Chapter Seven: Pointers, Part I

Slides by Lyan Gallagher

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

Pointers (7.1)

A variable *contains* a value,

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

A pointer denotes the
memory location of a variable

a = 25

Value of a is 25

where it is stored would be a pointer

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)

Chp 9

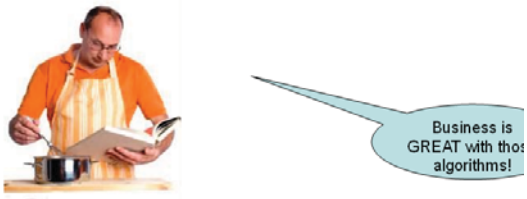
A Banking Problem

Consider a person.
A chef.

(Harry)

Harry Needs a Banking Program

Harry has more than one bank account.



Harry Needs a Banking Program

Harry wants a program for making bank deposits and withdrawals.

(You can write that code by now!)

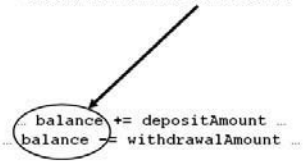
```
... balance += depositAmount ...
... balance -= withdrawalAmount ...
```

balance = balance - withdrawalAmount

-=

Harry Needs a Multi-Bank Banking Program

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



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.

(Ah, code reuse!)

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

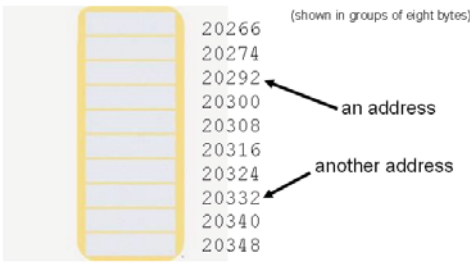
A *pointer to double* type can hold the address of a `double`.

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)

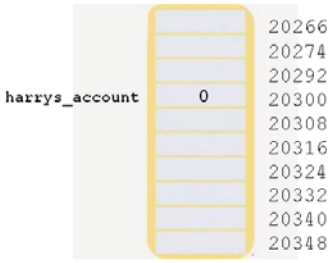


Addresses and Pointers

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

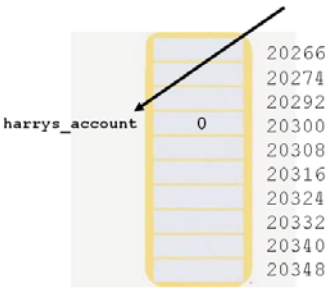
Addresses and Pointers

But really it's been like this all along:



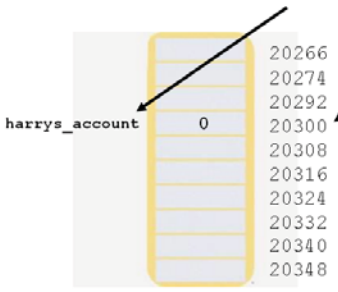
Addresses and Pointers

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;
```

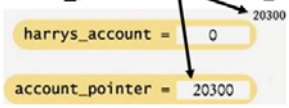
The `&` operator yields the location (or address) of a variable.

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`

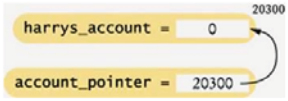
```
double harrys_account = 0;
double* account_pointer = &harrys_account;
```



Pointers to the Rescue

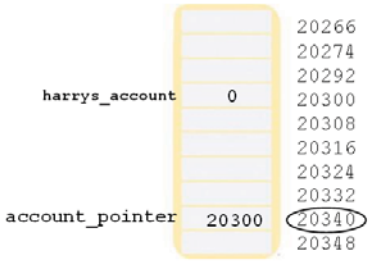
`account_pointer` now "points to" `harrys_account`

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



Addresses and Pointers

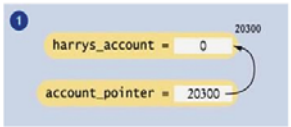
And, of course, `account_pointer` is *somewhere* in RAM:



Addresses and Pointers

To access a different account, Harry (and you) would change the pointer value stored in `account_pointer`:

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

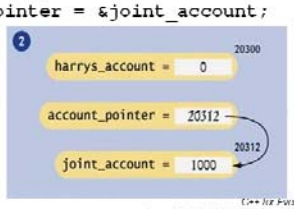


Harry (and you) would use `account_pointer` to access `harrys_account`.

Addresses and Pointers

To access a different account, like `joint_account`, Harry (and you) would 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;
```



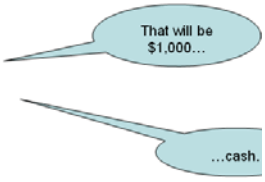
Addresses and Pointers – and ARROWS

Do note that the computer stores numbers,

not arrows.

Harry Sells An ALGORITHMMMMMCAKE

Harry makes his first ALGORITMMMMMCAKE sale.



– And Deposits the Money

Harry needs to depost this cash into his account
– into the `harrys_account` variable

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 access the memory location associated with a pointer (get the value in that location).

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
cout << *account_pointer << endl;
```

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

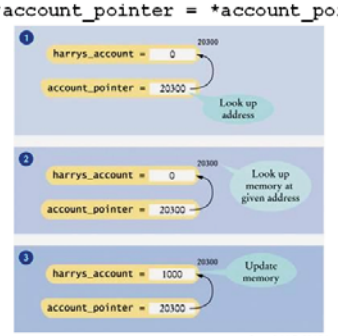
```
// withdraw $100
*account_pointer = *account_pointer - 100;
```

Could use -= here

(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
{
   cout << *account_pointer;
}
```

need to do this!

NULL

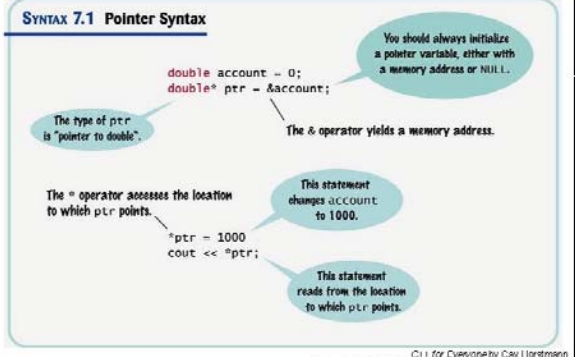
Trying to access data through a NULL pointer is still illegal,
and
it will cause your program to crash.

```
double* account_pointer = NULL;
cout << *account_pointer;
```

CRASH!!!

Syntax of Pointers

Syntax 7.1 Pointer Syntax



Pointer Syntax Examples

STOP

Table 1 Pointer Syntax Examples

Assume the following declarations:
```
int n = 10; // Assumed to be at address 20300
int m = 20; // Assumed to be at address 20304
int* p = &n;
```

Expression	Value	Comment
p	20300	The address of n.
*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.
n = p;	Error	n 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 that Harry wrote. It demonstrates the use of a pointer variable to allow *uniform access* to variables.

ch07/accounts.cpp

```
#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
```


Harry's Banking Program

ch07/accounts.cpp

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

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

   // 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
   cout << "Balance: " << *account_pointer
      << endl;

   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