
        /* f is a function pointer */
        int (*fp)(int, float);

        /* Assignment: let f point to F1 */
        fp = &F1; /* fp = F1; is also ok */

        /* Invocation */
        float a = fp(1, 2.0);
        /* This is equivalent to calling
        float b = F1(1, 2.0) */
        printf("a = %f, b = %f", a, b);
}
```

Comparing function pointers

- Can use the equality (==) operator

- Example:

```
/* f is a function pointer */
int (*fp)(int, float);

int F1(int i, float f)
{
        return i/f;
}

int main(void)
{
        /* Assignment: let f point to F1 */
        fp = &F1; /* fp = F1; is also ok */

        if(fp == &F1)
                printf("Points to F1\n");
}
```

Safety concerns

- What if uninitialized function pointer value is accessed
  - Safest outcome: memory error, and program is terminated
  - But what if the "garbage" value is a valid address?
    - Worst case: address contains program instruction –execution continues, with random results
    - Hard to trace the cause of the erroneous behavior

Usage of function pointers

- For implementing callback functions
  - Function pointer is passed as an argument to a function
  - The function will then invoke the passed function pointer at a given time

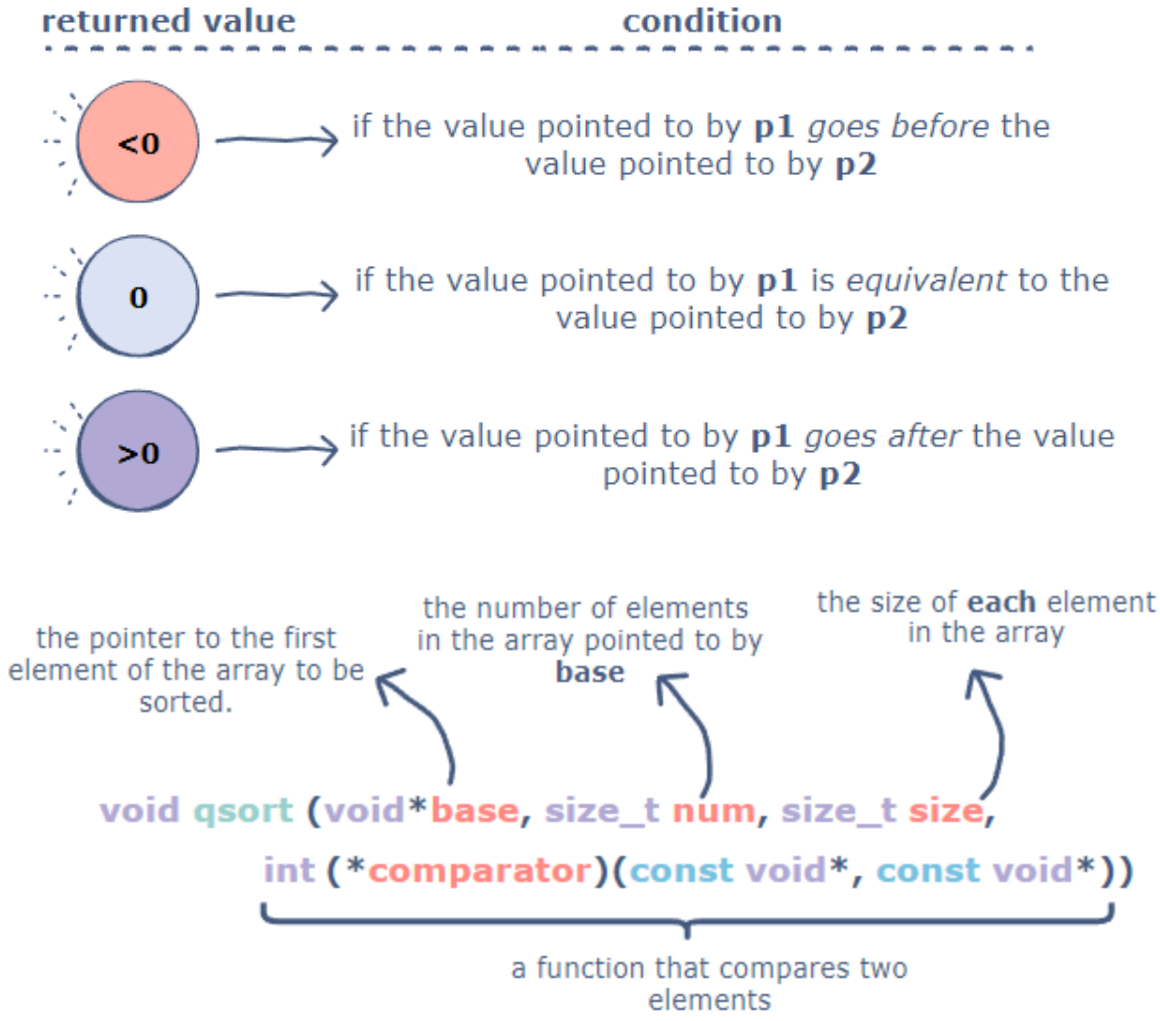
```
void qsort(void *base, size_t nitems, size_t size,
int (*compare)(const void *, const void*));
```

`base` – Pointer to the first element of the array to be sorted
`nitems` – Number of elements in the array pointed by base
`size` – Size in bytes of each element in the array.
`compare` – This is the function that compares two elements.

Function pointers: qsort example

```
int arr[] = {20, 15, 36, -8, 2, 7};

int comparator (const void * p1, const void * p2)
{
  return (*((int*)p1) - *((int*)p2));
}
int main ()
{
  int size = sizeof(arr) / sizeof(arr[0]);
  printf("The unsorted array is: \n");
  for(int i = 0; i < size; i++)
  {
    printf("%d ", arr[i]);
  }
  qsort(arr, size, sizeof(int), comparator);
  printf("\nThe sorted array is: \n");
  for(int i = 0; i < size; i++)
  {
    printf("%d ", arr[i]);
  }
}
```



Next lecture

- Structures