Chapter Seven: Pointers, Part I

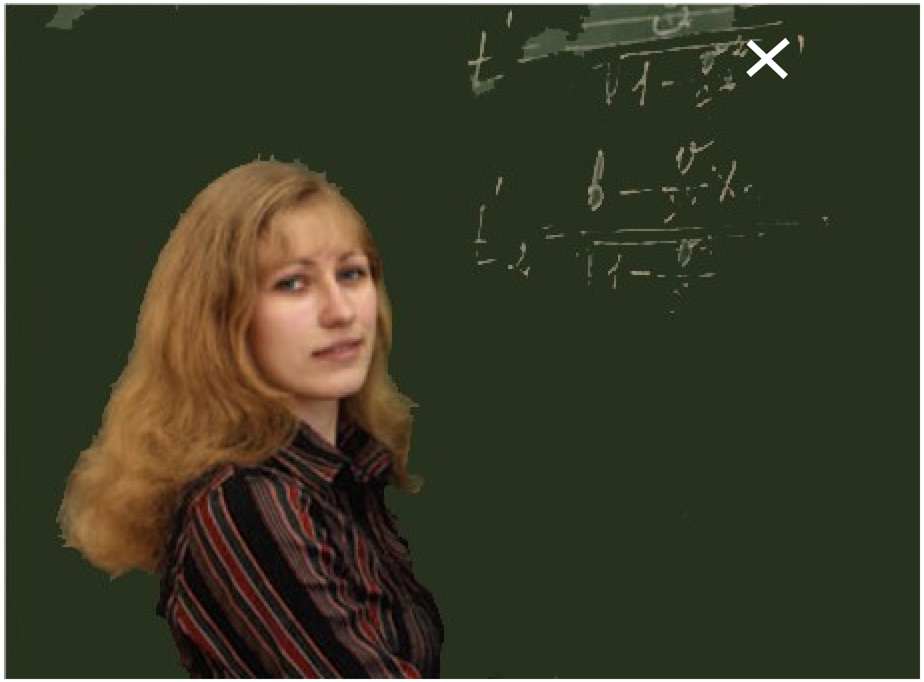
Slides by Evan Gallagher

C++ for Everyone by Cay Horstmann

Chapter Goals

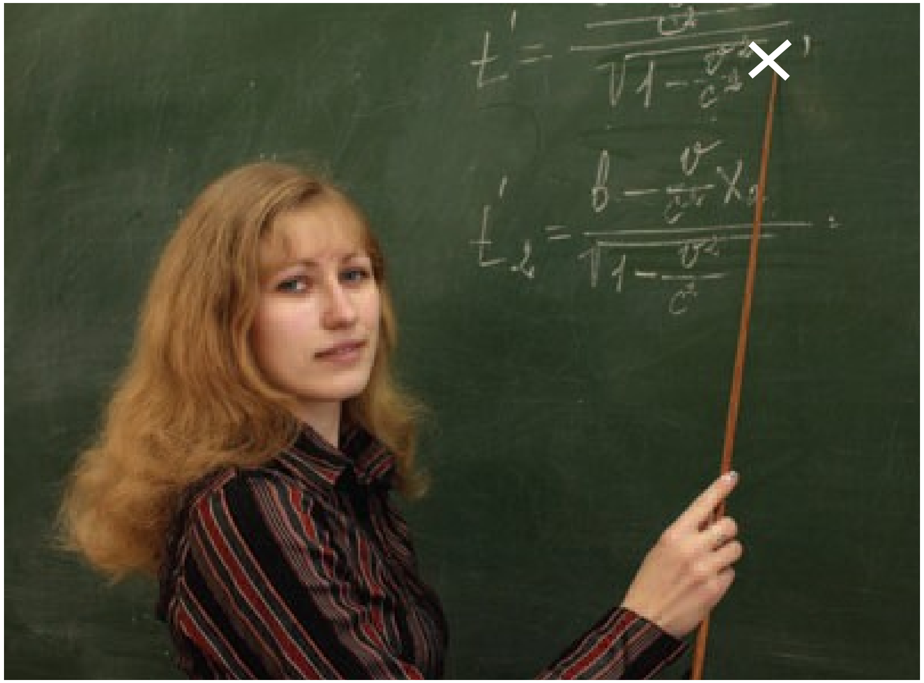
- To be able to declare, initialize, and use pointers
- To understand the relationship between arrays and pointers
- To become familiar with dynamic memory allocation and deallocation

Pointers



What's stored in that variable?

Pointers



that one – the one I'm ***pointing*** at!

Pointers

A variable *contains* a value,

but a *pointer* specifies *where* a value is located.

A pointer denotes the *memory location* of a variable

Pointers

- In C, pointers are important for several reasons.
  - ***Pointers allow sharing of values stored in variables in a uniform way***
  - ***Pointers can refer to values that are allocated on demand (dynamic memory allocation)***
  - ***Pointers are necessary for implementing polymorphism, an important concept in object-oriented programming (later for C++)***

A Banking Problem

Consider a person.

(Harry)

Harry has more than one bank account.

Harry Needs a Banking Program

Harry wants a program for making bank deposits and withdrawals.

```
… balance += depositAmount …
… balance -= withdrawalAmount …
```

Harry Needs a Multi-Bank Banking Program

But not all deposits and withdrawals should be from the *same* bank.

```
… balance += depositAmount …
… balance -= withdrawalAmount …
```

Good Design

But withdrawing is withdrawing

– no matter which bank it is.

Same with depositing.

Same problem – same code, right?

Pointers to the Rescue

By using a *pointer*,

it is possible to *switch* to a different account

without modifying the code for

deposits and withdrawals.

Pointers to the Rescue

Harry starts with a variable for storing an account balance.
It should be initialized to 0 since there is no money yet.

```
double harrys_account = 0;
```

Pointers to the Rescue

If Harry anticipates that he may someday use other accounts, he can use a pointer to access any accounts.

So Harry also declares a pointer variable named **`account_pointer`** :

```
double* account_pointer
```

The type of this variable is "*pointer to double*".

Addresses and Pointers

A *pointer to double* type can hold the address of a **`double`**.

So what's an address?

Addresses and Pointers

Here’s a picture of RAM.

Every byte in RAM has an *address*.

(shown in groups of eight bytes)

an address

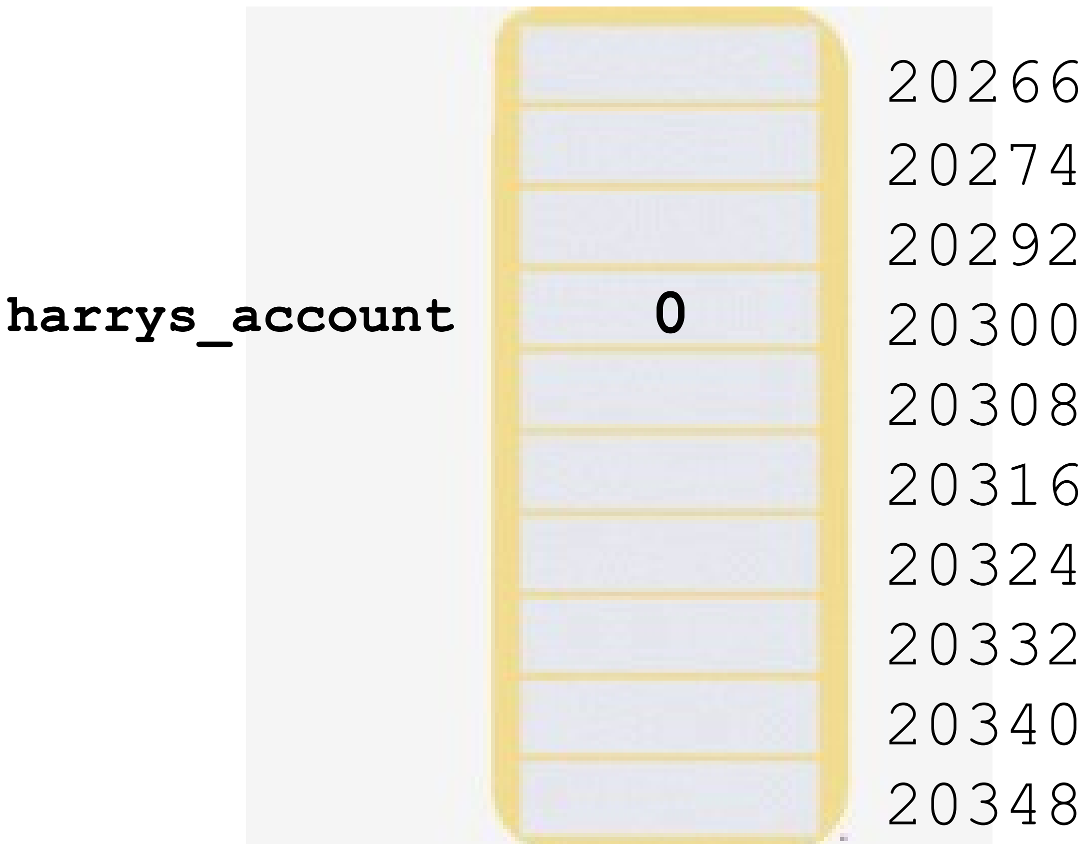
another address

Addresses and Pointers

Here's how we have pictured a variable in the past:

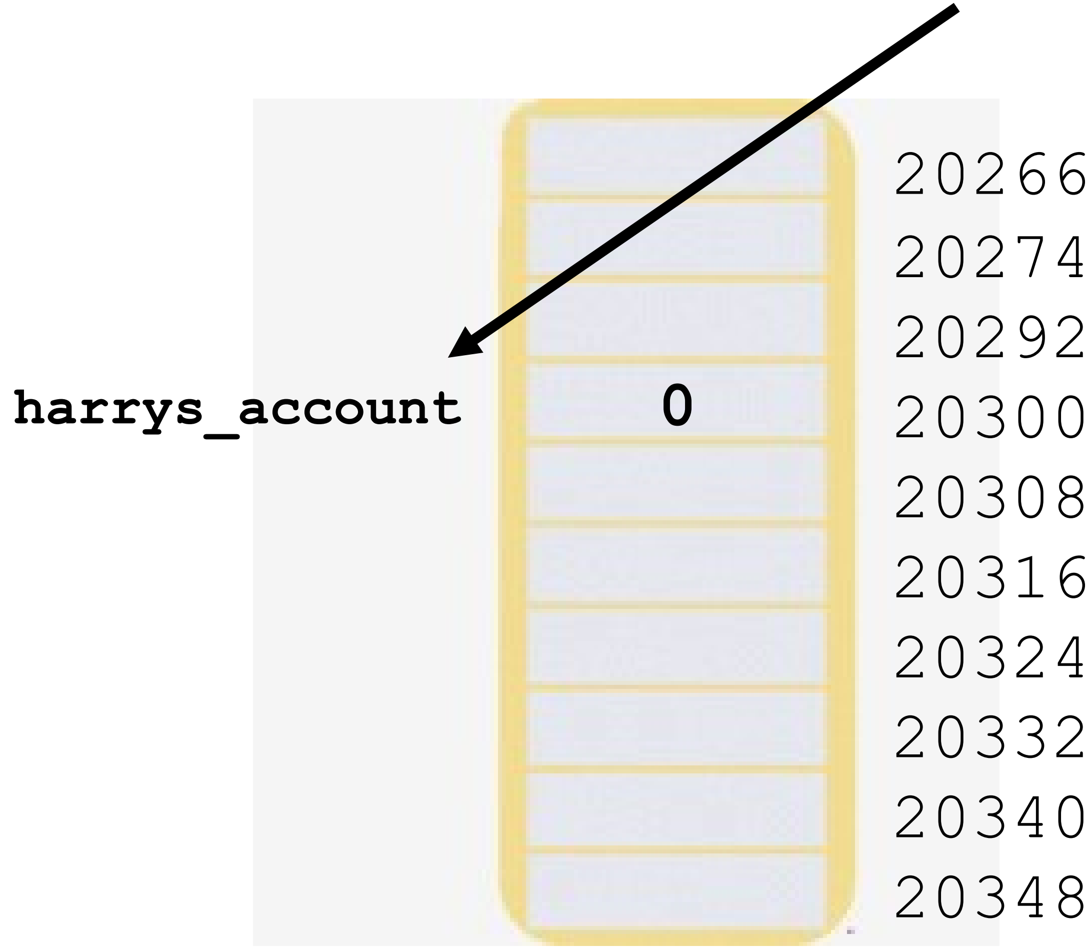
Addresses and Pointers

But really it's been like this all along:



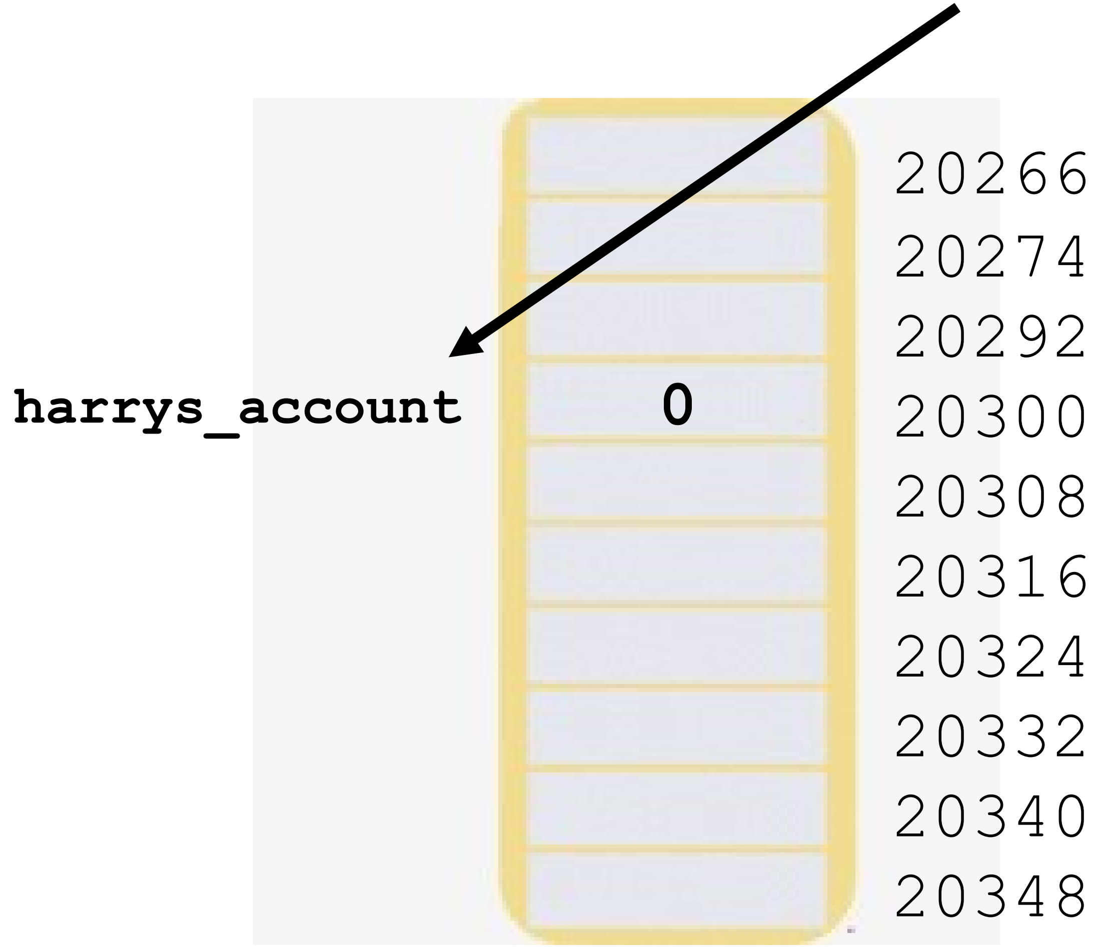
Addresses and Pointers

The address of the variable named **`harrys_account`**



Addresses and Pointers

The address of the variable named **`harrys_account`** is **`20300`**



Pointers to the Rescue

So when Harry declares a pointer variable,
he also initializes it to point to **`harrys_account`**:

```
double harrys_account = 0;
double* account_pointer = &harrys_account;
```

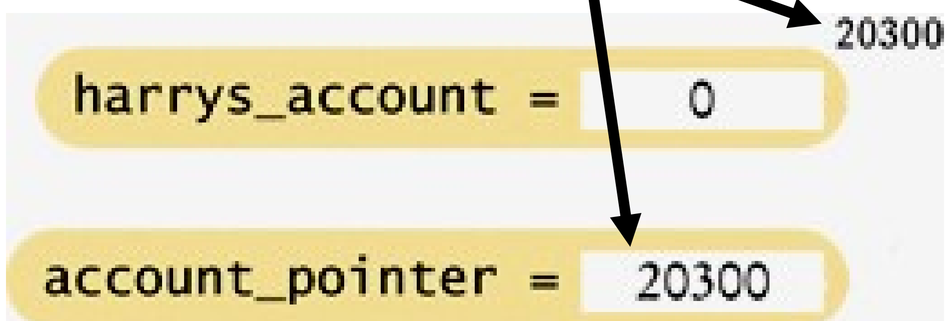
The **&** operator yields the location (or address) of a variable.

Taking the address of a **`double`** variable yields a value of type **`double*`** so everything fits together nicely.

Pointers to the Rescue

`account_pointer` now *contains* the *address* of **`harrys_account`**

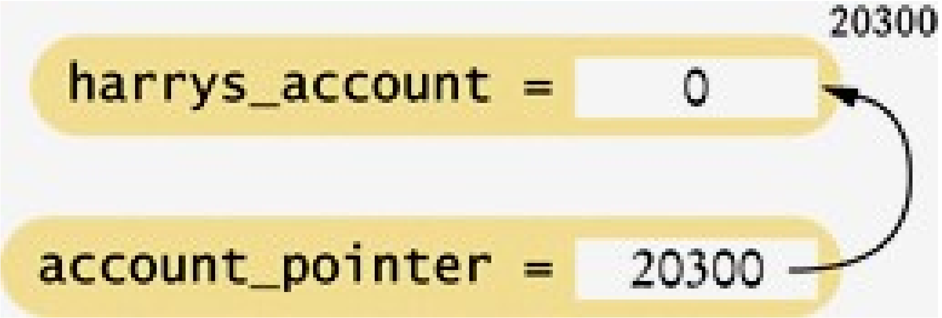
```
double harrys_account = 0;
double* account_pointer = &harrys_account;
```



Pointers to the Rescue

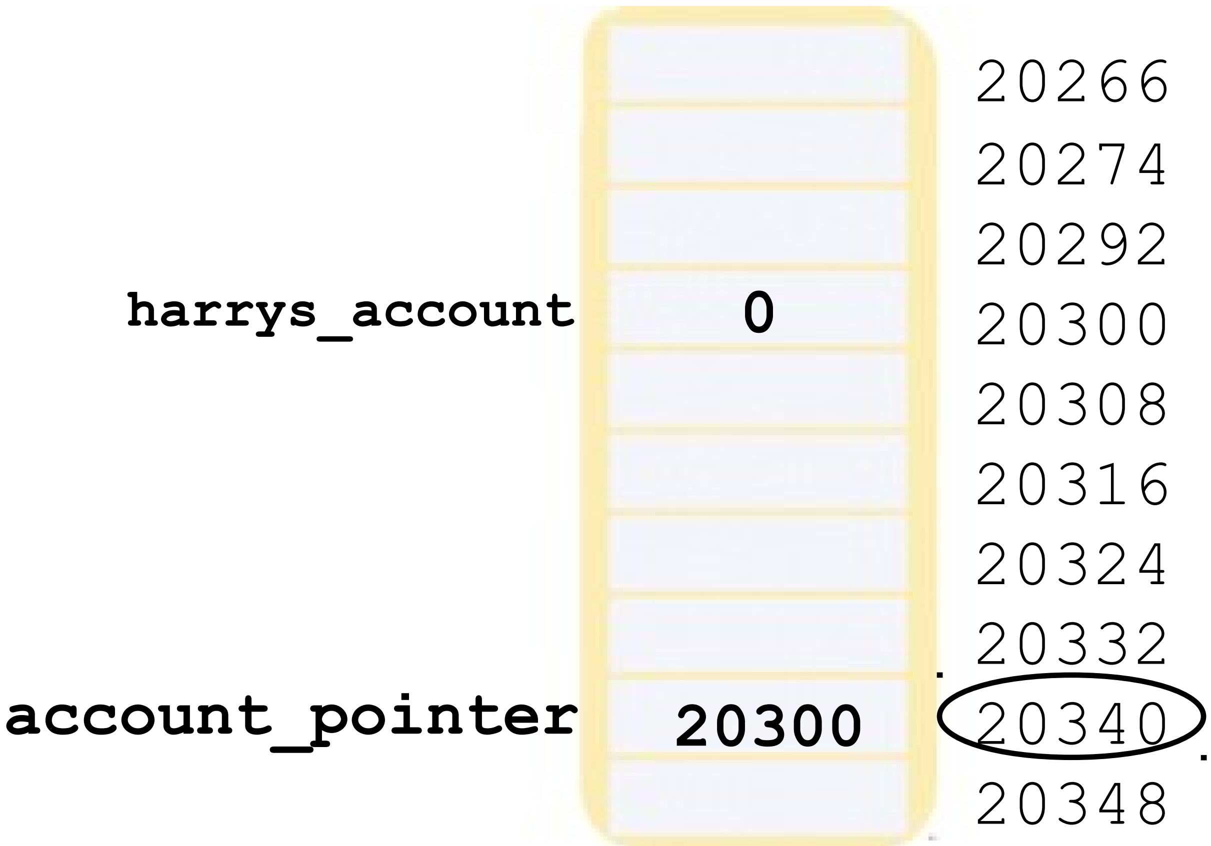
`account_pointer` now "points to" **`harrys_account`**

```
double harrys_account = 0;
double* account_pointer = &harrys_account;
```



Addresses and Pointers

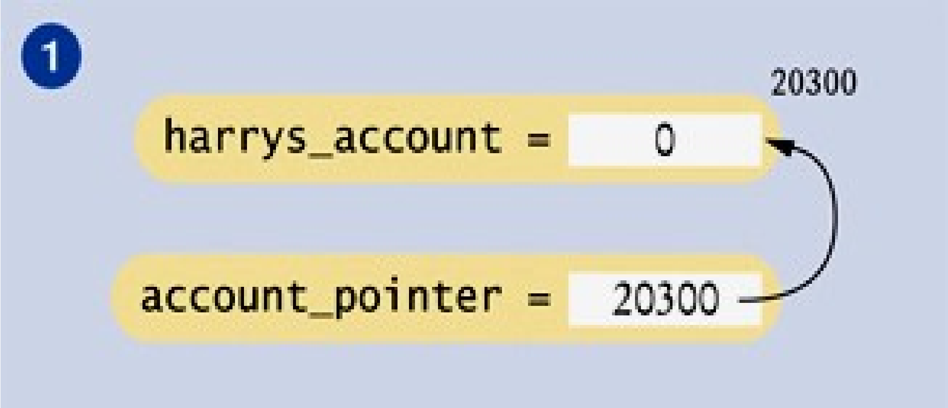
And, of course, **`account_pointer`** is *somewhere* in RAM:



Addresses and Pointers

To access a different account, you would change the pointer value stored in **`account_pointer`**:

```
double harrys_account = 0;
account_pointer = &harrys_account;
```

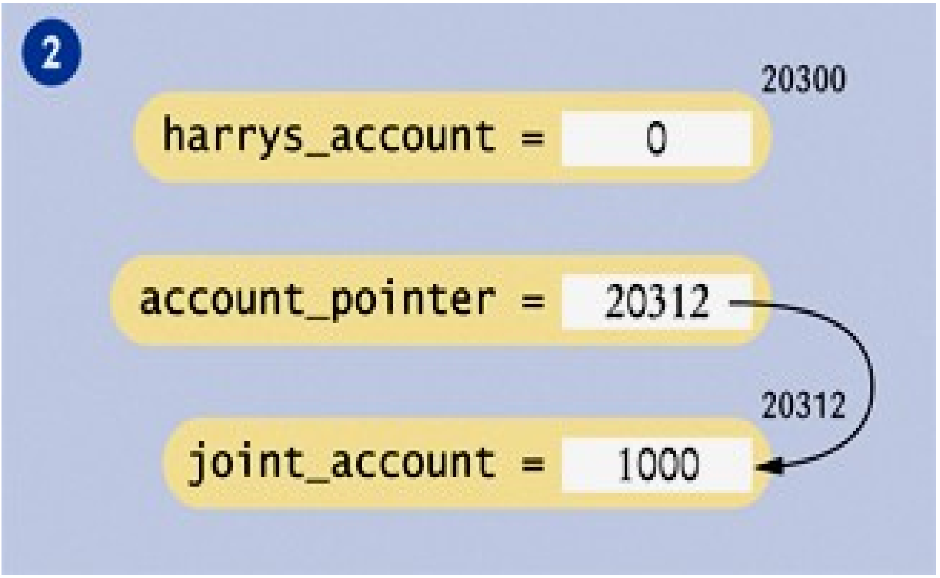


use **`account_pointer`** to access **`harrys_account`**

Addresses and Pointers

To access a different account, like **joint_account**, change the pointer value stored in **account_pointer** and similarly use **account_pointer**.

```
double harrys_account = 0;
account_pointer = &harrys_account;
double joint_account = 1000;
account_pointer = &joint_account;
```



Addresses and Pointers – and ARROWS

Do note that the computer stores numbers,

not arrows.

Accessing the Memory Pointed to by A Pointer Variable

When you have a pointer to a variable,
you will want to access the value to which it points.

```
... *account_pointer ...
```

In C, the * operator is used to indicate

the memory location associated with a pointer.

Accessing the Memory Pointed to by A Pointer Variable

An expression such as `*account_pointer` can be used wherever a variable name of the same type can be used:

```
// display the current balance
printf("%f\n", *account_pointer);
```

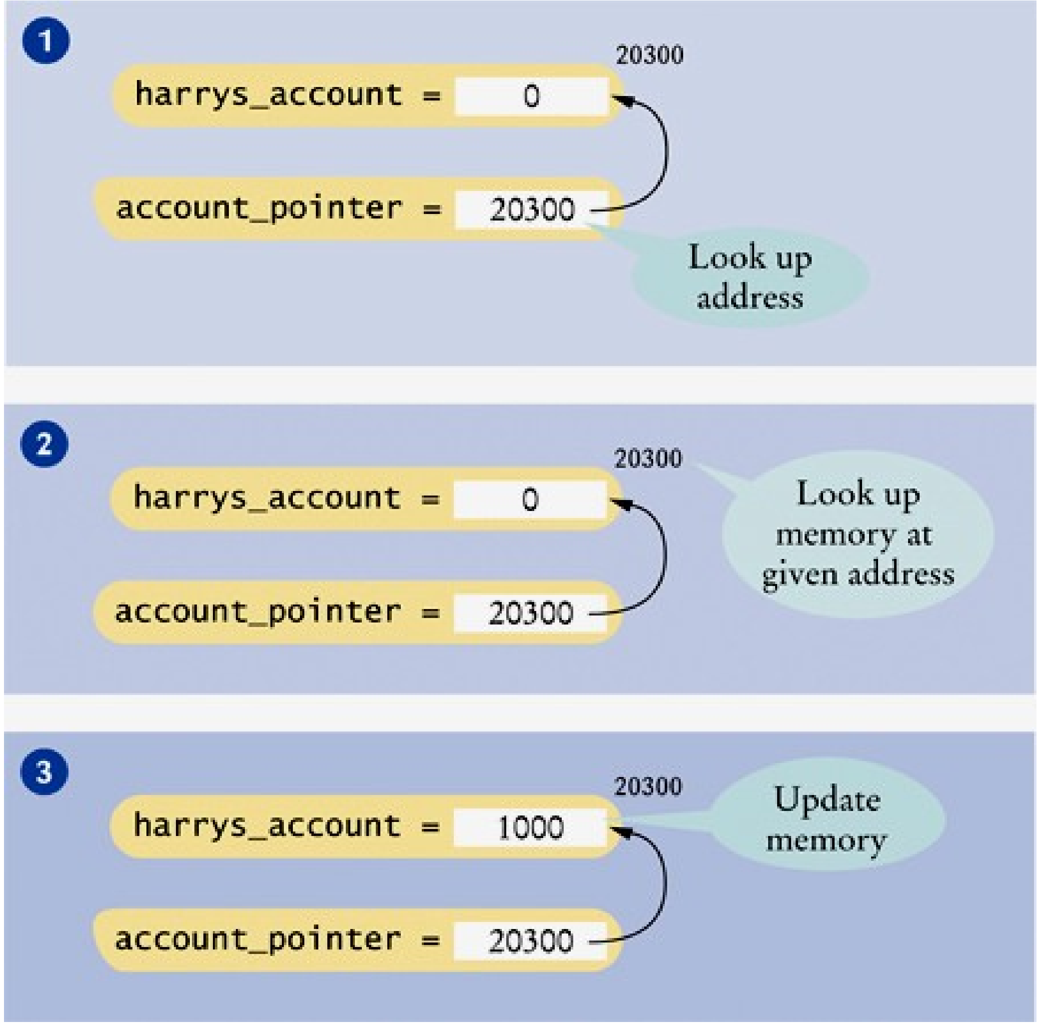
It can be used on the left or the right of an assignment:

```
// withdraw $100
*account_pointer = *account_pointer - 100;
```

(or both)

Harry Makes the Deposit

```
// deposit $1000
  *account_pointer = *account_pointer + 1000;
```



Accessing the Memory Pointed to by A Pointer Variable

Of course, this only works
if **`account_pointer`** is pointing
to **`harrys_account`**!

Errors Using Pointers – Uninitialized Pointer Variables

When a pointer variable is first defined,
it contains a random address.

Using that random address is an error.

Errors Using Pointers – Uninitialized Pointer Variables

In practice, your program will likely crash or mysteriously misbehave if you use an uninitialized pointer:

```
double* account_pointer; // No initialization

*account_pointer = 1000;
```

NO!

`account_pointer` contains an *unpredictable* value!

`Where is the 1000 going?`

NULL

There is a special value
that you can use
to indicate a pointer
that doesn't point anywhere:

`NULL`

NULL

If you define a pointer variable
and are not ready to initialize it quite yet,
it is a good idea to set it to **NULL**.

You can later test whether the pointer is **NULL**.
If it is, don't use it:

```
double* account_pointer = NULL; // Will set later
if (account_pointer != NULL) {  // OK to use
   printf("%f\n", *account_pointer);
}
```

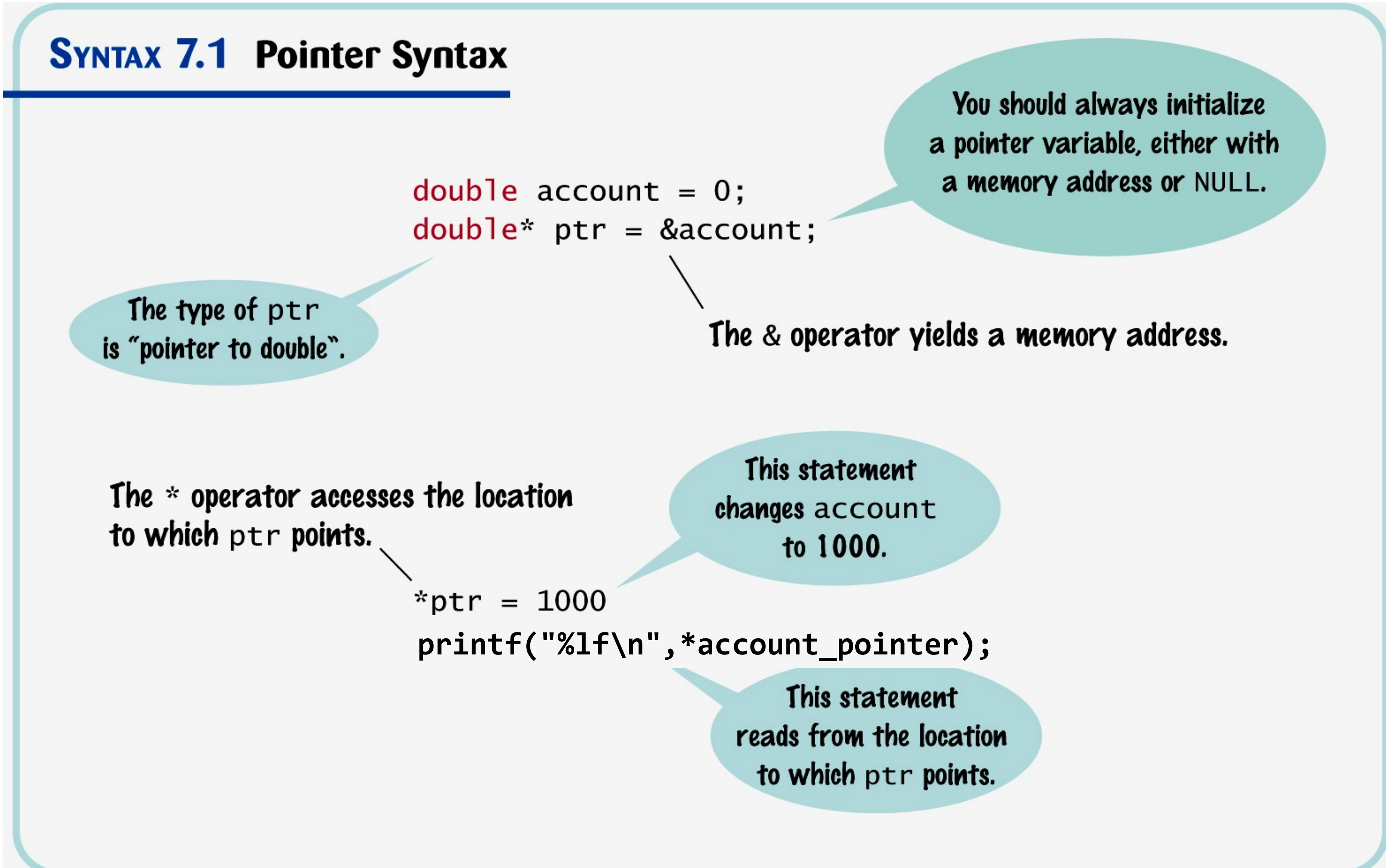
NULL

Trying to access data through a NULL pointer is still illegal, and
it will cause your program to crash.

```
double* account_pointer = NULL;

printf("%f\n", *account_pointer);
```

CRASH!!!

Syntax of Pointers



Pointer Syntax Examples

Table 1 Pointer Syntax Examples

Assume the following declarations:

```
int m = 10; // Assumed to be at address 20300
int n = 20; // Assumed to be at address 20304
int* p = &m;
```

Expression	Value	Comment
`p`	20300	The address of `m`.
`*p`	10	The value stored at that address.
`&n`	20304	The address of `n`.
`p = &n;`		Set `p` to the address of `n`.
`*p`	20	The value stored at the changed address.
`m = *p;`		Stores 20 into `m`.
`m = p;`	**Error**	`m` is an `int` value; `p` is an `int*` pointer. The types are not compatible.
`&10`	**Error**	You can only take the address of a variable.
`&p`	The address of `p`, perhaps 20308	This is the location of a pointer variable, not the location of an integer.
`double x = 0;` `p = &x;`	**Error**	`p` has type `int*`, `&x` has type `double*`. These types are incompatible.

Harry’s Banking Program

Here is the complete banking program.

It demonstrates the use of a pointer variable to allow *uniform access* to variables.

```
#include <stdio.h>

int main()
{
   double harrys_account = 0;
   double joint_account = 2000;
   double* account_pointer = &harrys_account;
   *account_pointer = 1000; // Initial deposit
```

Harry's Banking Program

ch07/accounts.cpp

```
    // Withdraw $100
    *account_pointer = *account_pointer - 100;

    // Print balance
    printf("Balance: %lf\n", *account_pointer);


    // Change the pointer value so that the same
    // statements now affect a different account
    account_pointer = &joint_account;

    // Withdraw $100
    *account_pointer = *account_pointer - 100;

    // Print balance
    printf("Balance: %lf\n", *account_pointer);

    return 0;
}
```

Common Error: Confusing Data And Pointers

A pointer is a memory address

– a number that tells where a value is located in memory.

It is a common error to confuse the pointer with the variable to which it points.

Common Error: Where's the *?

```
double* account_pointer = &joint_account;
account_pointer = 1000;
```

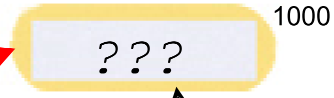
The assignment statement does *not* set the joint account balance to 1000.

It sets the pointer variable, **account_pointer**, to point to memory address 1000.

ERROR

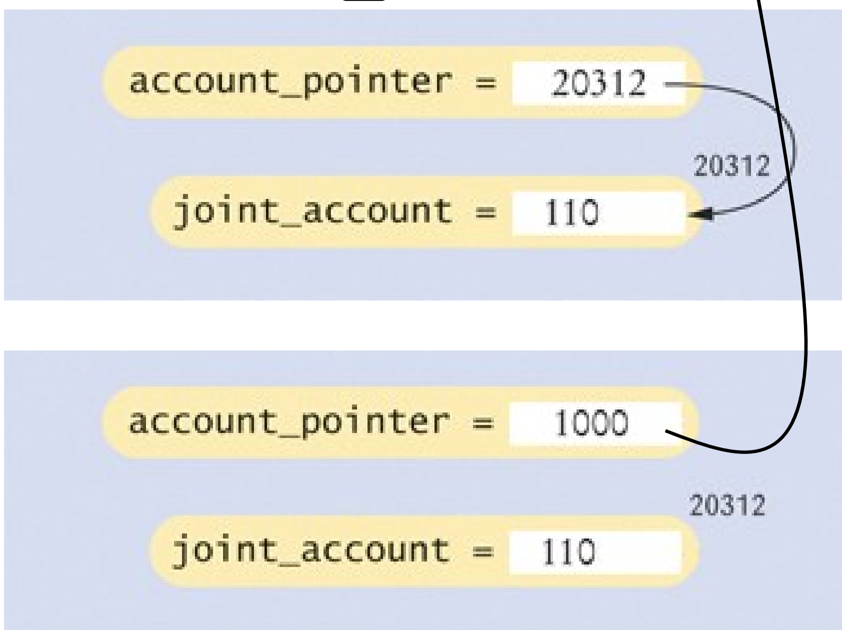
Common Error: Where's the *?

`joint_account` is almost certainly *not* located at address 1000!




```
double* account_pointer = &joint_account;
```

```
account_pointer = 1000;
```



Common Error: Where's the *?

Most compilers will report an error for this kind of error.

Confusing Definitions

It is legal in C to define multiple variables together, like this:

```
int i = 0, j = 1;
```

This style is confusing when used with pointers:

```
double* p, q;
```

The `*` associates only with the first variable.
That is, `p` is a `double*` pointer, and `q` is a `double` value.

To avoid any confusion, it is best to define each pointer variable separately:

```
double* p;
double* q;
```

Pointers and References

Changing value of parameter:

```
void withdraw(double* balance, double amount)
{
    if (*balance >= amount)
    {
        *balance = *balance - amount;
    }
}
```

but the call will have to be:

```
withdraw(&harrys_checking, 1000);
```

Arrays and Pointers

In C, there is a deep relationship between pointers and arrays.

This relationship explains a number of special properties and limitations of arrays.

Arrays and Pointers

Pointers are particularly useful for understanding the peculiarities of arrays.

The *name* of the array denotes a pointer to the starting element.

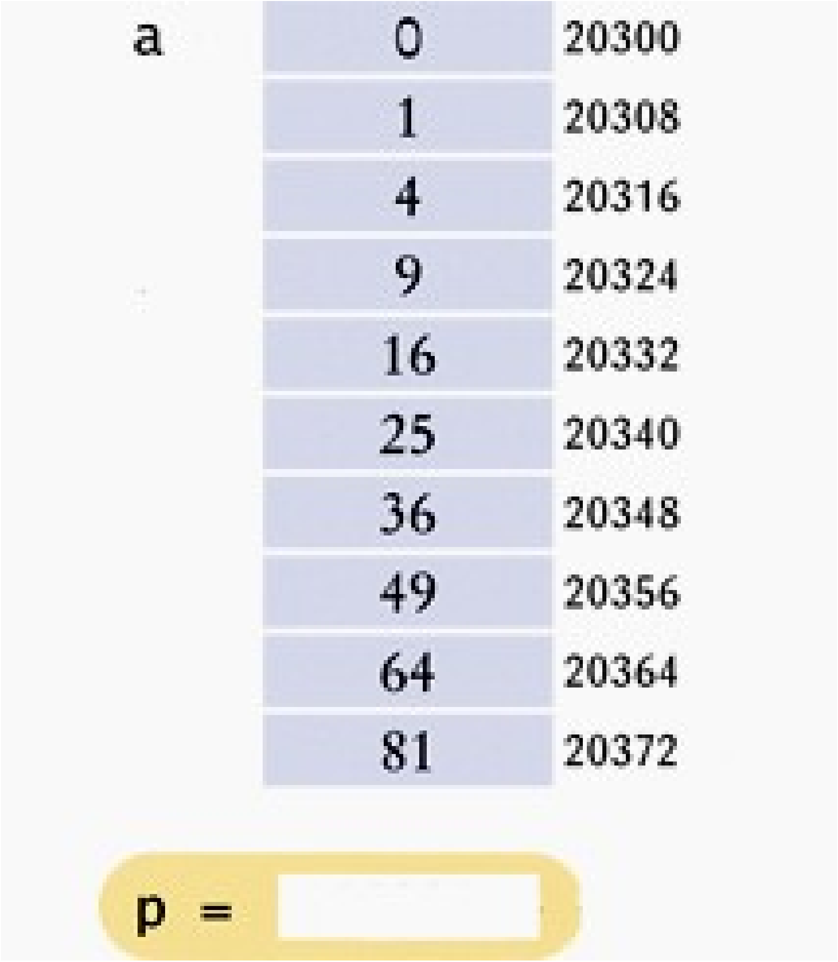
Arrays and Pointers

Consider this declaration:

```
int a[10];
```

(Assume we have filled it as shown.)

You can capture the pointer to the first element in the array in a variable:



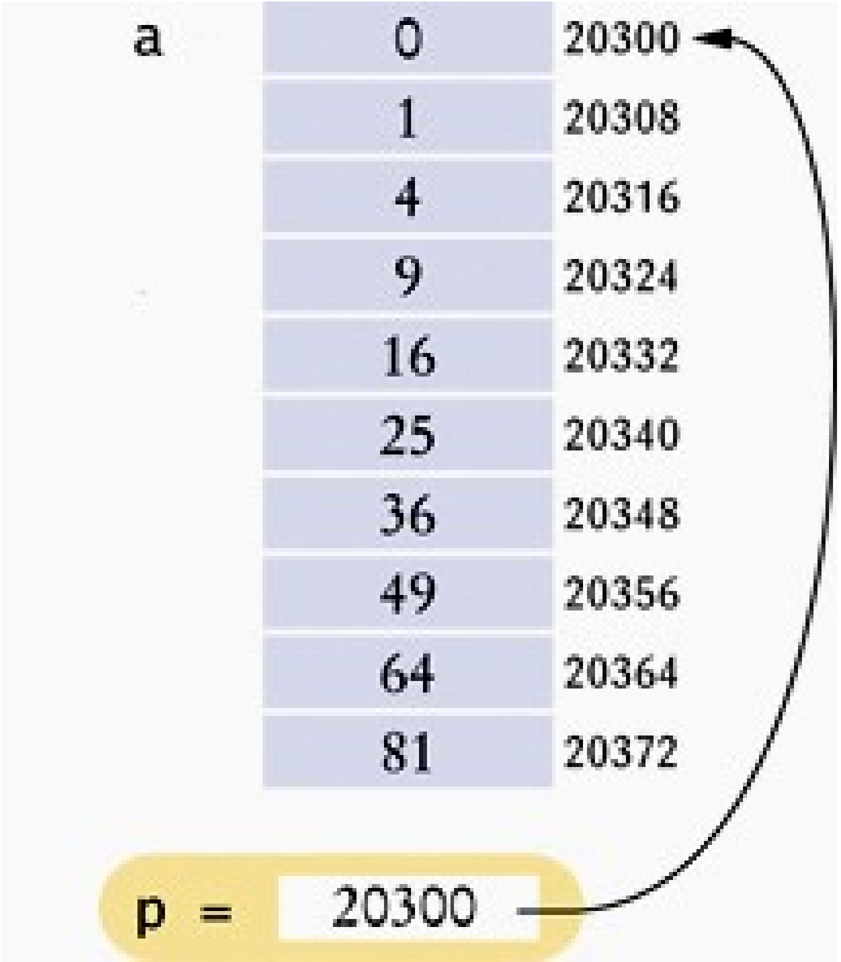
Arrays and Pointers

Consider this declaration:

```
int a[10];
```

(Assume we have
filled it as shown.)

You can capture the
pointer to the first
element in the array
in a variable:




```
int* p = a; // Now p points to a[0]
```

Arrays and Pointers – Same Use

You can use the array name **`a`** as you would a pointer:

These output statements are equivalent:

```
printf("%d", *a);
printf("%d", a[0]);
```

Pointer Arithmetic

Pointer arithmetic allows you to add an integer to an array name.

```
int* p = a;
```

`p + 3` is a pointer to the array element with index `3`

The expression: `*(p + 3)`

.

The Array/Pointer Duality Law

The *array/pointer duality law* states:

`a[n]` is identical to **`*(a + n)`**,

where **`a`** is a pointer into an array
and **`n`** is an integer offset.

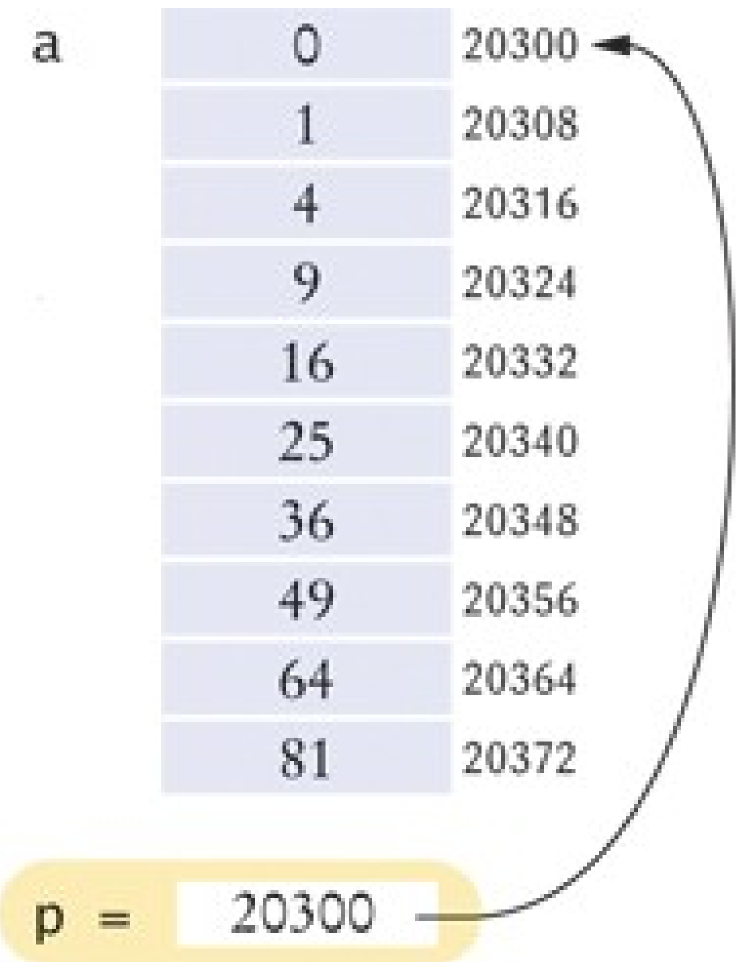
The Array/Pointer Duality Law

This law explains why all C arrays start with an index of zero.

The pointer **`a`** (or **`a + 0`**) points to the starting element of the array.

That element must therefore be **`a[0]`**.

You are adding 0 to the start of the array, thus *correctly going nowhere!*



The Array/Pointer Duality Law

Now it should be clear why array parameters are different from other parameter types.

(if not, we'll show you)

The Array/Pointer Duality Law

Consider this function that computes the sum of all values in an array:

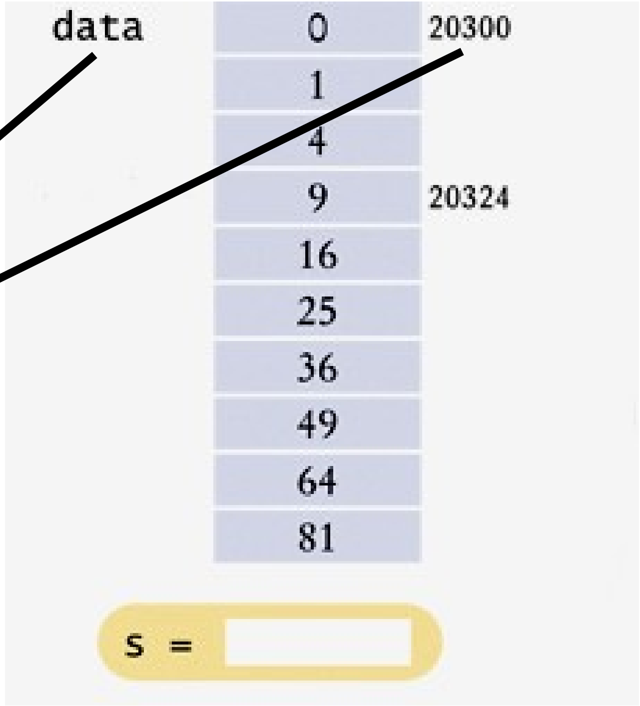
Look at this

```
double sum(double a[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++) {
      total = total + a[i];
   }
   return total;
}
```

The Array/Pointer Duality Law

Here is a call to the function.

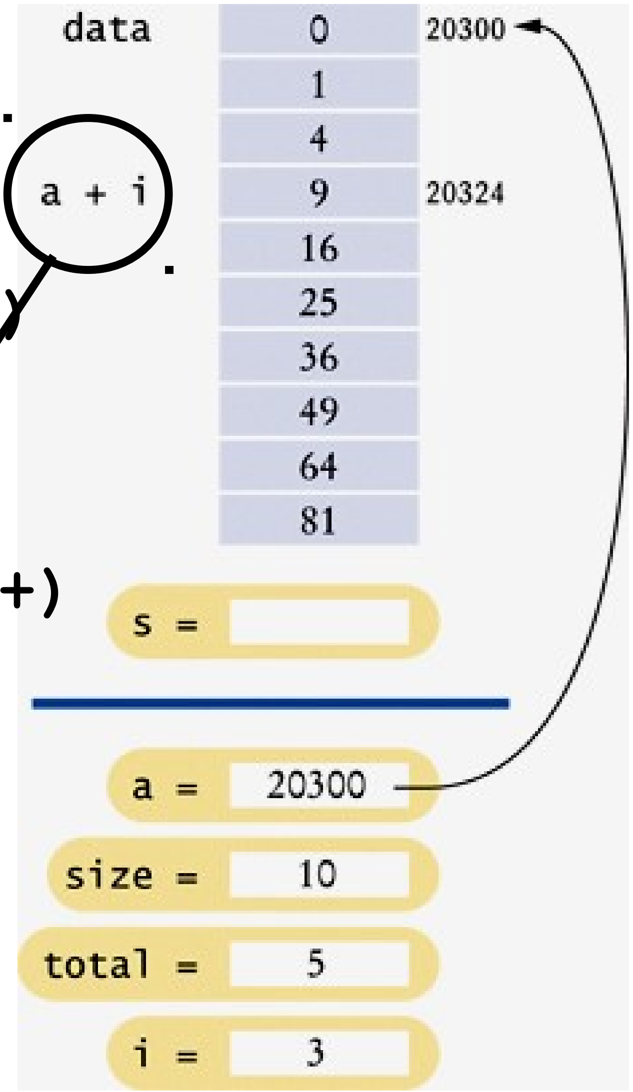
```
double data[10];
... // Initialize data

double s = sum(data, 10);
```



The Array/Pointer Duality Law

After the loop has run
to the point when `i` is 3:

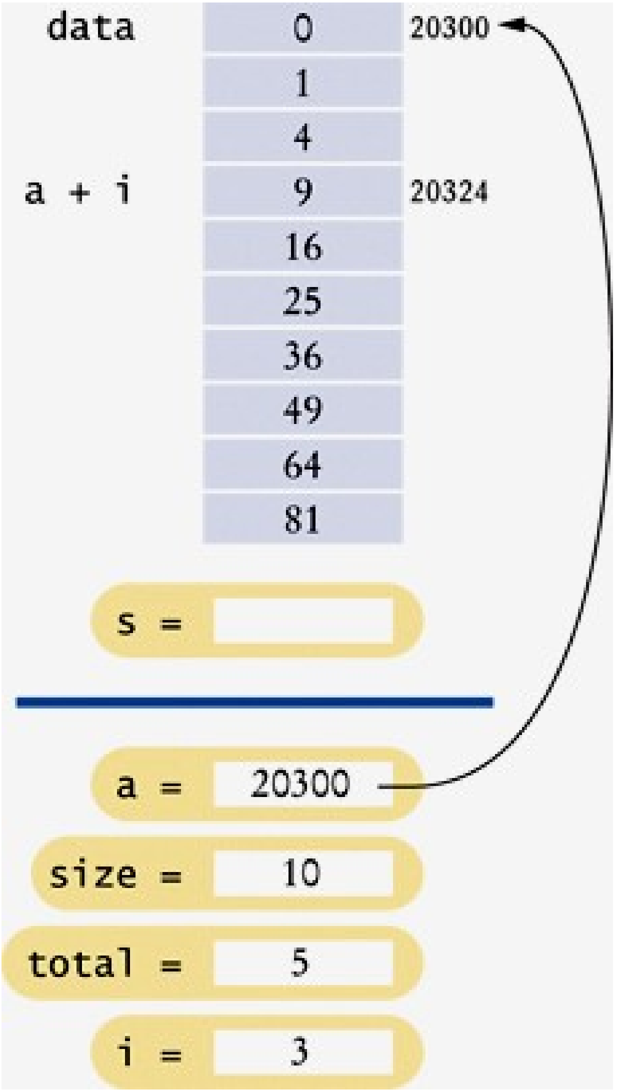
```
double sum(double a[], int size)
{
    double total = 0;
    for (int i = 0; i < size; i++)
    {
        total = total + a[i];
    }
    return total;
}
```



The Array/Pointer Duality Law

The C compiler considers
`a` to be a pointer, not an array.

The expression **`a[i]`**
is *syntactic sugar*
for **`*(a + i)`**.



Syntactic Sugar

Computer scientists use the term

"*syntactic sugar*"

to describe a notation that is easy to read for humans and that masks a complex implementation detail.

Syntactic Sugar

That masked complex implementation detail:

```
double sum(double* a, int size)
```

is how we *should* define the first parameter

but

```
double sum(double a[], int size)
```

looks a lot more like we are passing an array.

Arrays and Pointers

Table 2 Arrays and Pointers

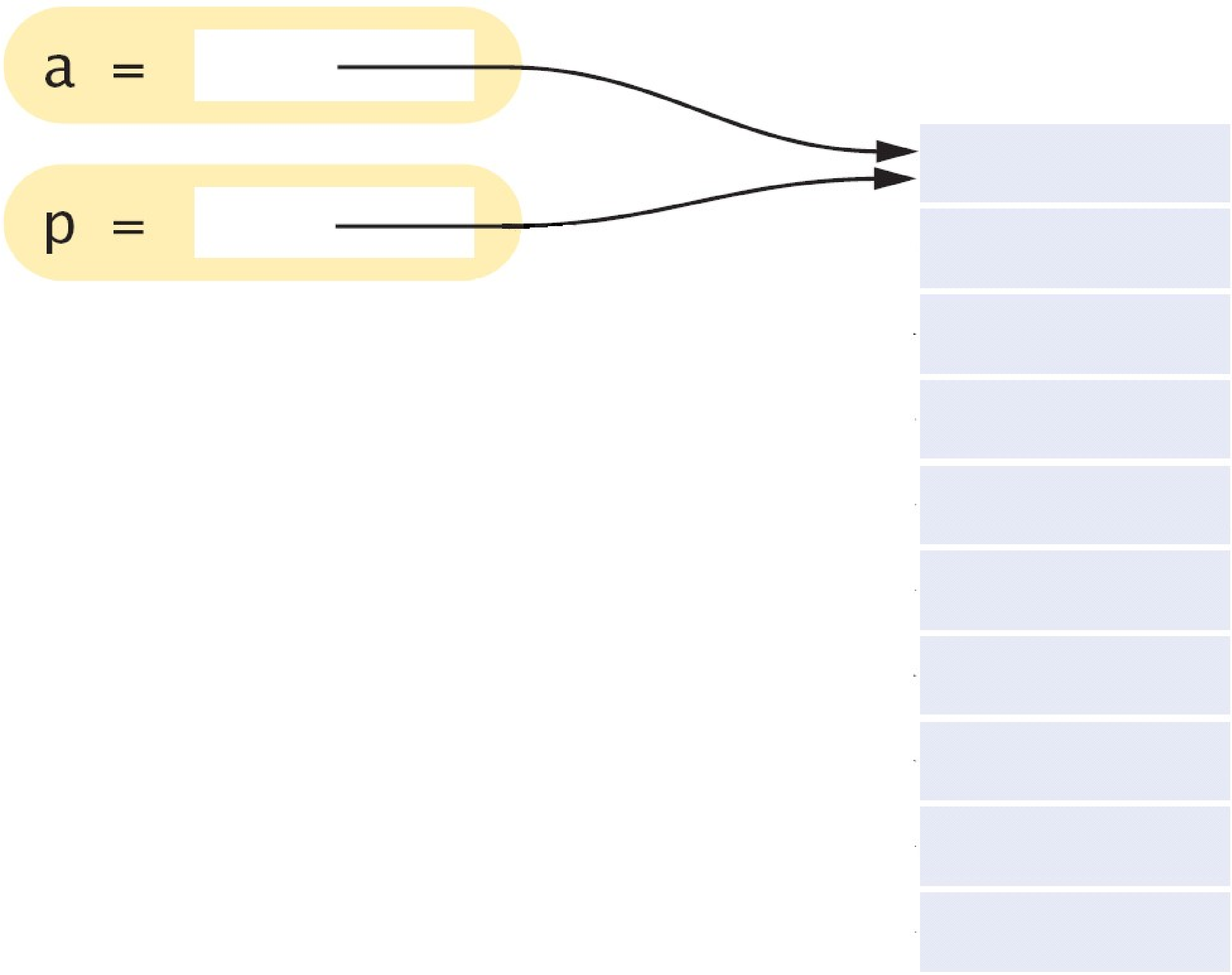
Expression	Value	Comment
`a`	20300	The starting address of the array, here assumed to be 20300.
`*a`	0	The value stored at that address. (The array contains values 0, 1, 4, 9, ...)
`a + 1`	20308	The address of the next `double` value in the array. A `double` occupies 8 bytes.
`a + 3`	20324	The address of the element with index 3, obtained by skipping past 3 × 8 bytes.
`*(a + 3)`	9	The value stored at address 20324.
`a[3]`	9	The same as `*(a + 3)` by array/pointer duality.
`*a + 3`	3	The sum of `*a` and 3. Since there are no parentheses, the `*` refers only to `a`.
`&a[3]`	20324	The address of the element with index 3, the same as `a + 3`.

Using a Pointer to Step Through an Array

Watch variable p as this code is executed.

```
double sum(double* a, int size)
{
    double total = 0;
    double* p = a;
    // p starts at the beginning of the array
    for (int i = 0; i < size; i++) {
        total = total + *p;
        // Add the value to which p points
        p++;
        // Advance p to the next array element
    }
    return total;
}
```

Using a Pointer to Step Through an Array

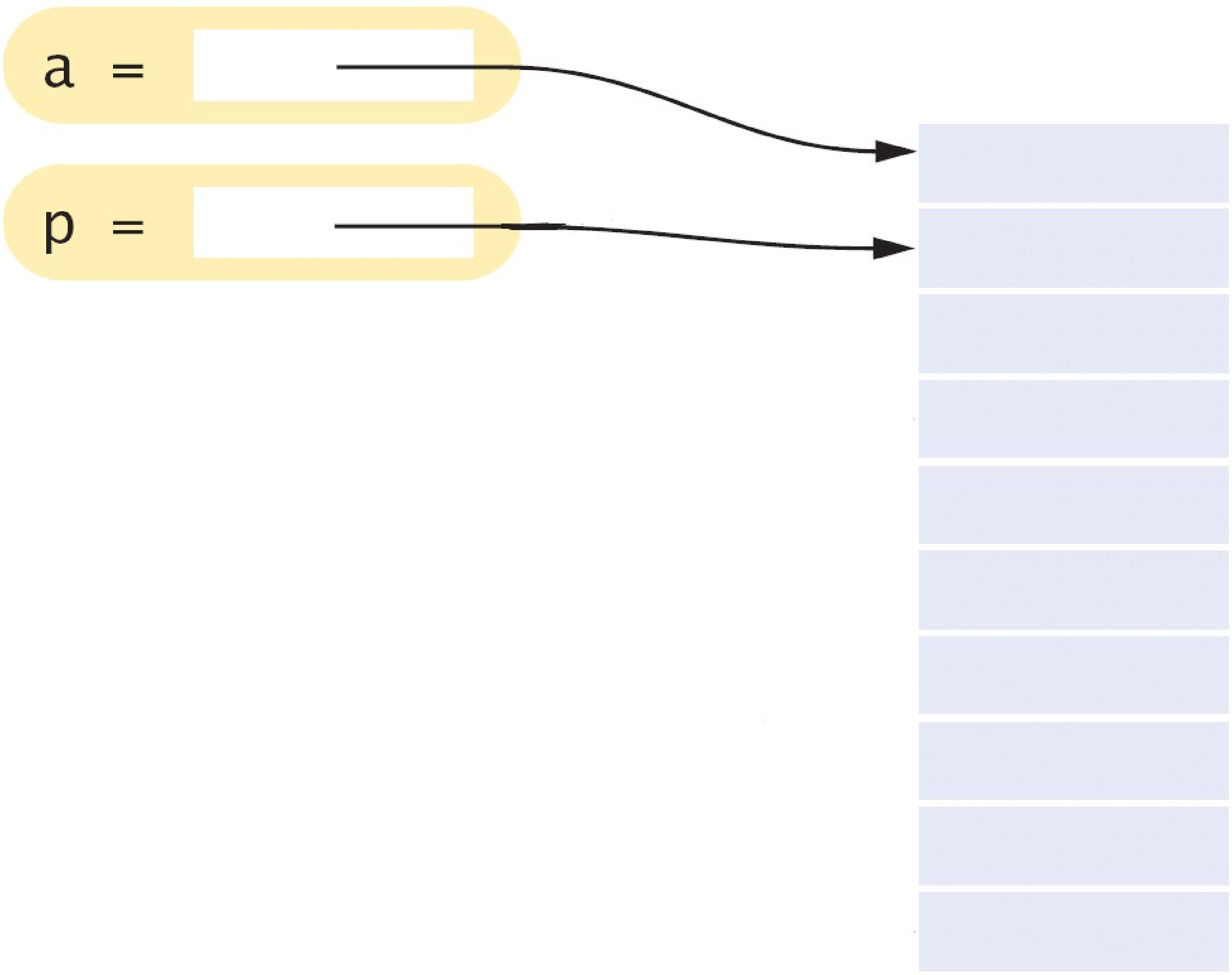


Using a Pointer to Step Through an Array

Watch variable p as this code is executed.

```
double sum(double* a, int size)
{
   double total = 0;
   double* p = a;
   // p starts at the beginning of the array
   for (int i = 0; i < size; i++)
   {
      total = total + *p;
      // Add the value to which p points
      p++;
      // Advance p to the next array element
   }
   return total;
}
```

Using a Pointer to Step Through an Array

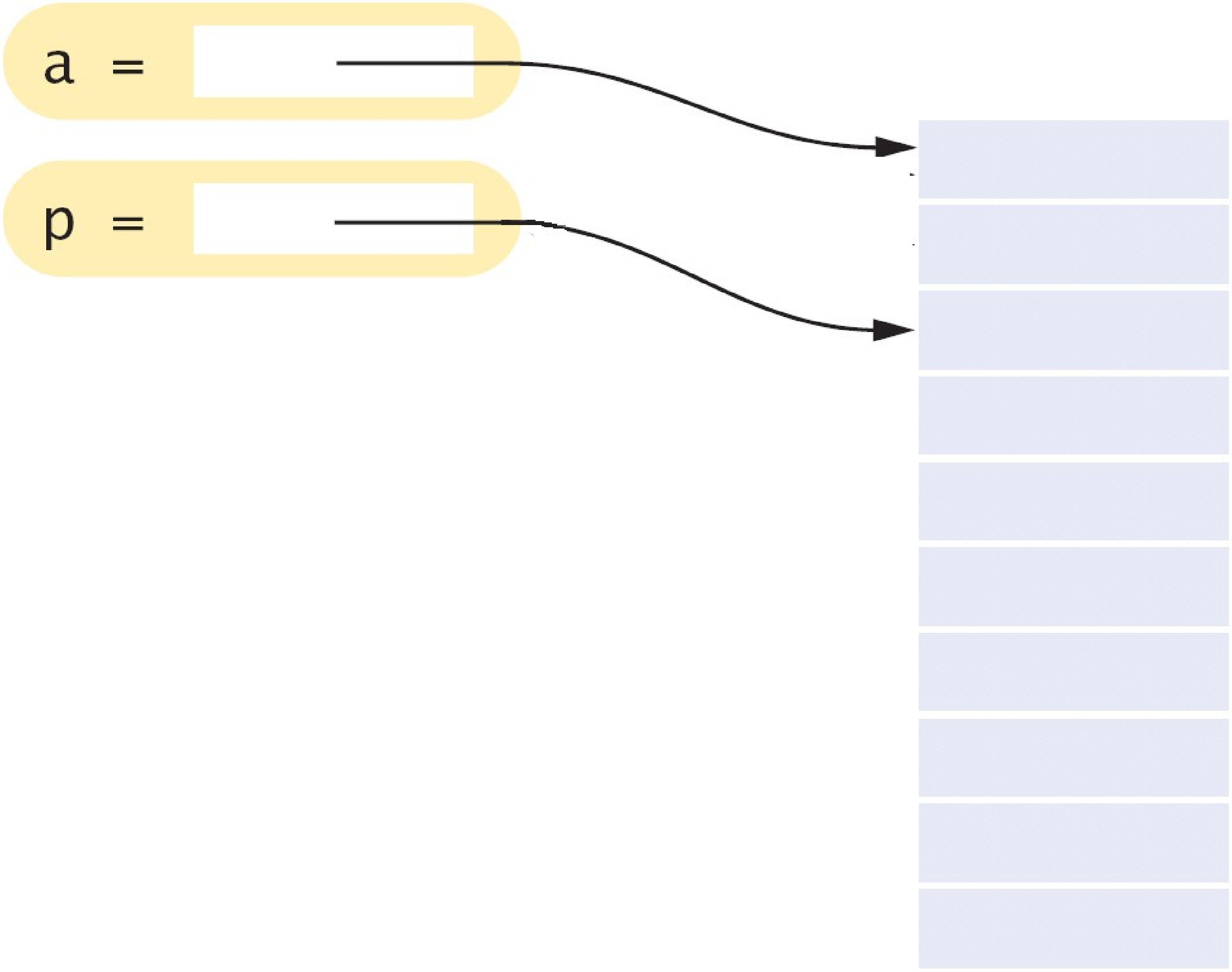


Using a Pointer to Step Through an Array

Watch variable p as this code is executed.

```
double sum(double* a, int size)
{
    double total = 0;
    double* p = a;
    // p starts at the beginning of the array
    for (int i = 0; i < size; i++)
    {
        total = total + *p;
        // Add the value to which p points
        p++;
        // Advance p to the next array element
    }
    return total;
}
```

Using a Pointer to Step Through an Array

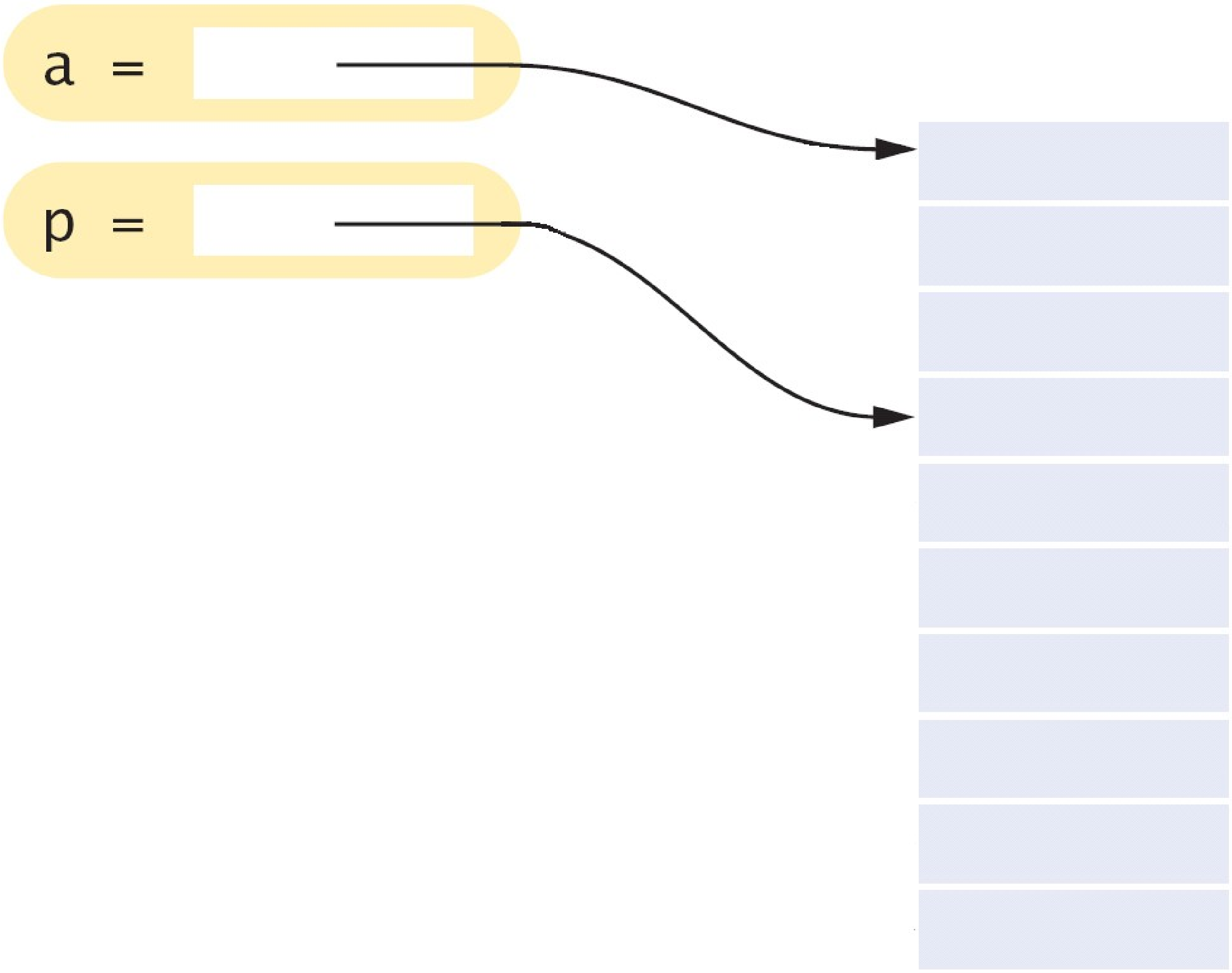


Using a Pointer to Step Through an Array

Add, then move **p** to the next position by incrementing.

```
double sum(double* a, int size)
{
    double total = 0;
    double* p = a;
    // p starts at the beginning of the array
    for (int i = 0; i < size; i++)
    {
       total = total + *p;
       // Add the value to which p points
       p++;
       // Advance p to the next array element
    }
    return total;
}
```

Using a Pointer to Step Through an Array

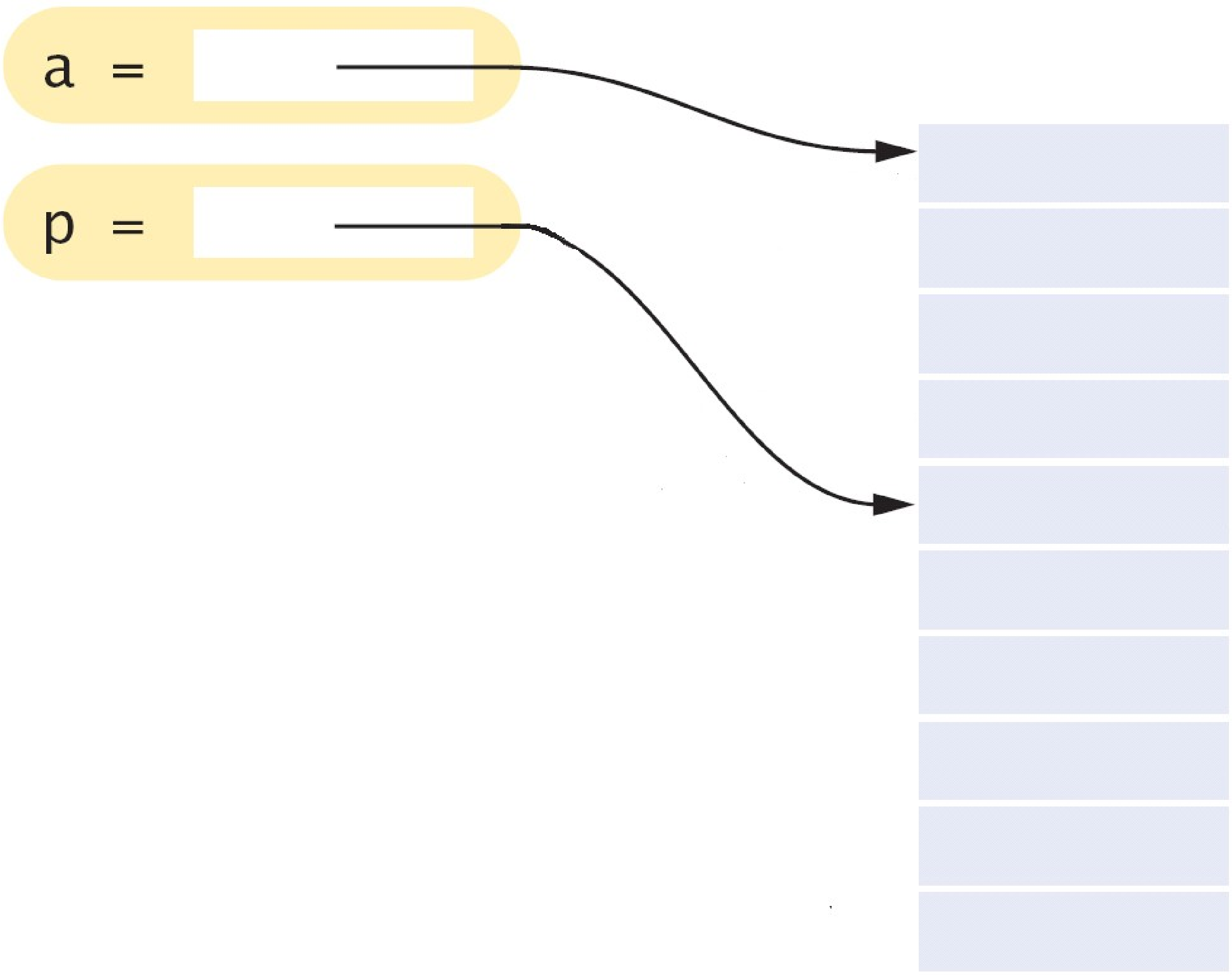


Using a Pointer to Step Through an Array

Add, then again move **p** to the next position by incrementing.

```
double sum(double* a, int size)
{
   double total = 0;
   double* p = a;
   // p starts at the beginning of the array
   for (int i = 0; i < size; i++)
   {
      total = total + *p;
      // Add the value to which p points
      p++;
      // Advance p to the next array element
   }
   return total;
}
```

Using a Pointer to Step Through an Array

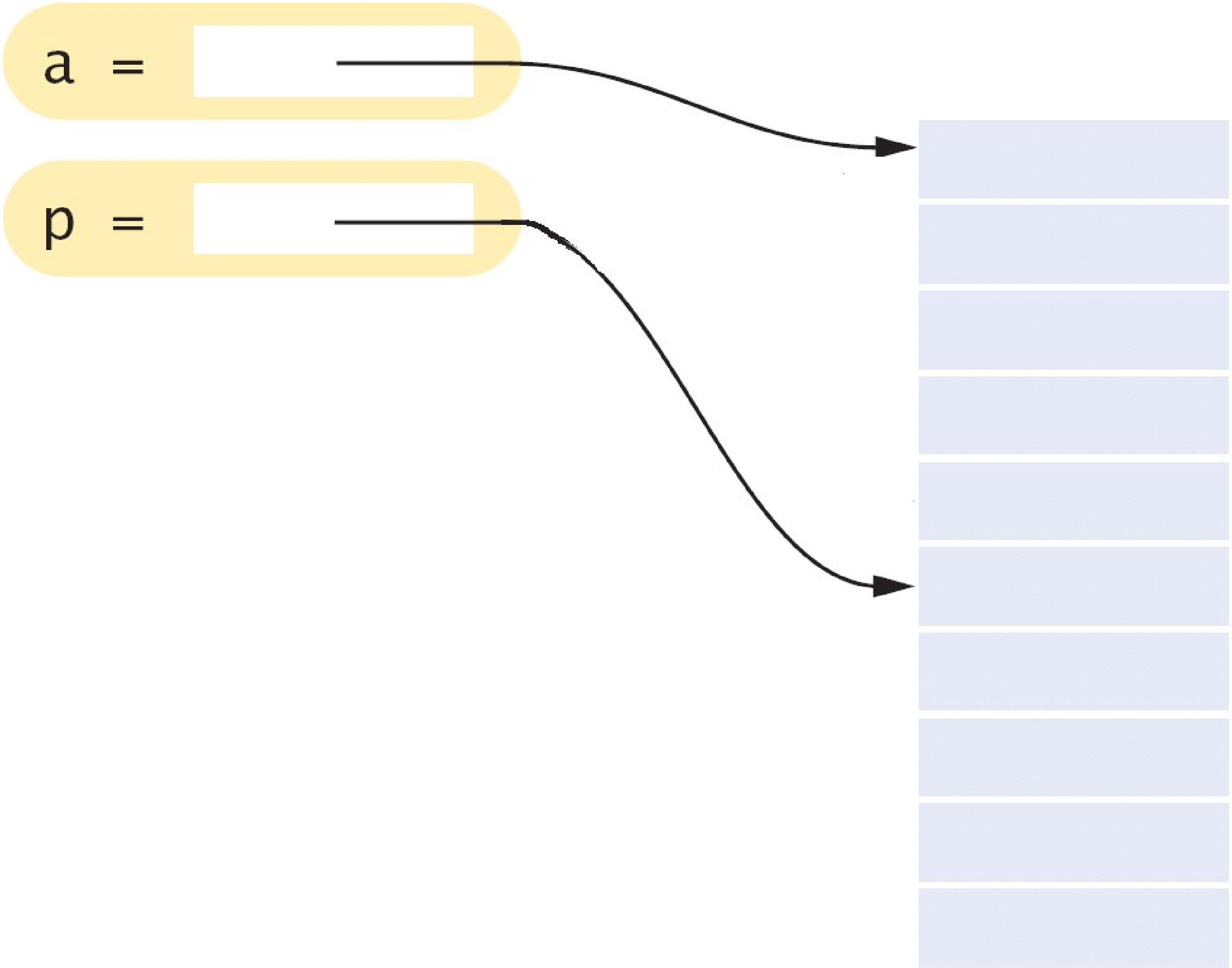


Using a Pointer to Step Through an Array

Add, then move **p**.

```
double sum(double* a, int size)
{
   double total = 0;
   double* p = a;
   // p starts at the beginning of the array
   for (int i = 0; i < size; i++)
   {
      total = total + *p;
      // Add the value to which p points
      p++;
      // Advance p to the next array element
   }
   return total;
}
```

Using a Pointer to Step Through an Array

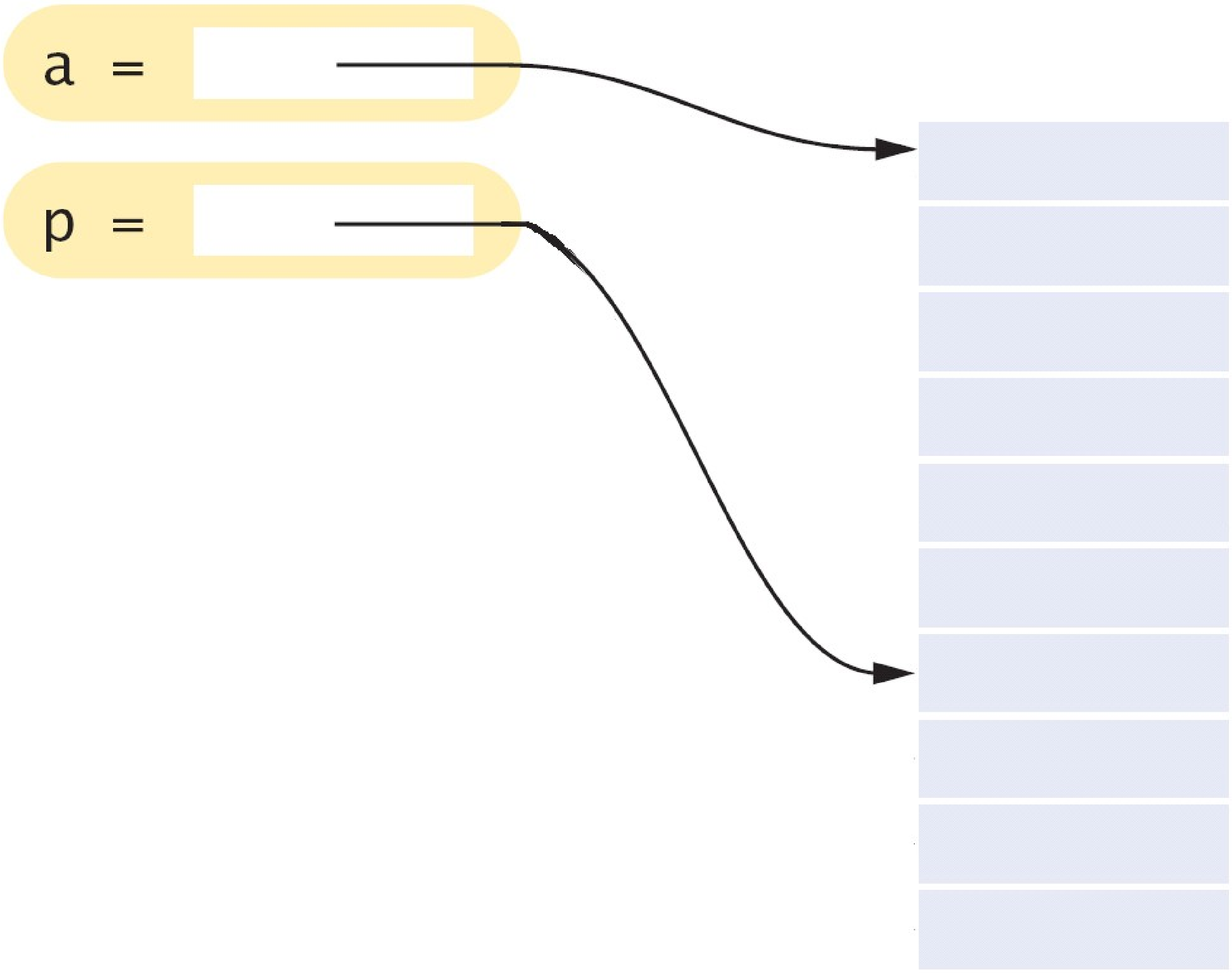


Using a Pointer to Step Through an Array

Again...

```
double sum(double* a, int size)
{
   double total = 0;
   double* p = a;
   // p starts at the beginning of the array
   for (int i = 0; i < size; i++)
   {
      total = total + *p;
      // Add the value to which p points
      p++;
      // Advance p to the next array element
   }
   return total;
}
```

Using a Pointer to Step Through an Array

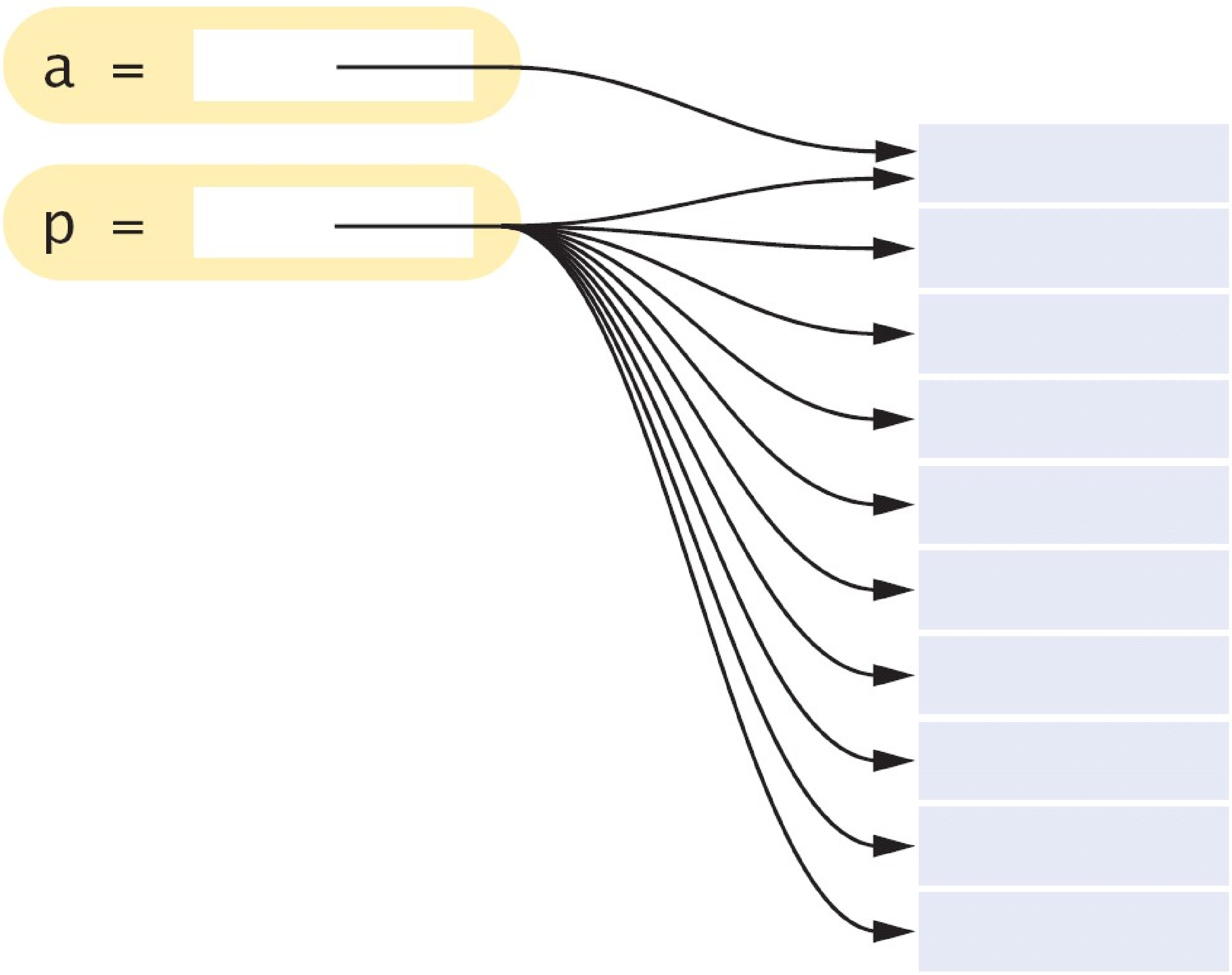


Using a Pointer to Step Through an Array

And so on until every single position in the array has been added.

```
double sum(double* a, int size)
{
   double total = 0;
   double* p = a;
   // p starts at the beginning of the array
   for (int i = 0; i < size; i++)
   {
      total = total + *p;
      // Add the value to which p points
      p++;
      // Advance p to the next array element
   }
   return total;
}
```

Using a Pointer to Step Through an Array



Using a Pointer to Step Through an Array

It is a tiny bit more efficient to use and increment a pointer than to access an array element.

Program Clearly, Not Cleverly

Some programmers take great pride
in minimizing the number of instructions,
even if the resulting code is hard to understand.

```
while (size-- > 0) // Loop size times
{
   total = total + *p;
   p++;
}
```

could be written as:

```
total = total + *p++;
```

Ah, so much better?

Program Clearly, Not Cleverly

```
while (size > 0)
{
    total = total + *p;
    p++;
    size--;
}
```

could be written as:

```
while (size-- > 0)
    total = total + *p++;
```

Ah, so much better?

Program Clearly, Not Cleverly

Please do not use this programming style.

Your job as a programmer is not to dazzle other programmers
with your cleverness,
but to write code that is easy
to understand and maintain.

Common Error: Returning a Pointer to a Local Variable

What would it mean to
"return an array"
?

Common Error: Returning a Pointer to a Local Variable

Consider this function that tries to return
a pointer to an array containing two elements,
the first and last values of an array:

```
double* firstlast(double a[], int size)
{
   double result[2];
   result[0] = a[0];
   result[1] = a[size - 1];
   return result;
}
```

***Local* memory is invalid after the function call has ended!**

What would the value the caller gets be pointing to?

Common Error: Returning a Pointer to a Local Variable

A solution would be to pass
in an array to hold the answer:

```
void firstlast(double a[], int size,
               double result)
{
   result[0] = a[0];
   result[1] = a[size - 1];
}
```

Strings

- Are represented as arrays of **`char`** values.

`char` Type and Some Famous Characters

The type **`char`** is used to store an individual character.

`char` Type and Some Famous Characters

Some of these characters are plain old letters and such:

```
char yes = 'y';
char no = 'n';
char maybe = '?';
```

`char` Type and Some Famous Characters

Some are numbers masquerading as digits:

```
char theThreeChar = '3';
```

That is not the number three – it's the *character* 3.

`'3'` is what is actually stored in a disk file when you write the `int` 3.

Writing the variable `theThreeChar` to a file would put the same `'3'` in a file.

`char` Type and Some Famous Characters

So some characters are literally what they are:

`'A'`

Some represent digits:

`'3'`

Some are other things that can be typed:

`'C'`

`'+'`

`'+'`

but…

Some Famous Characters

Some of these characters are “special”:

`'\n'`

`'\t'`

These are still single (individual) characters:

the ***escape sequence*** characters.

Some Famous Characters

And there is one special character that
is especially special to C strings:

The null terminator character:

`'\0'`

That is an escaped zero.
It’s in ASCII position zero.

It is the value `0` (not the character zero, `'0'`)
If you output it to screen nothing will appear.

Some Famous Characters

Table 3 Character Literals

`'y'`	The character y
`'0'`	The character for the digit `0`. In the ASCII code, `'0'` has the value 48.
`' '`	The space character
`'\n'`	The newline character
`'\t'`	The tab character
`'\0'`	The null terminator of a string
`"y"`	**Error:** Not a `char` value

The Null Terminator Character and C Strings

The null character is special to C strings because it is always the last character in them:

`"CAT"` is really this sequence of characters:

`'C' 'A' 'T' '\0'`

The null terminator character indicates the end of the C string

The Null Terminator Character and C Strings

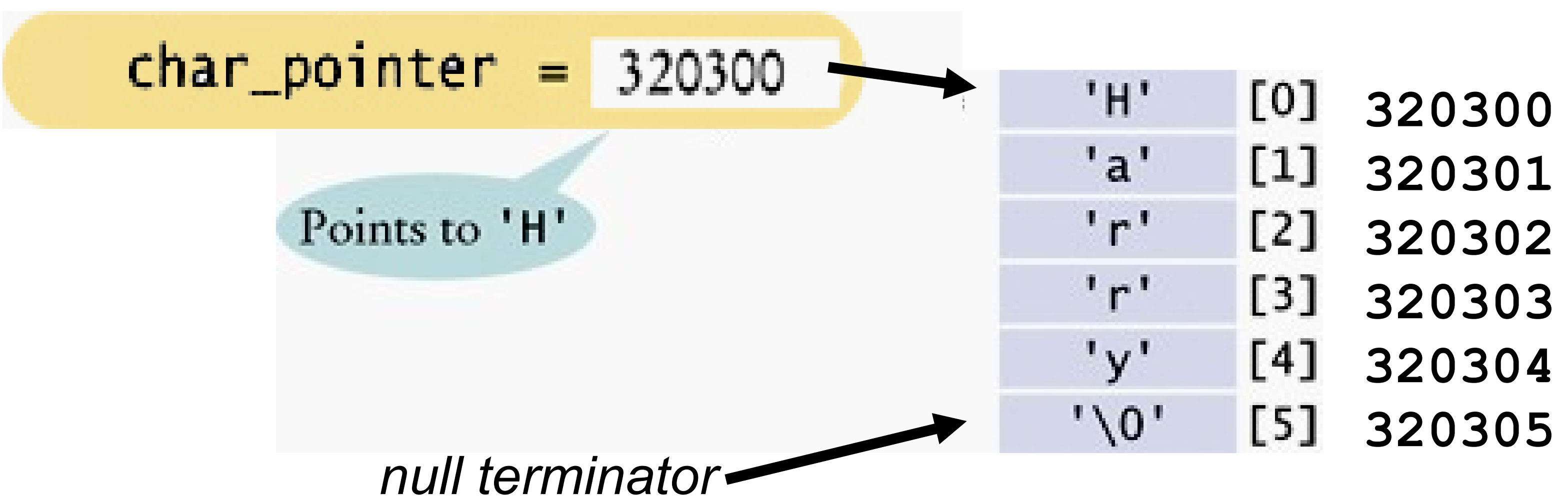
The literal C string **`"CAT"`** is actually an array of ***four*** **`char`**s stored somewhere in the computer.

In the C programming language, literal strings are always stored as character arrays.

Character Arrays as Storage for C Strings

As with all arrays, a string literal can be assigned to a pointer variable that points to the initial character in the array:

```
char* char_pointer = "Harry";
      // Points to the 'H'
```



Using the Null Terminator Character

Functions that operate on C strings rely on this terminator.

The strlen function returns the length of a C string.

```
int strlen(const char s[])
{
   int i = 0;
   // Count characters before
   // the null terminator
   while (s[i] != '\0') {
      i++;
   }
   return i;
}
```

Using the Null Terminator Character

The call `strlen("Harry")` returns 5.

The null terminator character is not counted as part of the “length” of the C string – but it’s there.

Literal C strings are considered constant.

You are not allowed to modify its characters.

Character Arrays

If you want to modify the characters in a C string,
define a character array to hold the characters instead.

For example:

```
// An array of 6 characters
char char_array[] = "Harry";
```

Isn't something missing?

Character Arrays

The compiler counts the characters in the string that is used for initializing the array, including the null terminator.

```
char char_array[] = "Harry";
```

(6)

Character Arrays

You can modify the characters in the array:

```
char char_array[] = "Harry";
char_array[0] = 'L';
```

C String Functions

Table 4 C String Functions

In this table, `s` and `t` are character arrays; `n` is an integer.	
Function	**Description**
`strlen(s)`	Returns the length of `s`.
`strcpy(t, s)`	Copies the characters from `s` into `t`.
`strncpy(t, s, n)`	Copies at most `n` characters from `s` into `t`.
`strcat(t, s)`	Appends the characters from `s` after the end of the characters in `t`.
`strncat(t, s, n)`	Appends at most `n` characters from `s` after the end of the characters in `t`.
`strcmp(s, t)`	Returns `0` if `s` and `t` have the same contents, a negative integer if `s` comes before `t` in lexicographic order, a positive integer otherwise.