Pointer Basics

What is a Pointer?

A pointer is a variable that stores a memory address. Pointers are used to store the addresses of other variables or memory items. Pointers are very useful for another type of parameter passing, usually referred to as **Pass By Address**. Pointers are essential for dynamic memory allocation.

Declaring pointers:

- Pointer declarations use the * operator. They follow this format:

```
  typeName * variableName;

int n;        // declaration of a variable n
int * p;      // declaration of a pointer, called p
```

- In the example above, p is a pointer, and its type will be specifically be referred to as "pointer to int", because it stores the address of an integer variable. We also can say its type is: `int*`
- The type is important. While pointers are all the same size, as they just store a memory address, we have to know **what** kind of thing they are pointing TO.

```
double * dptr;         // a pointer to a double
char * c1;             // a pointer to a character
float * fptr;          // a pointer to a float
```

- **Note:** Sometimes the notation is confusing, because different textbooks place the * differently. The three following declarations are equivalent:

```
int *p;
int* p;
int * p;
```

  All three of these declare the variable p as a pointer to an int.

- Another tricky aspect of notation: Recall that we can declare mulitple variables on one line under the same type, like this:

```
int x, y, z;           // three variables of type int
```

  Since the type of a "pointer-to-int" is `(int *)`, we might ask, does this create three pointers?

```
int* p, q, r;          // what did we just create?
```

  NO! This is not three pointers. Instead, this is one pointer and two integers. If you want to create mulitple pointers on one declaration, you must repeat the * operator each time:

```
int * p, * q, * r;             // three pointers-to-int
int * p, q, r;                 // p is a pointer, q and r are ints
```

Notation: Pointer dereferencing

- Once a pointer is declared, you can refer to the thing it points to, known as the **target** of the pointer, by "**dereferencing the pointer**". To do this, use the unary `*` operator:

```
int * ptr;            // ptr is now a pointer-to-int

// Notation:
//    ptr     refers to the pointer itself
//    *ptr    the dereferenced pointer -- refers now to the TARGET
```

- Suppose that `ptr` is the above pointer. Suppose it stores the address 1234. Also suppose that the integer stored at address 1234 has the value 99.

```
cout << "The pointer is: " << ptr;     // prints the pointer
cout << "The target is: " << *ptr;     // prints the target

// Output:
//  The pointer is: 1234               // exact printout here may vary
//  The target is: 99
```

  Note: the exact printout of an addres may vary based on the system. Some systems print out addresses in hexadecimal notation (base 16).
- **Note:** The notation can be a little confusing.
  - If you see the * in a *declaration statement*, with a type in front of the *, a pointer is being declared for the first time.
  - AFTER that, when you see the * on the pointer name, you are dereferencing the pointer to get to the target.
- Pointers **don't always have valid targets**.
  - A pointer refers to some address in the program's memory space.
  - A program's memory space is divided up into **segements**
  - Each memory segment has a different purpose. Some segments are for data storage, but some segments are for other things, and off limits for data storage
  - If a pointer is pointing into an "out-of-bounds" memory segment, then it does **NOT** have a valid target (for your usage)
  - **IMPORTANT**: If you try to dereference a pointer that doesn't have a valid target, your program will crash with a *segmentation fault* error. This means you tried to go into an off-limits segment
- Don't dereference a pointer until you know it has been initialized to a valid target!

pdeclare.cpp -- an example illustrating the declaration and dereferencing of pointers

Initializing Pointers

So, how do we initialize a pointer? i.e. what can we assign into it?

```
int * ptr;
```

```
ptr = ______;          // with what can we fill this blank?
```

Three ways are demonstrated here. (There is a 4th way, which is the most important one. This will be saved for later).

The null pointer

- There is a special pointer whose value is 0. It is called the *null pointer*
- You **can** assign 0 into a pointer:

  ```
  ptr = 0;
  ```

- The null pointer is the *only* integer literal that may be assigned to a pointer. You may NOT assign arbitrary numbers to pointers:

  ```
  int * p = 0;     // okay.  assignment of null pointer to p
  int * q;
  q = 0;           // okay.  null pointer again.
  int * z;
  z = 900;         // BAD!  cannot assign other literals to pointers!
  double * dp;
  dp = 1000;       // BAD!
  ```

- The null pointer is **never** a valid target, however. If you try to dereference the null pointer, you WILL get a segmentation fault.
- So why use it? The null pointer is typically used as a placeholder to initialize pointers until you are ready to use them (i.e. with valid targets), so that their values are **known**.
  - If a pointer's value was completely *unknown* -- random memory garbage -- you'd never know if it was safe to dereference
  - If you make sure your pointer is ALWAYS set to either a valid target, or to the null pointer, then you can test for it:

    ```
    if (ptr != 0)            // safe to dereference
       cout << *ptr;
    ```

- pnull.cpp -- example illustrating the null pointer

Pointers of the same type

- It is also legal to assign one pointer to another, provided that they are the same type:

  ```
  int * ptr1, * ptr2;           // two pointers of type int

  ptr1 = ptr2;                  // can assign one to the other
                                //  now they both point to the same place
  ```

- Although all pointers are addresses (and therefore represented similarly in data storage), we want the type of the pointer to indicate what is being pointed to. Therefore, C treats pointers to different types AS different types themselves.

  ```
  int * ip;             // pointer to int
  ```

```
char * cp;              // pointer to char
double * dp;            // poitner to double
```

- These three pointer variables (ip, dp, cp) are all considered to have different types, so assignment between any of them is illegal. The automatic type coercions that work on regular numerical data types **do not apply**:

```
ip = dp;                // ILLEGAL
dp = cp;                // ILLEGAL
ip = cp;                // ILLEGAL
```

- As with other data types, you can always force a coercion by performing an explicit cast operation. With pointers, you would usually use `reinterpret_cast`. Be careful that you really intend this, however!

```
ip = reinterpret_cast<int* >(dp);
```

The "address of" operator

- Recall, the `&` unary operator, applied to a variable, gives its address:

```
 int x;
 // the notation &x means "address of x"
```

- This is the best way to attach a pointer to an existing variable:

```
 int * ptr;             // a pointer
 int x;                 // an integer
 ptr = &x;              // assign the address of x into the pointer
                        // now ptr points to "x"!
```

- pinit.cpp -- example illustrating pointer initialization with the address-of operator, and of assigning pointers to other pointers