CSCI 135
Pointers

Review - Data Structures

Types:

- Primitive types: int, double, enum, etc.
- User-defined types: using typedef
- Strings
- Vectors
- Arrays
- Structures

Variables/Objects of each of these types are stored as binary data in random access memory (RAM).
Reminder: Don't confuse a type (*e.g.*, `int, struct Date`) with a variable of that type (*e.g.*, `n, today`)

Introduction to Pointers

Recall: entities stored in RAM are accessed by their address

A **pointer** is a variable that stores the address of another variable.

Ex (pointer p points to variable y):

RAM:

Address	**Data**	**Name**	**Type**
...	...	...	...
0x12345678	17	x	int
0x12345674	52	y	int
0x12345670	41	z	int
0x1234566C	0x12345674	p	pointer
...	...	...	...

- Similar to references, except that now we are actually storing the address in a programmer-accessible variable instead of just passing the address around.
- Two ways of accessing y in above example: either directly, or indirectly through p (called **aliases** of y)

Pointer Variables in C/C++

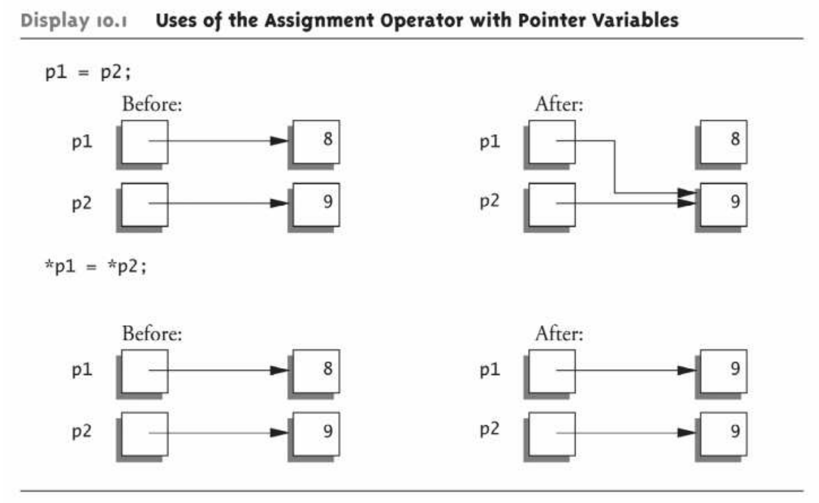
<type> * <pointerVariableName>;

where type is any type. Examples:

```
int * p1;
string * p2;
struct Time {int hours; int minutes;};
Time * p3, * p4;
```

- Read "p1 points to ..."
- The * is important! Ex: `int * p1, v;` declares p1 as a pointer to an int, and v as an int. They are not of the same type and should not be assigned to each other (even though p1 is an address and addresses are integers).
- All pointer variables are typed – p1 can only point to an int
- Pointer types can be typecast (but potentially dangerous)

Example


Display 10.1 Uses of the Assignment Operator with Pointer Variables

from our textbook

Pointer Operators

- Dereferencing (*): *p is the variable pointed to by p (read "the data pointed to by p")
- Address (&): &v is the address of v (NOT same as &p or v)

```
int *p1, *p2, v1, v2;
v1 = 4;
p1 = &v1;        read 'p1 points to v1'
cout << p1;      prints some address - e.g., 0xABCD1230
cout << *p1;     prints 4
*p1 = 5;         stores 5 in v1
(*p1)++;         stores 6 in v1
p2 = &v2;
*p2 = *p1;       stores 6 in v2
p2--;            points p2 to some variable
*p2 = 7;         writes 7 to somewhere in RAM
```

⚠The final assignment could be writing some arbitrary variable or even some OS location (probably a bug)!

Pointers - Initial and Null Values

- A pointer is not initialized ⇒ `*p` is likely an error
- Accessing a null pointer is still an error ⇒ Need to check `p != NULL` before accessing `*p` unless you know whats in `p`!

`NULL` **vs.** `nullptr`**:**

- `NULL` is defined to be an implementation-dependent constant (typically 0) of type int ⇒ ambiguity in resolving overloading (when p is NULL):
  ```
  void func (int *p)
  void func (int i)
  ```
- C++11 (and on): `nullptr` has type `nullptr_t` (castable to other pointer types)

Pointers vs. References

Recall: The & symbol indicates a pass by reference

C: Pass by reference is implemented by passing (by value) the address of the variable. *e.g.*, if x is stored at address 0x12345678 and passed by reference, the assembly code actually passes 0x12345678 by value (so the called function can indirectly change x by writing to RAM address 0x12345678)

BUT some BIG differences:

- A pointer is a variable that stores an address, while a reference is just the address and is not stored in any user accessible variable (and won't necessarily be stored anywhere in RAM).
- Thus, a pointer can be assigned to (*e.g.*, `p1 = p2; p1++;`), unlike references, which can't be changed.
- A pointer might be NULL or `nullptr` (*i.e.*, pointing nowhere)!

Example: Pointer Assignment

```
int *p1, *p2, v1, v2;
v1 = 5; v2 = 6;
p1 = &v1; p2 = &v2;
(*p2)++;             stores 7 in v2
*p2 = *p1;           stores 5 in v2
p2 = *p1;            Try to store int 5 in p2 - Compile Error!
p2 = (int*) *p1;     stores 5 in p2
cout << *p2;         Access Error! Accesses location 5 in RAM
*p2 = p1;            Try to store pointer in int - Compile Error!
int *p3;             Does NOT allocate storage for the int
*p3 = 8;             Access Error! Accesses who-knows-where
```

⚠Declaring a pointer (*e.g.*, p3) allocates storage for the pointer but not the variable pointed to (thus *p3 is meaningless)

Dereferencing Operators

- `*p`: Evaluates to value of what p is pointing to (NOT the address)
- `p->id`: Shorthand for (*p).id where points to a struct with an id field.

```
struct Student {
  int id;
  ...
};
Student s;          Allocates storage for a Student
Student *sp;
sp = &s;            Points sp to that student
sp -> id = 1234;    Updates that student's id
```

Why Pointers?

- Allows access of explicit memory locations (sometimes needed in embedded systems)
- Needed when you dont know how big your data structure is going to be – need to dynamically create new variable in memory accessed through pointer to it (since that variable has no fixed name)
- Needed for efficient access of dynamic data structures – *e.g.*, insertion/deletion in lists (a little later, but mostly CSCI 235).
- Used to implement other dynamically sized data structures such as strings, vectors, . . .

Example Applications:

- Lists of data with ability to efficiently insert/delete elements from middle of list (contrast with vectors/arrays)
- Trees and graphs (*e.g.*, your family tree)

Dynamic Variable Creation

- `new` operator used to create (and allocate space for) a new nameless variable, and return the address of that new variable

```
double *p1, *p2, x;
p2 = &x;
p1 = new double;        allocates storage for a double
*p1 = 2.5;              ... and stores 2.5 there
p1 = p2;
*p1 = 4.5;              stores 4.5 in x
```

⇒ x has 3 aliases at end: x, *p1, and *p2.

Proper Usage of `new` - 1

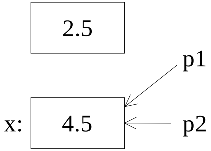
Storage for variables created using `new` is allocated in a dynamic memory area called a **heap** (different from static memory area where automatic variables are stored).
? What if we are out of heap space
! need to Check if `new` was successful by checking that it didn't return NULL.

```
int *p;
p = new int;
if (p == NULL) {
  cout << "Error: Out of heap memory\n";
  exit(1);
}
...
```

BUT, newer compilers *might* abort automatically if `new` fails (still a good practice to check for NULL)

Proper Usage of `new` - 2

```
double *p1, *p2, x;
p2 = &x;
p1 = new double;
*p1 = 2.5;
p1 = p2;
*p1 = 4.5;
```



The 2.5 is stored in memory but no longer accessible by program!

- Called **garbage** – some languages (but not C/C++) have *garbage collection* to identify and reclaim such space.
- Operating systems *should* reclaim garbage after program termination (but we shouldn't count on it)
- Called **memory leak** if space not reclaimed.
- Programmer needs to avoid creating garbage in the first place!

The delete operator

```
int *p;
p = new int(5);    also initializes *p to 5
...                do some processing
delete p;          avoid memory leak
```

Implementation: generated code returns storage allocated for *p to 'free memory' list, so that it can be reused (*e.g.*, by a future new).

⚠The storage allocated to *p is reclaimed, but the pointer variable p still exists and points to the place in memory that once stored 5 ⇒ **dangling pointer**, that might cause problems later if accessed indirectly.

The delete operator - 2

```
int *p;
p = new int(5);
...
delete p;            reclaim storage for *p
p = NULL;            avoid dangling pointer
```

After above, accesses to *p will not update some memory location that has been returned to free space (and possibly reallocated to some other variable, perhaps even an OS variable).

Using Pointers Safely

Guidelines:

- Use delete for dynamically created variables that are no longer needed.
- Use delete for all dynamically created variables before program termination (critical if OS doesn't handle it)
- Assign NULL to pointer after calling delete

Above is not always easy to do since your program will have multiple execution paths (error conditions are particularly problematic).

Pointer Arithmetic

Recall: pointers are typed – *i.e.*, the type, T of entity being pointed to is fixed at declaration time.

Let p be a pointer to type T currently pointing to an entity, x. Suppose n evaluate to an integer.

Overloaded Operators:

- p + n: points to the entity n T's past x
- p - n: points to the entity n T's before x

⇒ if y is of type T and stored immediately after x in RAM, `p++;` would point p to y.

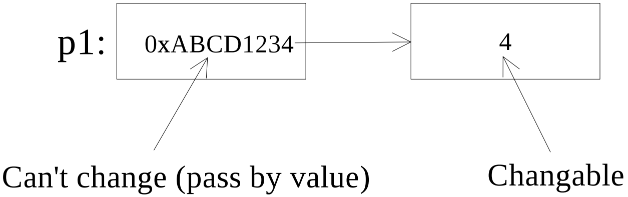
Pointers are integers but not the same as ints due to differences in arithmetic semantics.

⚠Pointer arithmetic is dangerous and can lead to undecipherable code! Avoid if possible.

Pointers as Function Parameters

```
foo(SomeType *p1, SomeType * & p2) {...}
```

- p1 is passed by value:
  - p1 can not change – *i.e.*, the address it stores is the same before and after calling foo.
  - Things pointed to by p1 can still change – *e.g.*, the statement *p1=4 will store 4 somewhere and any accesses to *p1 in the caller will now evaluate to 4.



- p2 is passed by reference: p2 itself can also change.

Passing Pointers to Functions

By Value A copy of the address stored in the pointer variable is passed.

By Ref. The address of the pointer variable (not the variable pointed to) is passed.

```
void foo(int * xval, int * & xref) {
  xval = new int(1);
  xref = new int(2);
  };
int main() {
  int *p1, *p2;
  p1 = new int(5); p2 = new int(6);
  foo(p1,p2);
  cout << *p1 << "␣␣" << *p2 << endl;
}
```

? Output?

Passing Pointers to Functions

By Value A copy of the address stored in the pointer variable is passed.

By Ref. The address of the pointer variable (not the variable pointed to) is passed.

```
void foo(int * xval, int * & xref) {
  xval = new int(1);
  xref = new int(2);
  };
int main() {
  int *p1, *p2;
  p1 = new int(5); p2 = new int(6);
  foo(p1,p2);
  cout << *p1 << "␣␣" << *p2 << endl;
}
```

? Output? 5 2 (also creating garbage)

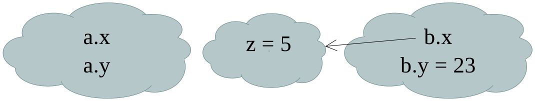
Shallow and Deep Copies

Shallow copy: All parts of an entity are copied, except that for pointer parts, only the pointer is copied, not the entity being pointed to (what C/C++ does for structs)

Deep copy: All parts of an entity are copied, including entities being pointed to.

```
struct SomeStruct {
  int *x;
  int y;
} a,b;
int z=5;
b.x = &z;
b.y = 23;
a = b;
*(b.x) = 8;
```

Before a=b:

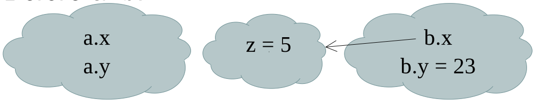


Shallow and Deep Copies

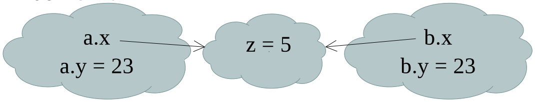
Shallow copy: All parts of an entity are copied, except that for pointer parts, only the pointer is copied, not the entity being pointed to (what C/C++ does for structs)
Deep copy: All parts of an entity are copied, including entities being pointed to.

```
struct SomeStruct {
   int *x;
   int y;
} a,b;
int z=5;
b.x = &z;
b.y = 23;
a = b;
*(b.x) = 8;
```

Before a=b:



After a=b:

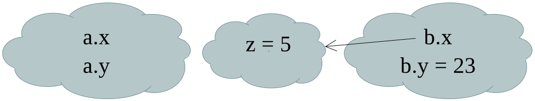


Shallow and Deep Copies

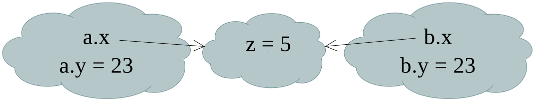
Shallow copy: All parts of an entity are copied, except that for pointer parts, only the pointer is copied, not the entity being pointed to (what C/C++ does for structs)
Deep copy: All parts of an entity are copied, including entities being pointed to.

```
struct SomeStruct {
   int *x;
   int y;
} a,b;
int z=5;
b.x = &z;
b.y = 23;
a = b;
*(b.x) = 8;
```

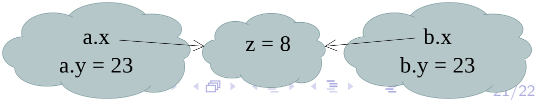
Before `a=b`:



After `a=b`:



At end:



Exercises

1. Declare a pointer (no initialization), and access it using the dereferencing operator. What happens? Your answer will differ across systems.
2. In the first exercise of structs, you declared a variable of type Student. Repeat the exercise, but declaring a variable that points to a Student instead.
3. Write a program that has variables in local and global scope, and all storage classes (some combinations aren't legal). Then print the addresses of all of these variables. Draw any conclusions you have about which regions of memory different kinds of variables (both scope and lifetime) are stored in.