Pointers

Pointers, pointers, pointers!

CIS 15 : Spring 2007

Functionalia

HW 3 is DUE this SUNDAY, 11:59 pm (Any ?'s)

HW 4 is OUT and PART A is Do April 1st, 11:59pm

HELP ROOM:

http://bridges.brooklyn.cuny.edu/room.html

TEA: Right after class! 1:30 to 3:30 PM, 0317N

Today:

- UNIX Web-space
- More Pointers

Working in UNIX?

Your UNIX account comes enabled to serve up simple web-pages.

Here is how you get started.

1. Login

2. Create a directory named **`public_html`** in your home directory (only need to do this once!)

```
$ mkdir public_html
```

3. Move the things you want to see into your **`public_html`**

```
$ mv sample.ps ~/public_html
```

4. Set the permissions of the things!

```
$ cd public_html
```

```
$ chmod a+r sample.ps
```

5. Point your web-browser to them... (Replace <login> with your login id)

`http://acc6.its.brooklyn.cuny.edu/~<login>/sample.ps`

Declaring and Initializing a Pointer Variable

```
int *ptr; // Need to specify the TYPE of data pointing to

...

int * ptr; // Same as above

...

int* ptr; // Same as above


int x = 25;


ptr = &x; // Now ptr "points" to the variable x.
```

`ptr` still only holds the **memory address** of **`x`**.

Using a Pointer Variable

Access the variable that a pointer points to by using the *de-reference operator* '*****'.

```
int * ptr;

int x = 25;

ptr = &x; // Now ptr "points" to the variable x.


cout << x << " == " << *ptr << endl;


*ptr = 100; // Change the value of x, NOT ptr!!!


cout << x << " == " << *ptr << endl;
```

Using Pointers

You can manipulate a de-referenced pointer in the same way as a normal variable

```
int * ptr;

int x = 0, y = 0, z = 0;

cout << x << ", " << y << ", " << z << endl;


ptr = &x;

*ptr += 100;

ptr = &y;

*ptr += 200;

ptr = &z;

*ptr += 300;



cout << x << ", " << y << ", " << z << endl;
```

Ouput?

And what is `ptr` pointing to at at the END of this code?

Using Pointers

You can manipulate a de-referenced pointer in the same way as a normal variable

```
int * ptr;

int x = 0, y = 0, z = 0;

cout << x << ", " << y << ", " << z << endl;


ptr = &x;

*ptr += 100;

ptr = &y;

*ptr += 200;

ptr = &z;

*ptr += 300;


cout << x << ", " << y << ", " << z << endl;
```

0, 0, 0

100, 200, 300

`ptr` is pointing to the integer z

Do not confuse the many uses of '*'

You have seen 3 uses of the asterisk ('*') so far:

1. Multiplication

```
distance = speed * time;
```

2. Declaring a pointer

```
int * ptr;
```

3. De-referencing a pointer

```
*ptr = 100;
```

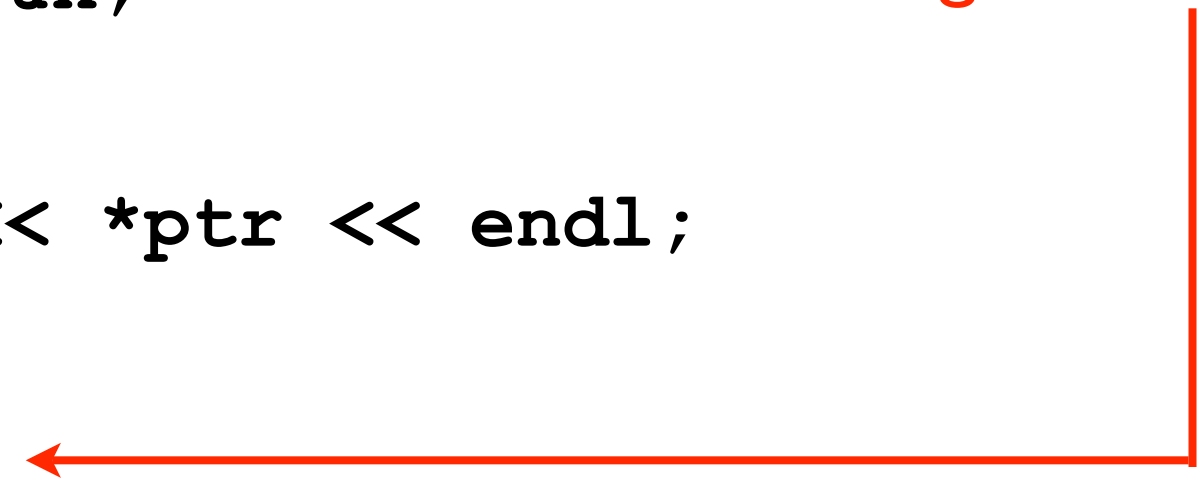
Something to think about...

```
int x = 100000;
int * ptr;

ptr = &x;

cout << *ptr << endl;

ptr++;

cout << *ptr << endl;
```

Something to think about...

```
int x = 100000;
int * ptr;

ptr = &x;

cout << *ptr << endl;

ptr++;

cout << *ptr << endl;
```

You can manipulate what Pointers Point To by doing *Pointer Arithmetic*



Array Names are Constant Pointers

```
int subway_stops[5] = {4, 14, 23, 34, 42};


cout << subway_stops[0] << endl;
```

same as

```
cout << *subway_stops << endl;



cout << subways_stops[2] << endl;
```

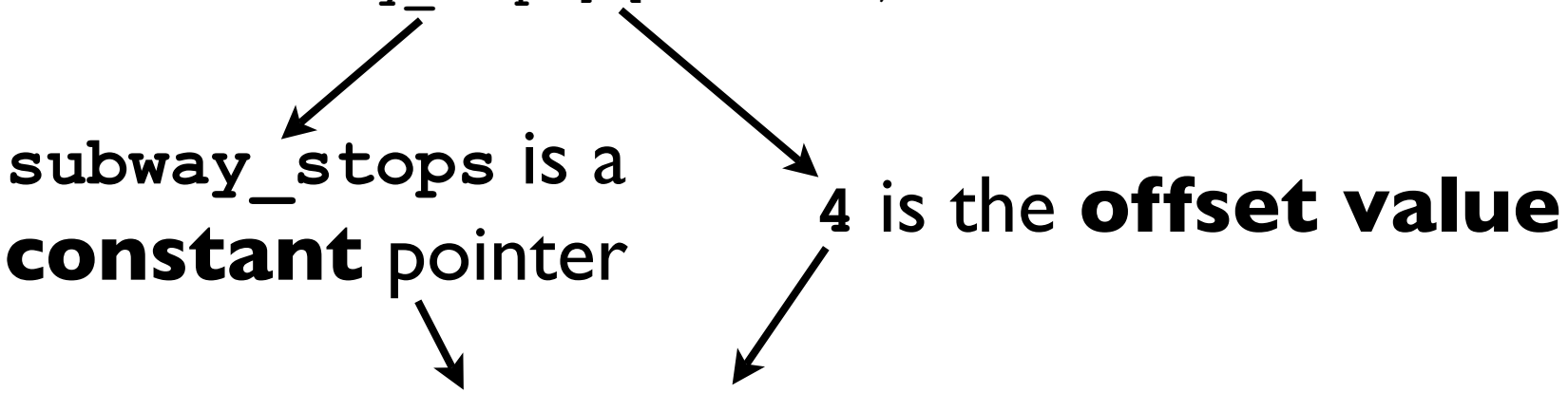
same as

```
cout << *(subways_stops + 2) << endl;
```

Array Names are Constant Pointers

```
int subway_stops[5] = {4, 14, 23, 34, 42};
```

```
cout << subway_stops[4] << endl;
```




```
cout << *(subway_stops + 4) << endl;
```

When you add an **offset value** to a pointer, you are are adding the **offset value** *times the size of the data type of that pointer.*

Why not *subway_stops + 4 ?

```
int subway_stops[5] = {4, 14, 23, 34, 42};

cout << *(subway_stops + 4) << endl;



cout << *subway_stops + 4 << endl;
```

Why not *subway_stops + 4 ?

```
int subway_stops[5] = {4, 14, 23, 34, 42};

cout << *(subway_stops + 4) << endl;
```

42

```
cout << *subway_stops + 4 << endl;
```

4 + 4 = 8

Pointers can Point to Array Elements

```
int subway_stops[5] = {4, 14, 23, 34, 42};

int * A_train;

A_train = subway_stops;

cout << *A_train << endl;

cout << *(A_train + 4) << endl;

A_train++;

cout << *A_train << endl;
```

Pointers can Point to Array Elements

```
int subway_stops[5] = {4, 14, 23, 34, 42};

int * A_train;

A_train = subway_stops;

cout << *A_train << endl;

cout << *(A_train + 4) << endl;

A_train++;

cout << *A_train << endl;
```

Points to beginning of array
(no & is needed)

4

42

14

Pointers can Point to Array Elements

```
int subway_stops[5] = {4, 14, 23, 34, 42};
```

```
for(int * A_train = subway_stops; A_train; A_train++)
{
   cout << *A_train << endl;
}
```

So, what is this?

Pointers can Point to Array Elements

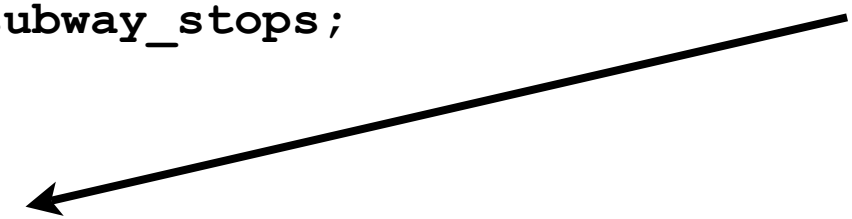
```
int subway_stops[5] = {4, 14, 23, 34, 42};
```

```
for(int * A_train = subway_stops; A_train; A_train++)
{
   cout << *A_train << endl;
}
```

A run-away A Train!

Array names are Pointer Constants

```
int subway_stops[5] = {4, 14, 23, 34, 42};

int * A_train;

A_train = subway_stops;

A_train+=2;

subway_stops = A_train;
```



What Integer Does `A_train` Point to now?

What Integer Does `subway_stops` Point to now?

Array names are Pointer Constants

```
int subway_stops[5] = {4, 14, 23, 34, 42};

int * A_train;

A_train = subway_stops;

A_train+=2;

subway_stops = A_train;
```

What Integer Does `A_train` Point to now?

23

What Integer Does `subway_stops` Point to now?

ERROR!
Array names are Constants!

Initialization of Pointers

1.
```
int myValue;
int * ptr = &myValue;
```

2.
```
int manyValues[20];
int * ptr2 = manyValues;
```

3.
```
float anotherValue;
int * ptr3 = anotherValue;
```

4.
```
int quickValue, *ptr4 = &quickValue;
```

5.
```
float alotOfValues[20], * ptr5 = alotOfValues;
```

6.
```
int * ptr6 = &myFutureValue;
int myFutureValue;
```

Guess: Which of these ways of initializing these pointers are Legal?

Initialization of Pointers

1.
```
int myValue;
int * ptr = &myValue;
```

2.
```
int manyValues[20];
int * ptr2 = manyValues;
```

3.
```
float anotherValue;
int * ptr3 = anotherValue;
```
Error! Type mismatch!

4.
```
int quickValue, *ptr4 = &quickValue;
```

5.
```
float alotOfValues[20], * ptr5 = alotOfValues;
```

6.
```
int * ptr6 = &myFutureValue;
int myFutureValue;
```
Error! Value not initialized yet!

Pointers can be Compared

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};

int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
```

```
if(4_train > 6_train)
   cout << "True";
else
   cout << "False";
```

Guess: True or False?

Pointers can be Compared

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};

int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
```

```
if(4_train > 6_train)
   cout << "True";
else
   cout << "False";
```

True, `4_train` points to a HIGHER memory address than `6_train`

Pointers can be Compared

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};

int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
```

```
if(*4_train > *6_train)
   cout << "True";
else
   cout << "False";
```

Guess: True or False?

Pointers can be Compared

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};

int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
```

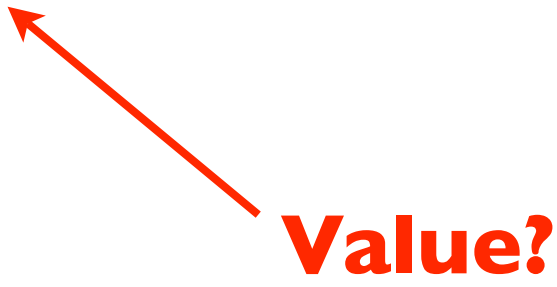
```
if(*4_train > *6_train)
   cout << "True";
else
   cout << "False";
```

False, `4_train` value is 86

`6_train` value is 125

Pointers can be Subtracted

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};


int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
6_train++;
4_train += 3;
6_train++;


int stopsAway = 4_train - 6_train;
```



Pointers can be Subtracted

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};


int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
6_train++;
4_train += 3;
6_train++;


int stopsAway = 4_train - 6_train;
```

Count the number of Integers

8 integers - 2 integers = 6

What does this mean?

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};


int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
6_train++;
4_train += 3;
6_train++;


int meaning = 4_train + 6_train;
```

What does this mean?

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};

int * 4_train = all_stops, * 6_train = all_stops;
4_train += 5;
6_train++;
4_train += 3;
6_train++;

int meaning = 4_train + 6_train;
```

Undefined! Error: one does not add pointers.

Resulting address is beyond the bounds of the array.

Find the index

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};


int * 4_train = all_stops;
4_train += 5;


cout << 4_train << endl;
```

86

In the array `all_stops`, what is the index of 86?

How can we always find that array index?

Find the index

```
int all_stops [15] = {125, 116, 110, 103, 96, 86, 77, 68,
                      59, 51, 42, 33, 28, 23, 14};


int * 4_train = all_stops;
4_train += 5;

cout << 4_train << endl;
```

86

In the array `all_stops`, what is the index of 86?

```
int index = 4_train - all_stops;
```

Pointers as Function Parameters

A pointer can be used as a function parameter.

It gives the function access to the original argument, much like what a reference parameter does.

```
void doubleValue(int *val)

{

   *val *= 2;

}



int main()

{

   int number = 5;

   doubleValue(&number);

}
```

Pointers as Function Parameters

A pointer can be used as a function parameter.

It gives the function access to the original argument, much like what a reference parameter does.

```
void doubleValue(int *val)
{
   *val *= 2;
}


int main()
{
   int number = 5;
   doubleValue(&number);
}
```

* is used to *pass the argument by reference*

* is used to *dereference the variable*

& is used to pass the memory address of the variable `number` to the function

Pointers as Function Parameters

Reference variables hide all of the "mechanics" of dereferencing and indirection.

```
void doubleValuePtr(int *val)
{
  *val *= 2;
}


int main()
{
  int number = 5;
  doubleValuePtr(&number);
}
```

EQUIVALENT

```
void doubleValueRef(int &val)
{
  val *= 2;
}


int main()
{
  int number = 5;
  doubleValueRef(number);
}
```

Pointers as Function Parameters

What variables are being passed by reference?

```
void prompt(int *choice)
{
   cout << "What is your choice: ";
   cin >> *choice;
}

int main()
{
   int menuOption;

   prompt(&menuOption);
}
```

Array Names are the Same as Pointers

```
double classAvg(double * grades, int size)
{
   double sum = 0.0;
   double * gPtr = grades;

   for(int count = 0; count < size; count++)
   {
      sum += *gPtr;
      gPtr++;
   }
   return (sum / size);
}
```

```
double grades[5] = {66.6, 50.5, 76.5, 34.4, 98.1};
cout << "Class average is: " << classAvg(grades, 5) << endl;
```

Equivalent to...

```
double classAvg(double grades[], int size)
{
   double sum = 0.0;
   double * gPtr = grades;

   for(int count = 0; count < size; count++)
   {
      sum += *gPtr;
      gPtr++;
   }
   return (sum / size);
}
```

```
double grades[5] = {66.6, 50.5, 76.5, 34.4, 98.1};
cout << "Class average is: " << classAvg(grades, 5) << endl;
```

More than one way of stepping through an array

```
double classAvg(double * grades, int size)
{
   double sum = 0.0;
   double * gPtr = grades;

   for(int count = 0; count < size; count++)
   {
      sum += *(gPtr + count);
   }
   return (sum / size);
}
```

```
double grades[5] = {66.6, 50.5, 76.5, 34.4, 98.1};
cout << "Class average is: " << classAvg(grades, 5) << endl;
```

Const variables need Pointers to a Const

```
const int SIZE = 3;
const double payRates[SIZE] = {12.11, 20.34, 34.32};


displayPayRates(payRates, SIZE);
```

Prototype:

```
void displayPayRates(const double *rates, int size);
```

Not necessary. Why?

Const variables need Pointers to a Const

```
const int SIZE = 3;
const double payRates[SIZE] = {12.11, 20.34, 34.32};

displayPayRates(payRates, SIZE);
```

```
void displayPayRates(const double *rates, int size);
```

Write the code to display the Pay Rates in Dollars ($)
Using POINTERS (*) AND NOT ARRAYS ([])

Constant Pointers

A pointer ITSELF can be a constant.

```
int valueA = 29;
int valueB = 35;
int * const ptr = &valueA;

ptr = &valueB;    // error: assignment of read-only variable 'ptr'
*ptr = valueB;    // OK!

ptr++;            // error: increment of read-only variable 'ptr'
(*ptr)++;           // OK!
```

What is this?

```
int value = 29;
const int * const ptr = &value;

cout << *ptr << endl;

ptr++;

(*ptr)++;
```

What is this?

```
int value = 29;
```

```
const int * const ptr = &value;
```

A constant pointer to a constant variable.

(Note that the variable itself can be a non-constant, but the `const int` protects it)

```
ptr++;
```

Error!

```
(*ptr)++;
```

Error!

Exercises

```
int array[5] = {1,2,3,4,5};
```

```
//A
if(array < &array[1])
    cout << "True";
else
    cout << "False";
```

```
//B
if(&array[4] < &array[1])
    cout << "True";
else
    cout << "False";
```

```
//C
if(array != &array[2])
    cout << "True";
else
    cout << "False";
```

```
//D
if(array != &array[0])
    cout << "True";
else
    cout << "False";
```

Exercises

Write a function that takes a pointer to an integer as its input and converts it to a negative integer (only if it is positive, however)!

Here is the prototype to help you get started:

```
void makeNegative(int *val);
```

Write some code the demonstrates the use of this function.