Chapters 10 & 11

MEMORY DIAGRAMS OF VARIABLES INCLUDING ARRAYS & POINTERS

Diagrams of Basic Data Types

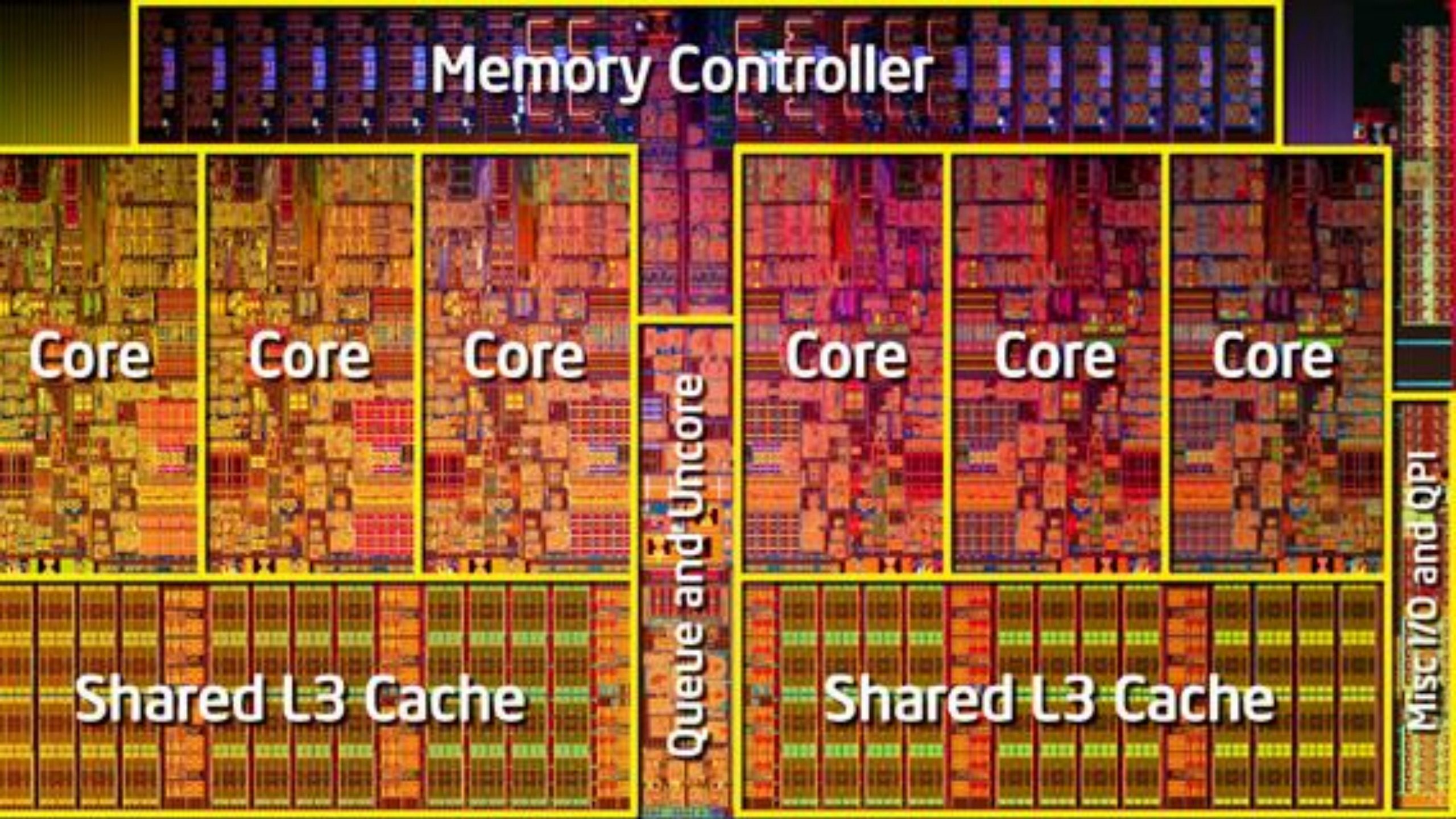
CHAPTER 10

Basic Types in C

C has many built-in types. These include:

- int and unsigned int
- short and unsigned short
- long and unsigned long
- char, signed char, and unsigned char
- bool (since C-99)
- Float, double, and long double

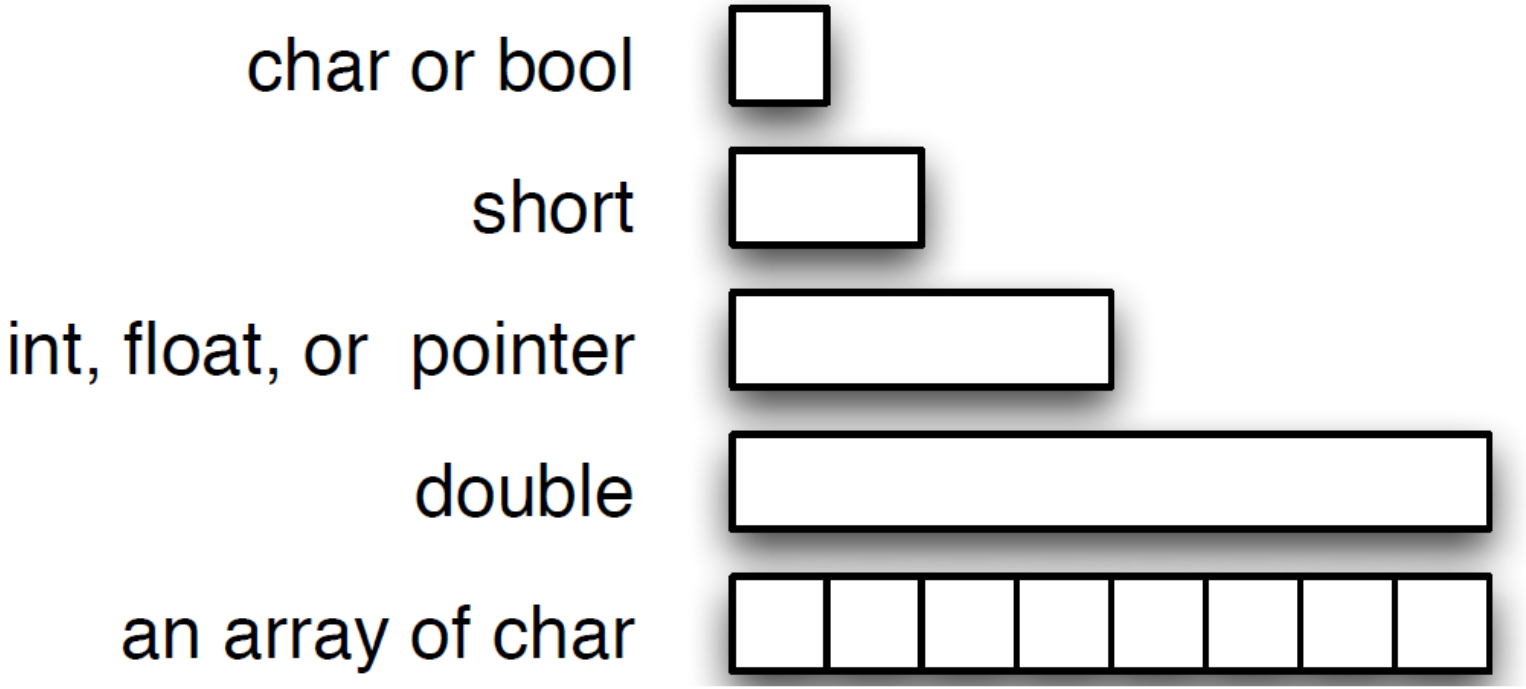
AS we discussed last week, Chapter 7 lists the range of values that can be stored in each type and gives the limits of precision for the floating types.


Memory Controller
Core
Core
Core
Core
Core
Core
Queue and Uncore
Misc I/O and QPI
Shared L3 Cache
Shared L3 Cache

Diagrams of the Basic Types in RAM

In these diagrams, the size of the box is proportional to the number of bytes needed to store a value.

Arrays show a space (slot) for each value.

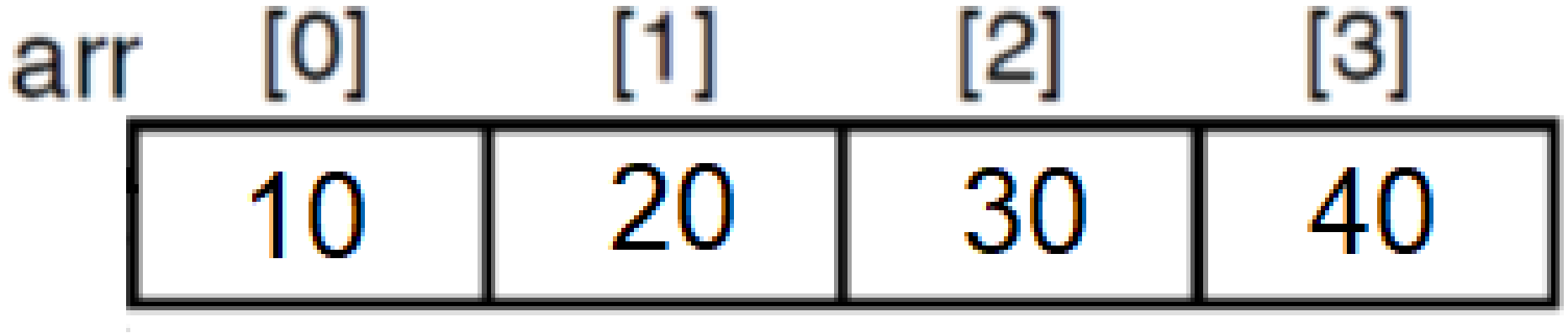


An Array is a complex data type

An **array** consists of **two parts**:

- the **name,** which is bound to the beginning of the array
- and a **set of values** (the slots of the array).

```
int arr[4] = {10, 20, 30, 40};
```



Array Information

- **The name of an array** is translated to the address of the array, which is the same as **slot 0, the head of the array**.
- The slot 0 of an array is guaranteed to be at a lower address than slot 1.

Accessing the slots of an array using [n]

You can get the value from an element of your array by using the square bracket notation. For example:

```
for (short n = 0; n < 4; ++n){
     printf("%i, ", arr[n]);
}
```

Console output:

```
10, 20, 30, 40,
```

sizeof() tells you the **number of bytes in a variable**

```
cout << "What is the size of each basic type in bytes?" << endl;

cout <<"int is " << sizeof(num) << " bytes\nYou can also check by type,
\ne.g. this is short " << sizeof(short) << " bytes" << endl;

cout <<"float is " << sizeof(amt) << " bytes" <<endl;

cout <<"double is " << sizeof(total) << " bytes" <<endl;

cout <<"char is " << sizeof(ID) << " bytes" <<endl;

cout << "bool is " << sizeof(bool) << " bytes" << en
```

```
What is the size of each
basic type in bytes?
int is 4 bytes.
You can also check by type,
e.g. this is short 2 bytes
float is 4 bytes
double is 8 bytes
char is 1 byte
bool is 1 byte
```

sizeof() tells you the **total bytes in an array**

```
cout << "\n++++++++++++++++++++++++++++++++++++++++++++++++++++++++" << endl;

cout << "Arrays are complex, using sizeof() requires awareness of what the answer represents" << endl;

cout <<"The array of ints size is " << sizeof(prices) << " bytes" <<endl;

cout <<"The first slot in the array of ints size is " << sizeof(prices[0]) << " bytes" <<endl;

cout <<"The array of chars size is " << sizeof(list) << " bytes" <<endl;

cout <<"The first slot in the array of chars size is " << sizeof(list[0]) << " bytes" <<endl;
```

```
++++++++++++++++++++++++++++++++++++++++++++++++
Arrays are complex, using sizeof() requires
awareness of what the answer represents
The array of ints size is 40 bytes
The first slot in the array of ints size is 4 bytes
The array of chars size is 10 bytes
The first slot in the array of chars size is 1 bytes
```

What does sizeof() tell you about **strings**?

```
cout << "\n++++++++++++++++++++++++++++++++++++++++++++++++++++++++" << endl;

//Strings are even more complex,

//sizeof() does not tell you what you probably want to know."

cout <<"The size of the string is " << sizeof(descr) << " bytes" <<endl;

descr = "Even if you change the contents of the string, sizeof() returns the same size. Think about why. How many bits is 8 bytes?";

cout << descr << "\nNow this string size is " << sizeof(descr) << " bytes" << endl;

cout << "Sting is a class and it has its own internal functions to tell you what it contains." << endl;
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++
The size of the string is 8 bytes
Even if you change the contents of the string, sizeof() returns the same size. Think
about why. How many bits is 8 bytes?
Now this string size is 8 bytes
Sting is a class and it has its own internal functions to tell you what it contains.
```

Working with Arrays – What Info is needed?

- **You can** use sizeof() to find the number of slots in an array, but it takes some skill. The **number of allocated elements** is

$$\frac{sizeof(arrayName)}{sizeof(type)}.$$

- **You CANNOT find the number of OCCUPIED elements** in your array.
- At a minimum**, you need to keep track of the number of occupied elements** in a **separate variable**.
- If you are using an array in a function – to print it, to put data in it, to process it - **you must pass the name of the array and the number of occupied elements**. You can also pass the size of the array to make the program simpler to understand.

Parallel Arrays – two or more separate arrays with associated data

Parallel Arrays were very common before struct and class came along to solve the problem of needing to associate data of different types.

Each array is like a column in a spreadsheet.

All the data on the same row goes together.

Row	Name	Grade	Age	State	GPA
0	Mary	92.5	19	CT	3.8
1	George	78.75	18	MA	2.7
2	Rachna	80.0	21	IN	3.2

Working with Parallel Arrays

```
int prices[5] = {1,2,3,5,8};
char inits[5] = {'M', 'E', 'R', 'J', 'L'};

for(short i = 0; i<5; ++i){
    cout << "Position "<< i << " contains " << inits[i]
         << " prices "  << prices[i] << endl;
}
```

```
Position 0 contains M prices  1
Position 1 contains E prices  2
Position 2 contains R prices  3
Position 3 contains J prices  5
Position 4 contains L prices  8
```

Pointers
Diagrams of pointers
Pointer operations

CHAPTER 11

Pointers are Useful

- A **pointer variable** is a variable that stores an address.
- A pointer contains **a memory address (in your allocated area of RAM)**. It has **star-level 1.**
- A variable that **contains a value**, such as an int. It has **star-level 0**. The **address of the simple variable, &n, has star-level 1.**

Why bother with pointers?

- Pointers are used **to process arrays or files**.
- Pointers are used **to return answers from functions**.

Pointer Declaration Syntax and Usage

- In the **declaration**, **the * is part of the type,** not part of the variable name.

```
int* p;
```

- **Avoid declaring more than one pointer in a declaration**, since the * must be repeated for each variable, it is too easy to forget.

Using * and & operators with pointers

- If you write `p =` , you will **change the contents of p, the right side must be an address, star level 1**.
- If you write `*p =` **you will change the value stored at the address** of the variable p points at. The **right side must be a data value** of the type the pointer points at, **star level 0**. e.g. an integer.
- If you write `p = &num`, you are **extracting** the underlying **address of the variable** and **storing it as the value stored in the pointer variable, star level 1**.
- The **star level on both sides of the assignment operator,** =, must be the **same**.

Pointer Syntax and Usage

```
int num = 25;
int* p1 = arr; // previously: int arr[4] = {10, 20, 30, 40};
int* p2 = arr[2];
```

- In the **code:**
  - ***p references the contents of the variable** that `p` points at.
    - `*p1 = *p2;` **//changes the contents of the element of the array that p1 points at to the contents that p2 points at;** arr[0] is changed to 30.
  - **p means the address** that `p` contains.
    - `p1 = p2;` **// changes the address** that p1 contains; P1 now also points to arr[0].
  - **&num means the address of the variable `num`.**
    - `p1 = &num;` **//changes the address that p contains to the address of num into p1; p1 now points at num, not the array.**

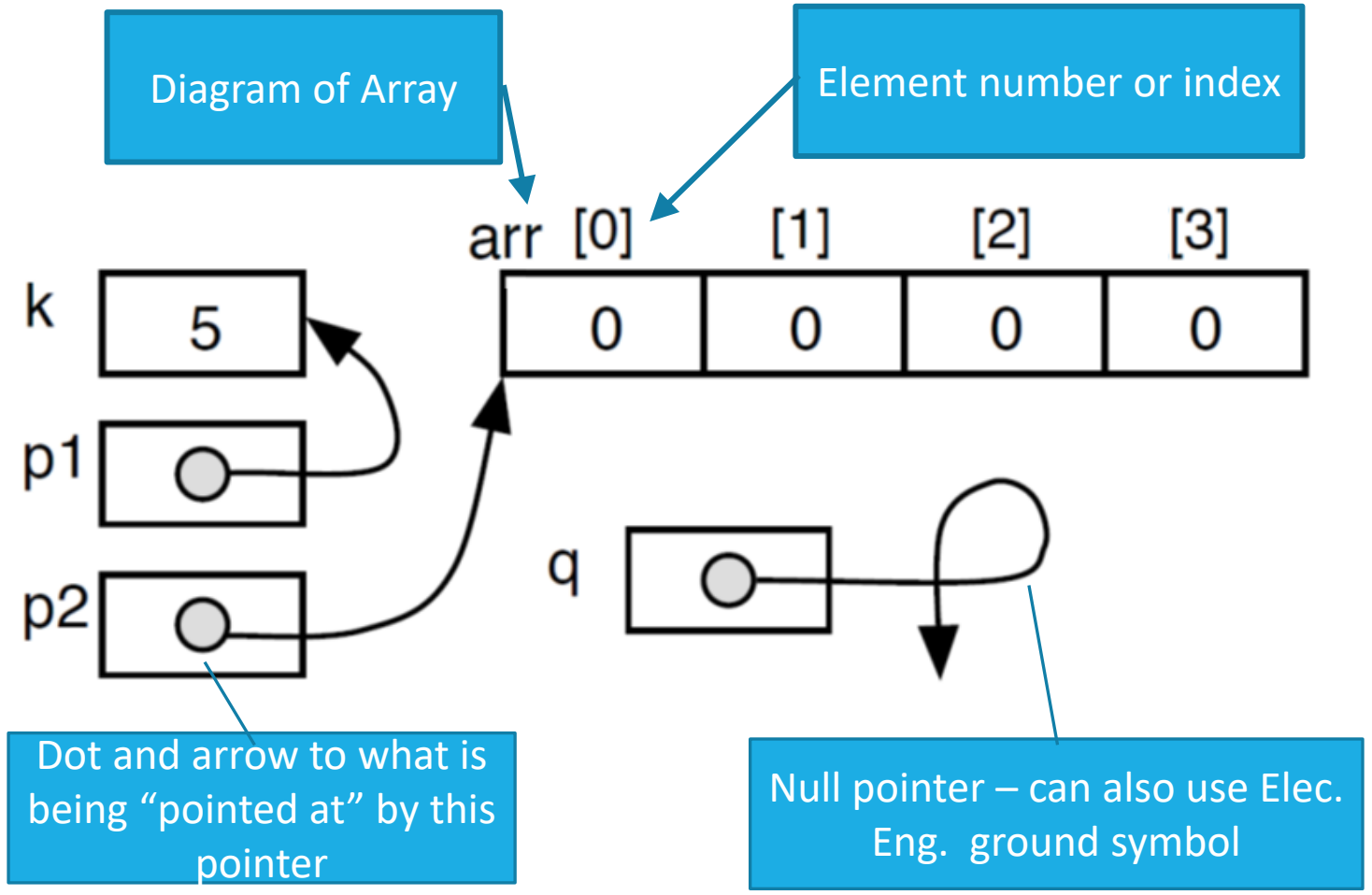
Declaring Pointers

```
int k = 5;
int arr[4] = {};
int* p1 = &k;
int* p2= arr;
int* q = nullptr;
```

NOTE:

```
int* p2 = arr;
int* p2 = &arr[0];
// Have the SAME result.
```

Pointing to the start of the array arr!



Working with

```
Using Variables
35 is age
0x7ffcfc7cd8d4 is the address of age
0x7ffcfc7cd8d4 agePtr contents is the address of the var age
35 dereferencing agePtr, which gives me the contents of age
0x7ffcfc7cd8c8 Gives me the address of agePtr, which is a
different address
Bye!
```

```
int age = 35;
int* agePtr = &age;

cout << "Using Variables" << endl;
cout << age << " is age" << endl;
cout << &age << " is the address of age" << endl;
cout << agePtr << " agePtr contents is the address of the var
age" << endl;
cout << *agePtr << " dereferencing agePtr, which gives me the
contents of age" << endl;

cout << &agePtr << " Gives me the address of agePtr, which is a
different address" << endl;

cout <<"Bye!" << endl;
```

Pointer Arithmetic

- **Pointer arithmetic** works in **units of array elements, not bytes**.
- The following things make sense **ONLY if pointers p and q are pointing into the same array**.
  - **Add int N to p to move** p N elements toward end of the array.
  - **Subtract q-p** to **find the # of array elements between** p and q
  - **Compare p==q** to find if they **point to the same array element (address)**
  - **Compare p > q** to learn **which pointer references the larger subscript**
  - You can have a **pointer to the first byte that is NOT in the array** (past the array). This is **useful to end a loop**. Add the number of elements in the array to the array name; e.g `q = arr + 4;`

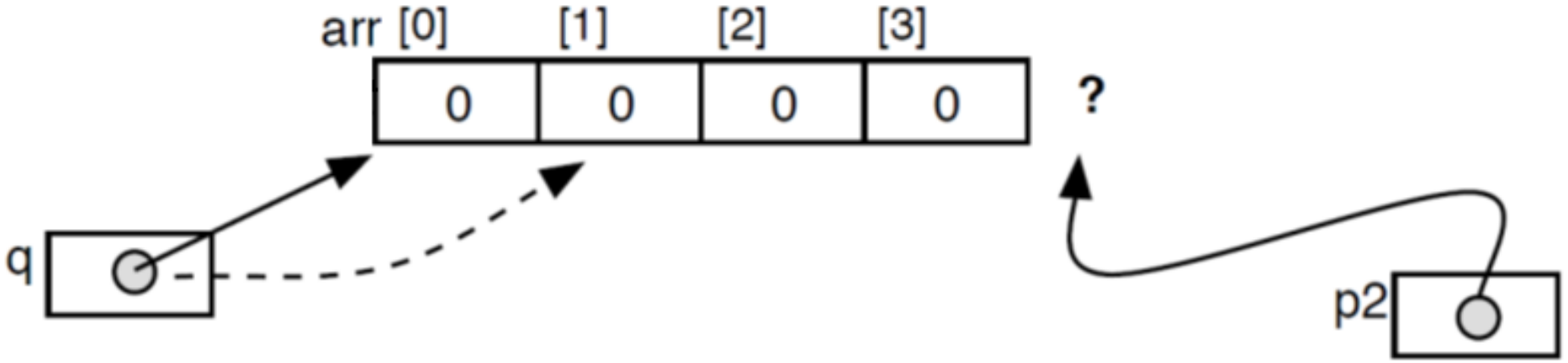
Usage: array processing

Pointers are often used to process arrays. Here is how:

- Establish a single name that represents the **beginning of the array**, the **head pointer**, either the **name of the array** you used when you declared it, or the **name of the pointer** in which you store the result of a new dynamic allocation. Later, this is the name to use when you free the dynamic memory.
- Create an **end pointer** to point at the **first address that is not in the array**. Use this to end loops.
- Create a **scanning pointer** that **starts at the head and goes to the tail**.

Diagram of Pointers in memory

```
int arr[4] = {}; //empty array with 4 elements
p2 = arr+4; // Sentinel or tail pointer, points to next byte after the array.
q = arr; // Cursor or scanner, will change during the loop, starts at head.
for (q=arr; q<p2; q++). . .
```



Processing an Array Sequentially in C

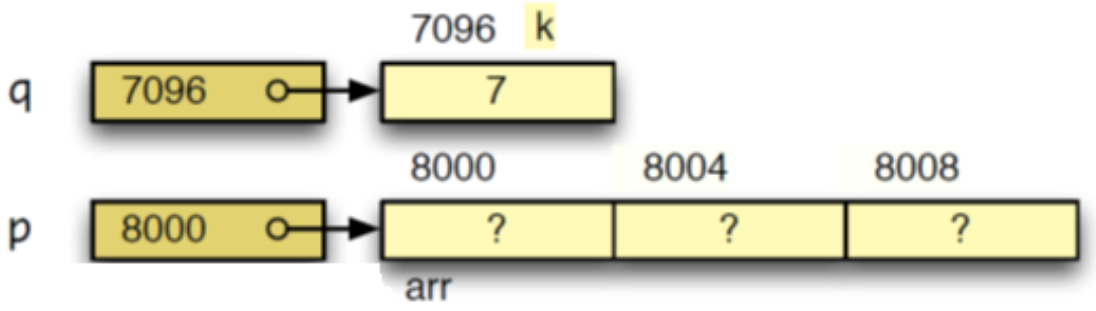
```
// process the whole array.
int k = 5;
int arr[4] = {10, 20, 30, 40};
int* p2= arr + 4; // sentinel
int* q = NULL; //cursor set to null, use ‘nullptr’ in C++
for (q=arr; q<p2; ++q) {
   printf(“%i, “,*q); //prints the element the cursor q points at
}
printf(“Bye!”);
```

Using pointers

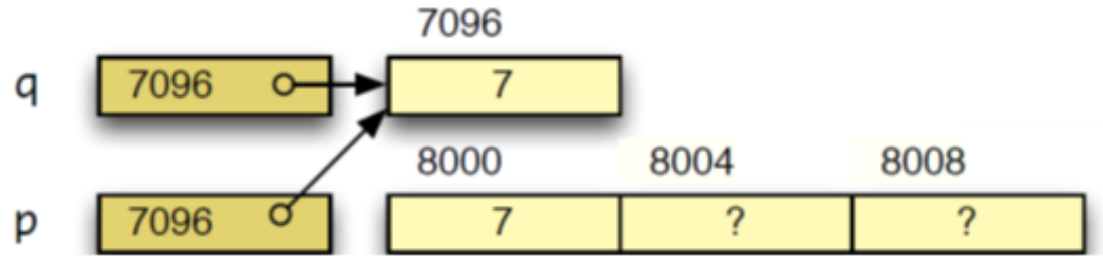
`p =` **changes the contents of p, which is an address. No asterisk.**

`*p =` **changes the value stored at the address p points at. Has an asterisk.**

```
int k = 7;
int arr[3];
int* p = arr;
int* q = &k;
```




```
*p = *q;
p  =  q;
```



How Pointers Work, the details

How and why pointers are used to **return answers from functions?**

- **Functions can only return one value**.
- **Passing a pointer as a parameter allows you to access the RAM allocated to the calling function** and **store the answer in that memory location**.
- When **control of the program returns to the calling function**, the **answer is in a variable that the calling function can access**.

Array Parameters

- An **array argument is ALWAYS passed by address**. This saves time and space.
- **In the function call**, **use the array name alone**, without any & or subscript.
- **In the function**, **declare the parameter to be a pointer or an array**; both work the same way.
- In the function, **you can use subscripts, even if the parameter is a * type.**
- The **length of the array is NOT passed to the function as part of the array**, so it **must be passed separately in an int**. If you declare an array parameter with a length, the length will be ignored.

Pointer Parameters

- Using a **parameter that points** at the calling functions memory allows you write the answer in **the calling program's variable**.

```
int myFunction(double* db){
```

- This is the **right way to share memory** in your program.
- For example, to share a double, you should declare the parameter to be:

```
double* db;
```

- Inside the code, near the end, you would write

```
*db = answer;
```

Processing an Array in a C Function

Pointers are used to process the array in the function and the resulting changes can be seen in main().

```
19 int main(){
20     int myArr[4] = {10, 20, 30, 40};
21     myFunction(myArr, 4);
22
23     puts("\n\nBack in Main: ");
24     for(int n = 0; n < 4; n++){
25         printf("myArr[%i] now contains %i\n", n, myArr[n]);
26     }
27     //TODO NOTE: printf("%i", n); // n is out of scope here, n does not exist.
28
29     puts("Bye!");
30     return 0;
31 }
32
34  * myFunction(int*, int) accepts two arguments, a reference to
39 int myFunction(int pHead[], int numSlots){  //int* pHead or int pHead[] works
40     int* pSentinel = pHead + numSlots;
41     int* pCursor;
42     puts("In the function:");
43     for(pCursor = pHead; pCursor < pSentinel; pCursor++){
44         printf("ary[%i] contains %i\n", pCursor - pHead, *pCursor);
45         *pCursor *= 12;  //Multiply contents times 12
46     }
47     return 0;
48 }
```

```
#include <iostream> // Including IO Library
using namespace std; // Namespace call
void foo(string s, int* num); //function prototype
/* Main Function */
int main() {
int ages[4] = {21, 18, 32, 65};
int* cursor; // ages or &ages[0] would be the same address
int* sentinel = ages + 4;

 for(cursor = ages; cursor < sentinel; cursor++ )
    cout << *cursor <<", ";

 cout << endl;
foo("Bye! ", ages);
return 0;
}
//noot noot
// A peer-reviewed publication
void foo(string s, int* num){
    cout << s << *num << endl;
}
```

A Simple C++ Function using Pointers

Console Output:
21, 18, 32, 65,
Bye! 21

The End