

## Statically-allocated Variables

- Size is fixed throughout execution
- Size is known at compile time
- Space/memory is allocated at execution

## Dynamically-allocated Variables

- Created during execution
  - † "dynamic allocation"
- No space allocated at compilation time
- Size may vary
  - † Structures are created and destroyed during execution.
- Knowledge of structure size not needed
- Memory is not wasted by non-used allocated space.
- Storage is required for addresses.

On modern computers, memory is organized in a manner similar to a one-dimensional array:

- memory is a sequence of bytes (8 bits)
- each byte is assigned a numerical address, similar to array indexing
- addresses are nonnegative integers; valid range is determined by physical system and OS memory management scheme
- OS (should) keep track of which addresses each process (executing program) is allowed to access, and attempts to access addresses that are not allocated to a process should result in intervention by the OS
- OS usually reserves a block of memory starting at address 0 for its own use

## Pointer Type

- Simple type of variables for storing the memory addresses of other memory locations

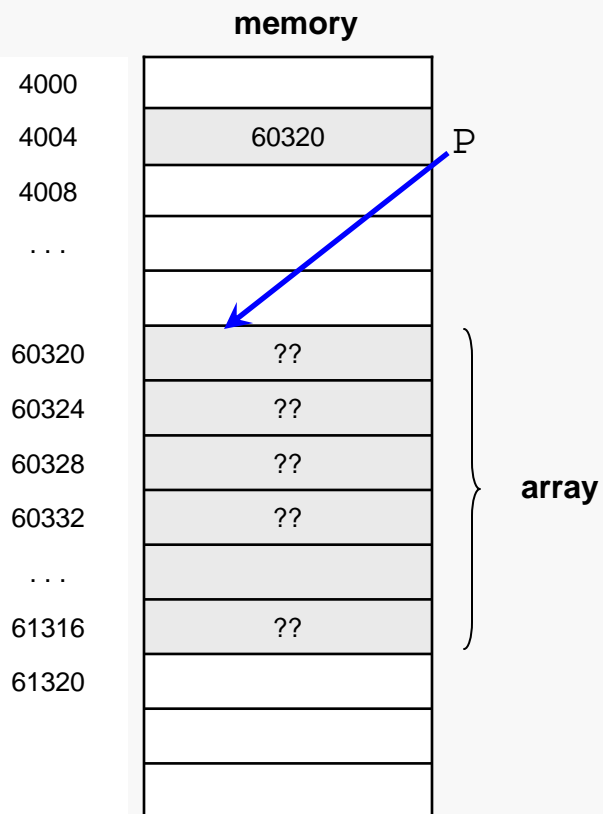
## Declaring Pointer Variables

- The asterisk '\*' character is used for pointer variable declarations:

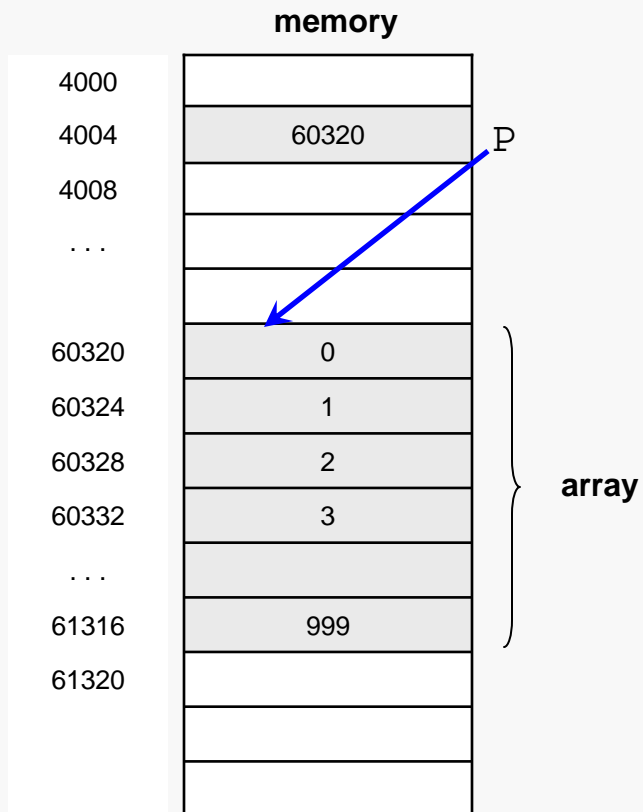
```
int      *P;    // can point to an integer value
double   *Q;    // can point to a double value
```

# Allocating an Array Dynamically

```
int Size = 1000;  
  
int *P;    // can point to an integer value; uninitialized  
  
p = new int[Size];    // creates an array of 1000 integers;  
                      // array cells are not initialized;  
                      // p stores address of first array cell
```

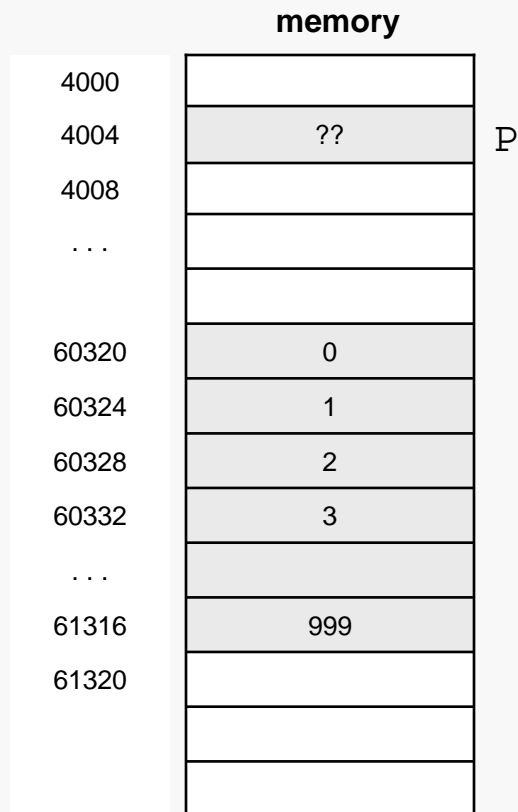


```
for (int Idx = 0; Idx < Size; Idx++) {  
    P[Idx] = Idx;           // same access syntax; just use the  
                           // pointer variable the same way you  
                           // would use an array name.  
}
```



# Deallocating the Array

```
delete [] P; // returns the memory for the array to the OS;  
             // when you're done with it, you must do this  
             // to return control of the memory so the OS can  
             // use it for other purposes
```



A pointer can also be used to point to just a single value:

```
int *Q;           // Q has random value; no logical target

Q = new int;     // Q has a target; target has random value

*Q = 42;         // target of Q has value 42;
                // * operator is used to access the target
                // of the pointer

cout << *Q;      // writes value 42 to console

delete Q;        // memory returned to OS; Q has no target;
                // note difference in syntax from array use
```

Trying to use the target of a pointer, when the pointer has no target, will cause a run-time error:

```
int *Q;           // Q has random value; no logical target
*Q = 42;         // Q has no target; this blows up
```

There's no way to ask "does this pointer have a value?"

But, we can use a special value, NULL, to indicate that:

```
int *Q = NULL;    // Q has no target; but has value NULL
if ( Q != NULL ) // we can check to see if Q is NULL
    *Q = 42;      // now we know this is safe
```

NULL is a special constant, defined to have the value 0.

Pointers to struct variables have a special syntax for accessing the members:

```
struct SkyDiver {  
    string Name;  
    double Weight;  
    double dragK;  
    double timeFalling;  
    double Speed;  
};
```

```
SkyDiver *pS = new SkyDiver;    // pS has an uninitialized target  
  
pS->Name = "George H W Bush";  // "arrow" operator lets you access  
                                // members of the target
```