Pointers

CS A150 - C++ Programming 1

Pointers

- **Computer memory** is divided into numbered locations (**bytes**)
  - **Variables** are implemented as a sequence of adjacent bytes.
- A **pointer** is a **memory address** of a variable
  - Specifies *where* the variable is located
    (where the variable starts)
- You have already used pointers:
  - Passing by reference passes the address of a variable, not the actual value.

Addresses and Numbers

- A **pointer** is an **address**
- An **address** is an **integer**
- A pointer is **NOT** an integer!
  - This is **abstraction**
- C++ forces **pointers** be used as **addresses**
  - Cannot be used as numbers
  - Even though a pointer ***is*** a number

Pointer Variables

- **Pointers** are "typed"
  - You can store a pointer in a variable, but a pointer is **<u>not</u>** a type (**`int`**, **`double`**, etc.)
    - A pointer *points to* an **`int`**, **`double`**, etc.
- Example:

```
int *p;
```

- **`p`** is a pointer that can point to an **`int`**
  - Cannot point to anything else
- **`p`** will contain the address of where an integer is located.

Declaring Pointer Variables

- **Pointers** are declared like other types
  - Add ***** *before* the variable name
  - Produces "*pointer to*" that type
- `int *p` is the same as `int* p`

- **Dereference operator ***
  - Pointer variable "***dereferenced***"
  - Means: "Get data that p1 points to"

Common Error

- Declaring two pointers on the same line:

```
int *p1, *p2;
```

  - Both pointers need the (*) operator
  - Writing:

```
int *p1, p2;
```

    - Declares a ***pointer*** **`p1`** and a ***variable*** **`p2`**

POINTING TO…

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

- Sets pointer variable **`p1`** to "point to" **`int`** variable **`v1`**

- Operator **&**
  - Determines " address of " variable

- Read like:
  - "**`p1`** equals address of **`v1`**"
  - Or "**`p1`** points to **`v1`**"

Pointing to… (cont.)

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

- Two ways to refer to **v1** now:
  - Variable **v1** itself:

    ```
    cout << v1;
    ```
  - Via pointer **p1**:

    ```
    cout << *p1;
    ```

& Operator

- The "*address of* " operator **&**
- Also used to specify **call-by-reference parameter**
  - Recall: call-by-reference parameters pass "*address of*" the actual argument.

Example: "Pointing to"

- Consider:

```
int v1, *p1;
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

- Produces output:

```
42
42
```

- **p1** and **v1** refer to same variable.

Pointer Assignments

- Pointer variables can be "assigned":

```
int *p1, *p2;
p2 = p1;
```

  - Assigns one pointer to another
  - "Make **p2** point to where p1 points"

- Do not confuse with:

```
*p1 = *p2;
```

  - Assigns "value pointed to" by **p1**, to "value pointed to" by **p2**

Assigning Same Values

- You can assign the value of one pointer to another pointer variable

```
int *p1, *p2, v;  //declare two pointers and a variable
v = 0;            //variable is equal to 0
p1 = &v;          //pointer holds the address of the var
p2 = p1;          //set p2 to point to v1 as well

cout << v;        // 0
cout << p1;       // 0040FC4C (some address)
cout << p2;       // 0040FC4C  (some address)
cout << *p1;      // 0
cout << *p2;      // 0
```

Example 1

- Pointers

Common Error

- Do ***not*** confuse

```
p1 = p2;
```

  - You are setting **p1** to point to the same address **p2** is pointing to

```
*p1 = *p2;
```

  - You are changing the value of the variable that **p1** is pointing to → it will have the value of the variable **p2** is pointing to

    (both **p1** and **p2** will keep their original address)

THE **new** OPERATOR

- Since pointers can refer to variables…
  - No “real” need to have a standard identifier.
- Can ***dynamically*** allocate variables
  - Operator **`new`** creates variables
    - No identifiers to refer to them
    - Just a pointer!
- Example:

```
int *p1;
p1 = new int;
```

  - Creates new "*nameless*" variable, and assigns `p1` to "point to" it
  - Can access with **`*p1`**
    - Use just like ordinary variable.

Manipulating Pointers

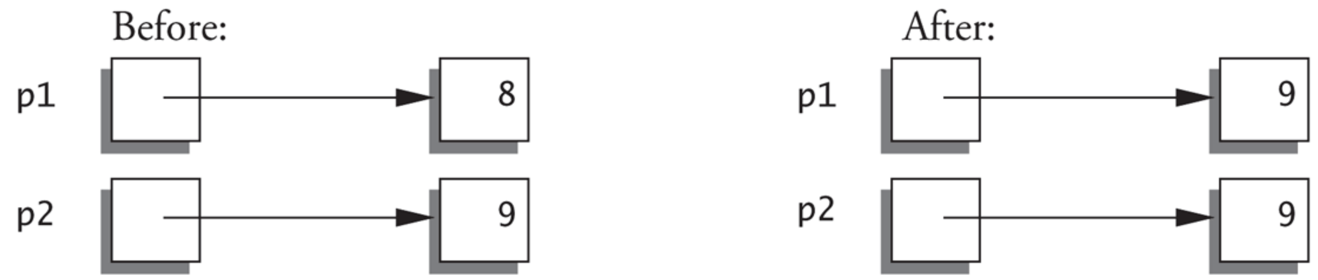
- Dynamic variables are created and destroyed while the program is running

```
//declare a pointer
int *p;                        // also: int *p = new int;

//let p1 point to a new integer
p = new int;

cout << "Enter an integer: ";
//if you want to output the value of the int variable,
//  you do not need to have the variable name since
//  you do have its address
cin >> *p;  //user will provide value

//the same, you may compute calculations
*p = *p + 5;
cout << *p;
```

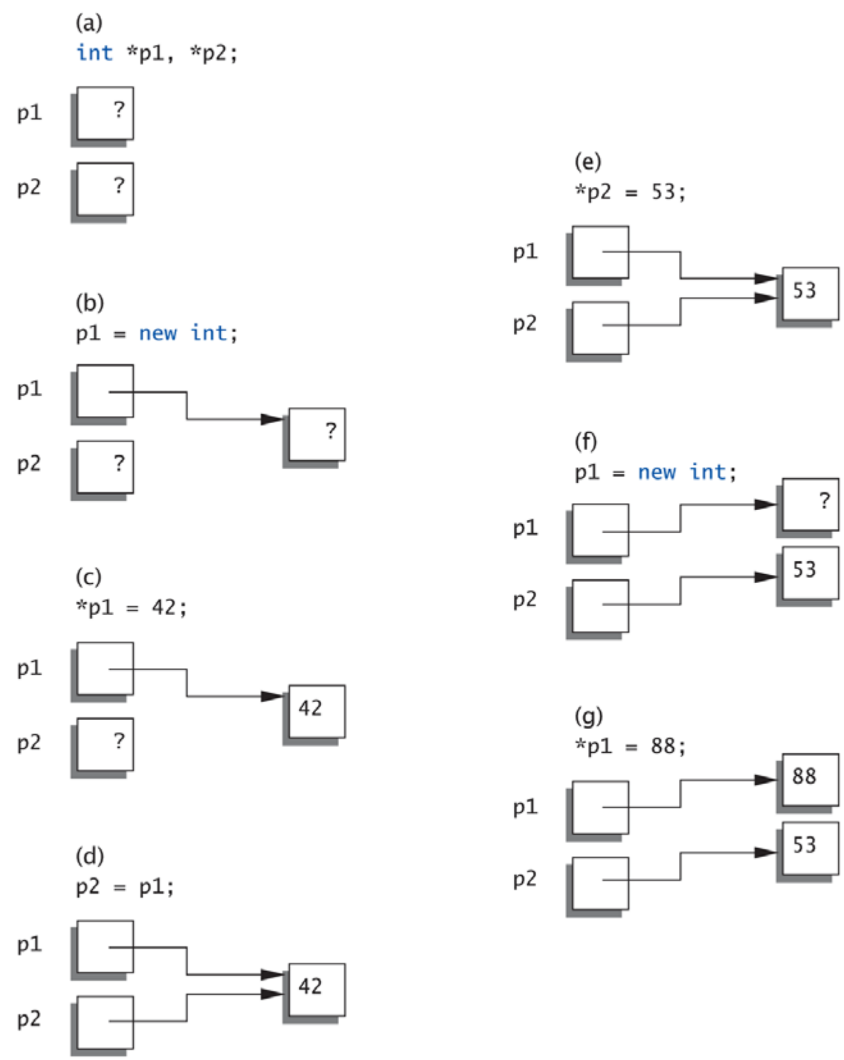
Pointer Assignments

```
p1 = p2;
```


```
*p1 = *p2;
```



EXAMPLE 2: GRAPHICAL REPRESENTATION



Memory Management

- **Heap**
  - Also called "**freestore**"
  - Reserved for ***dynamically-allocated variables***
  - All **new** dynamic variables consume memory in freestore
    - If too many → could use all freestore memory
- Future "new" operations will fail if heap is "full"

HEAP SIZE

- The **size** of the **heap**
  - Varies with implementations
  - Typically large
    - Most program will not use all memory
  - Memory management
    - Still good practice
    - Solid software engineering principle
    - Memory ***is*** finite

CHECKING new CAN BE ALLOCATED

- For ***older*** compilers:
  - Test if **`NULL`** returned by call to new:

```
int *p;
p = new int;
if (p == NULL)
{
      cerr << "Error: Insufficient memory.\n";
      exit(1);
}
```

  - If **`new`** succeeded, program continues.

Newer Compilers

- If **new** operation fails:
  - Program terminates automatically
  - Produces error message
- Still good practice to use **NULL** check.

Example 2

- Pointers and dynamic variables

Pointers and Functions

- **Pointers** are full-fledged types
  - Can be used just like other types
- Can be function **parameters**
- Can be ***returned*** from functions
- Example:

```
int* someFunction(int* p);
```

  - This function declaration
    - Has a "*pointer to an int*" parameter
    - Returns a "*pointer to an int*" variable

Example 3

- Call-by-value pointer

Define Pointer Types

- Can "name" pointer types
- To eliminate the need for * in pointer declaration
- Declare:

```
Typedef int* IntPtr;
```

  - Defines a "new type" alias
  - The following becomes equivalent:

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

Types of Variables

- **Dynamic variables**
  - Created with the **new** operator
  - Created and destroyed while the program is running
- **Automatic variables**
  - **Local** variables
  - *Automatic* because *controlled* by the programmer
  - Created when the function in which they are declared is called and *automatically* destroyed when the function call ends
- **Global variables**
  - Variables declared outside any function or class definition
  - Generally, there is **NO** need for them

Dynamic Arrays

- Limitations of **static arrays**
  - Must specify size first → can be a waste of memory
  - May not know until program runs
- **Dynamic arrays**
  - Size ***not*** specified at programming time
  - Determined **while program is *running***
  - Use **`new`** operator

```
a = new double[10];
```

`delete` OPERATOR

- You need to **deallocate** dynamic memory
  - When ***no*** longer needed
  - To return memory to heap
- Example:

```
int *p;
p = new int(5);
//some processing…
delete p;
p = NULL;
```

- Deallocates dynamic memory "pointed to by pointer p" and re-sets the ponter to point to nothing.

Dangling Pointers

- The expression:

  ```
  delete p;
  ```

  - Destroys dynamic memory
  - But **p** still points there!
    - Called "**dangling pointer**"
  - If **p** is then "dereferenced" (***p**)
    - Unpredictable results!
- **Avoid** dangling pointers
  - Assign pointer to **NULL** after delete:

    ```
    delete p;
    p = NULL;
    ```

How to Delete Dynamic Arrays

- To delete a dynamic array

```
a = new double[10];
```

- You will need to add the squared brackets [ ]

```
delete [ ] a;
a = NULL;
```

- If you use **`delete a`**, ***without*** the squared brackets [ ], it will delete ***only*** the first element in the array, leaving the heap with occupied memory.

EXAMPLE 4

- Dynamic Arrays

Summing Up

```
int *p;
```

- **`p`** → *address* of the variable p is pointing to
- **`*p`** → *value* of the variable it is pointing to
- **`&p`** → *address* of itself

Questions?

(Pointers)