© Suzanne Tucker/iStockphoto.

Chapter Seven: Pointers and Structures

Chapter Goals

- To be able to declare, initialize, and use pointers
- To understand the relationship between arrays and pointers
- To be able to convert between string objects and character pointers
- To become familiar with dynamic memory allocation and deallocation
- To use structures to aggregate data items

Topic 1

1. Defining and using pointers
2. Arrays and pointers
3. C and C++ strings
4. Dynamic memory allocation
5. Arrays and vectors of pointers
6. Problem solving: draw a picture
7. Structures
8. Pointers and structures

Pointers

A variable *contains* a value,

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

A pointer denotes the
memory location of a variable

Pointer Usages

- 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)

Harry Needs a Banking Program

Harry wants a program to manage bank deposits and withdrawals.

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

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

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 his account balance.
It should be initialized to 0 since there is no money yet.

```
double harrys_account = 0;
```

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`** :

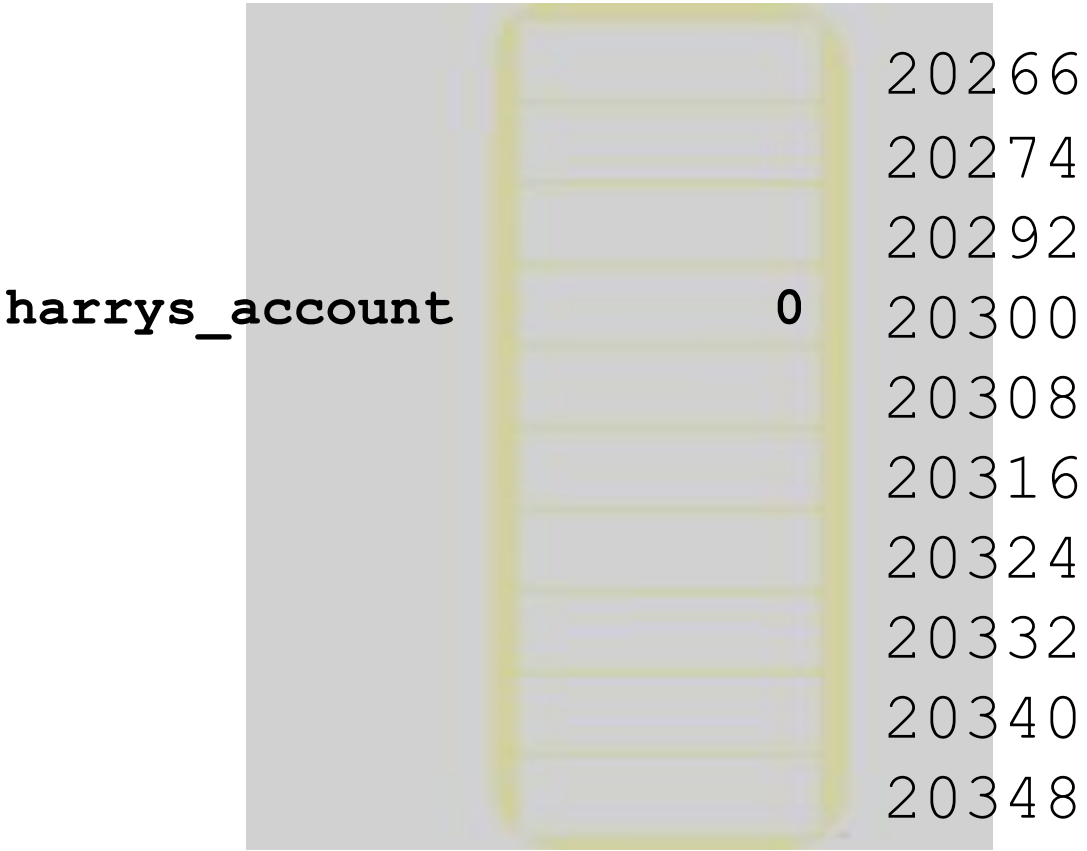
```
double* account_pointer;
```

The type of this variable is “pointer to double”.

Addresses and Pointers

Every byte in RAM has an address as pictured here (this small RAM block is addressed 20266 through 20348, shown in groups of eight bytes)

`harrys_account` as a `double`, happens to be located at address **`20300`**.



Pointer Initialization

When Harry declares a pointer variable, he initializes it to point to **`harrys_account`**:

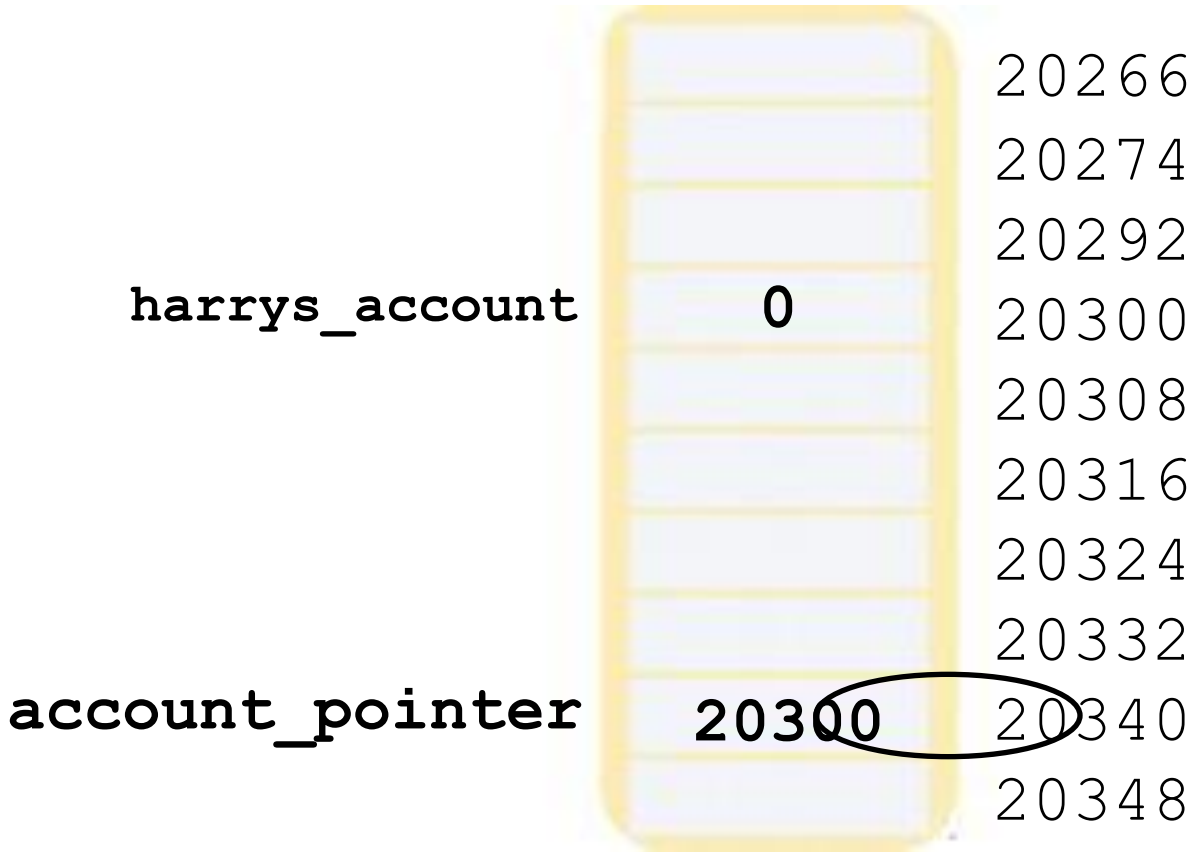
```
double harrys_account = 0;
double* account_pointer = &harrys_account;
```

- The **&** operator yields the location (address) of a variable.
- Taking the address of a `double` variable yields a value of type `double*` so everything fits together nicely.

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

Pointers Also Reside in RAM

And, of course, **`account_pointer`** is *somewhere* in RAM, though we really don't care where it is:



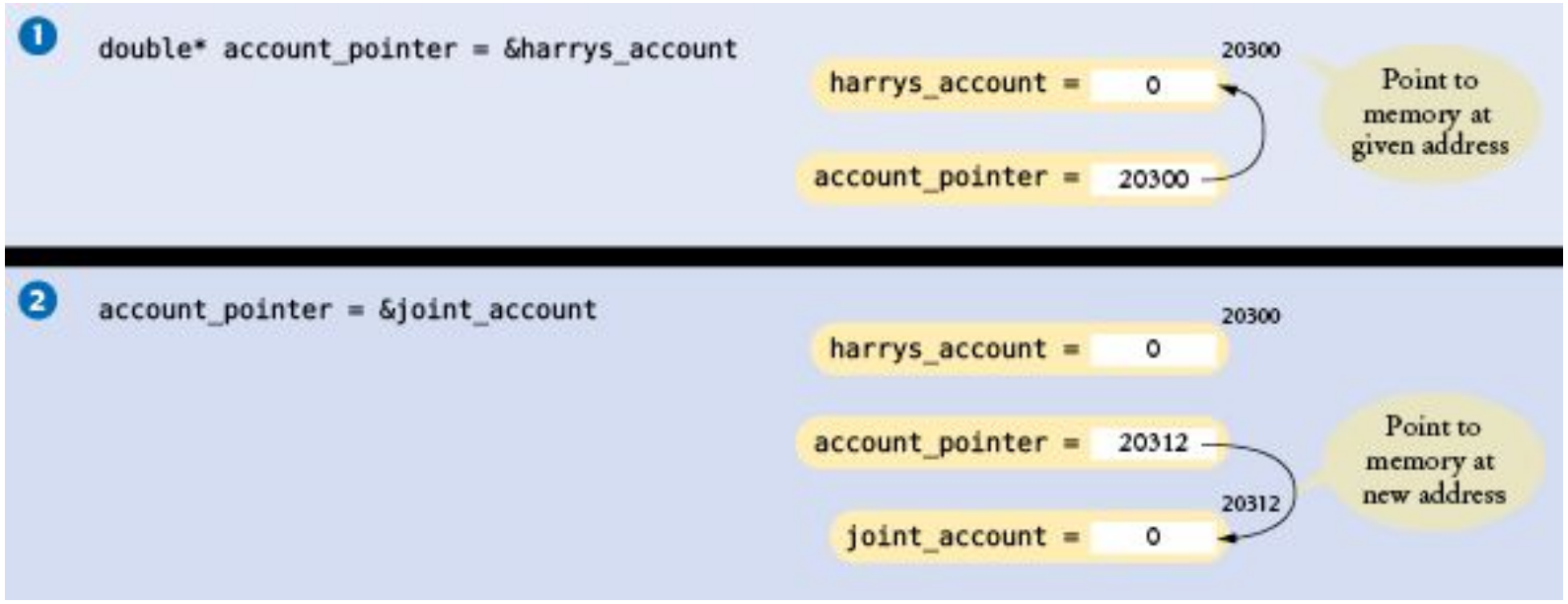
Addresses and Pointers

Harry wanted to use his account, but he found the balance was zero:

```
double harrys_account = 0;
account_pointer = &harrys_account; //Picture #1
 double joint_account = 1000;
```

To access his joint account hoping it still has a non-zero balance, Harry would change the pointer:

```
account_pointer = &joint_account; //Picture #2
```



Addresses and Pointers – and ARROWS

Do note that the computer stores numbers,

not arrows.

Accessing the Memory Pointed to by A Pointer Variable

The "dereferencing operator" * lets you use a pointer to get the data. Use **`*account_pointer`** as a substitute for the name of the variable the pointer points to:

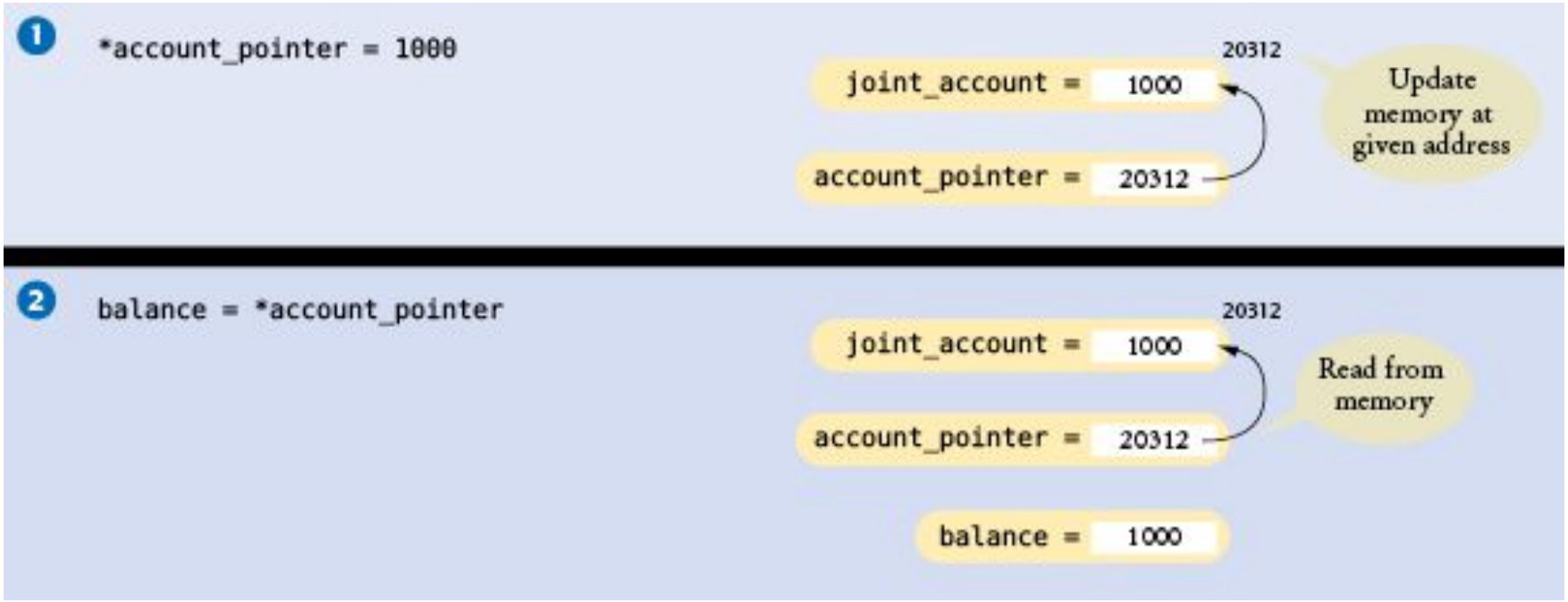
```
// display the current balance
cout << *account_pointer << endl;
```

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

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

Harry Makes the Deposit

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



Pointer Syntax Examples: Table 1, part 1

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;`	p gets 20304	Set p to the address of n.
`*p`	20	The value stored at the changed address.
`m = *p;`	m gets 20	Stores 20 into m.

Pointer Syntax Examples: Table 1, part 2: Bad Syntax

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
`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	Warning: This is the location of a pointer variable, not the location of an integer. You almost never want to use the address of a pointer variable.
`double x = 0; p = &x;`	**Error**	p has type int*, &x has type double*. These types are incompatible.

Errors Using Pointers – Uninitialized Pointer Variables

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

Using that pointer (and its random address) is an error, until the pointer has been initialized.

```
double* account_pointer; // Forgot to initialize
*account_pointer = 1000; // ERROR! account_pointer
// contains an unpredictable value, program crashes
```

If you don't already know what the pointer will point to, initialize it with `nullptr`:

```
double* account_pointer = nullptr;
```

Trying to access data through a `nullptr` pointer will cause your program to terminate (but more gracefully than an uninitialized pointer would).

Harry's Banking Program, part 1

```
// Here is the complete banking program
#include <iostream>
using namespace std;

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

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

   // Print balance
   cout << "Balance: " << *account_pointer << endl;
```

Harry's Banking Program, part 2

```
   // 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 (of joint account)
   cout << "Balance: " << *account_pointer << endl;

   return 0;
}
```

Practice It

Two groups jointly charter a bus and fill it with travelers. A variable

```
int count = 0;
```

is to be accessed through two pointers `p` and `q`.

1. Declare the pointer variable `p`. Do not initialize:
2. Initialize p with the address of `count`:
3. Complete this statement to check whether there is space in the bus for another passenger, using the pointer `p`:
   - `if (_____ < CAPACITY)`
4. Increment the value to which `p` points, using `++`:

5. Declare the pointer variable `q` and initialize it with `p`:

Practice It More

Show the output of each of these code snippets. Answer "?" if the output cannot be determined:

```
int a = 1;
int b = 2;
int* p = &a;
cout << *p << " ";    ____________________
p = &b;
cout << *p << endl;   ____________________
```

```
int a = 15;
int* p = &a;
int* q = &a;
cout << *p + *q << endl;   ____________________
```

```
int a = 15;
int* p = &a;
cout << *p << " " << p << endl;   ________________
```

Common Error: Confusing Data And Pointers: Where's the *?

```
double* account_pointer = &joint_account;
account_pointer = 1000; // ERROR !
```

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: Multiple Pointers Defined in a Single Statement

It is legal 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;
```

Alternatively, you can move the * next to the variable name:

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

Function Arguments: Pointers vs. References

Recall that the `&` symbol is used for reference parameters:

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

A call of this function would be:

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

We can accomplish the same thing using pointers:

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

But the call will have to feed the function an address (pointer variable or reference):

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