Pointers II

Review

- A ***pointer*** is a data type that holds the address of another variable in memory.
  - It "points to" that variable.
- Uses:
  - Access a variable from more than one place in your program without copying it.
  - Create more sophisticated data structures (later).

Syntax

- Reference operator: `&`*`variable`*
  - Returns the memory address of *`variable`*.
- Declare a pointer:
  - *`type *ptr_var;`*
  - Creates a pointer to the type specified, points nowhere.
  - type can be `int`, `double`, `string`, a class name, ...
- Make a pointer point to a certain variable:
  - `ptr_var = &variable`;
  - Only works if `ptr_var` is declared to point to the same data type as what `variable` is.

Examples

```
int x = 5;
double a = 6.4;

int *p1 = &x;       // OK
int *p2 = &a;       // illegal
double *p3 = &a;    // OK
```

Syntax

- Two ways to change where a pointer points.
  - Use reference operator to point a pointer to a specific variable:
  - `ptr_var = &variable;`

  - Point a pointer to where another pointer points (assign a pointer to another pointer).
  - `ptr_var = other_pointer_var;`

Examples

```
int x = 5, y = 10;

int *p1 = &x;       // p1 points to x
int *p2 = &y;       // p2 points to y
int *p3 = p2;       // p3 also points to y
```

Nothing magic is happening with the last line. The statement copies the value of p2 into p3. Since the value in p2 is an address, this has the same effect as making p3 point to y.

Syntax

- Using the name of a pointer variable gives you a memory address.
- To access the value ***at*** that address, use the dereference operator: *
- The reference operator (&) and dereference operator (*) are inverses of each other:
  - `&var` takes a variable and makes a pointer to it (an address).
  - `*ptr_var` takes a pointer variable and gives you the value of the variable it points to.
- The `*ptr_var` syntax can be used anywhere a regular variable would (print, pass to functions, return from functions, store in a variable, etc).

Examples

```
int x = 5, y = 10, z = 15;

int *p1 = &x, *p2 = &y, *p3 = &z;
cout << *p1 << *p2 << *p3 << endl;
*p1 = 8;
*p2 -= *p1;
cout << *p1 << *p2 << *p3 << endl;
p3 = p2;
*p2 = *p1;
cout << *p1 << *p2 << *p3 << endl;
```

Syntax

- Be careful of the difference between
- `ptr1 = ptr2;`
- `*ptr1 = *ptr2;`
- The first one changes where `ptr1` points to.
- The second one changes the value stored in the variable that `ptr1` points to.

Null pointer

- There is a special address you can use when you want a pointer to point "nowhere:" the ***null pointer***.
  - `NULL` or `nullptr`.
- Good rule of thumb to always set pointers to nullptr if you can't initialize them to something else.
- The null pointer is also used to represent a "missing value" for a pointer.

Examples

```
int *ptr = nullptr;
// do some stuff here
int x = call_some_big_function();
ptr = &x;

// different code:
int *ptr2 = nullptr;
// code here to possibly set ptr2
if (ptr2 == nullptr)
  cout << "Missing value!";
```

Pointers and functions

```
void func(int *fptr) {
  *fptr += 1;
}

int main() {
  int x = 5;
  int *ptr = &x;
  func(ptr);
}
```

Pointers and functions

```
void func(int *fptr) {
  int y = *fptr + 1;
  fptr = &y;
}

int main() {
  int x = 5;
  int *ptr = &x;
  func(ptr);
}
```

Pointers and functions

```
int* func() {
  int y = 10;
  int *fptr = &y;
  return fptr;
}

int main() {
  int *ptr = func();
  cout << *ptr;
}
```

Vectors of pointers

```
vector<int*> vec;
int x = 5, y = 10, z = 15;
vec.push_back(&x);
vec.push_back(&y);
int *ptr1 = vec[0];
int *ptr2 = vec[1];
vec.push_back(ptr1);
*vec[0]++;
vec[1] = vec[2];
*vec[2] = z;
z++;
```

Reversing a vector of pointers

Pointers to objects

- Normally we use the dot operator to access the fields and methods of an object:
  - `dog mydog;`
  - `mydog.setAge(3);`
- If you want to access the fields and methods of an object through a pointer to that object, you should use the arrow operator: **`->`**
  - `dog mydog;`
  - `dog *dogptr = &mydog;`
  - `mydog->setAge(3);`

```
dog lassie;
lassie.setAge(4);
dog rowlf = lassie;
// copies all of lassie's fields to rowlf.
// The two dogs are still 100% separate.
```

```
dog* toto = &lassie;
toto->setAge(6);
// sets lassie's age (toto is just a pointer,
// not a separate standalone dog)
```

Use dot operator when left side is an ***object***.

Use arrow operator when left side is a ***pointer to an object.***