Pointers

C++ data types incorporate a set of values and operations
- e.g. int: -2B to 2B and +, -, ...
- categories
  - simple
    - integral (no decimal)
    - floating point
  - structured (collection of other data items)
    - arrays, structs, classes
  - pointers
    - something new!

pointers defined
- set of values: memory addresses
- so it's a variable whose content is a memory address
- does it have a type then?
  - e.g. int x; stores an integer at location named x
- but since it points to an address, it makes sense to identify what kind of value can be stored there
- so we specify what kind of data is stored at that memory location
- so no name associated with one (i.e. no pointer x;)

declaration
- data_type *identifier;
- e.g.
  - int *p;
  - char *ch;
- similarities:
  - int *p;
  - int* p;
  - int * p;

  - probably makes more sense to keep the asterisk with the variable name:
    - int *p;
  - why?
    - int* p, q; // p is a pointer to an integer; q is an integer
    - int *p, *q; // now both are pointers to integers

address of operator
```
int x;
int *p;

p = &x; // assigns the address of x to p
```
- so x and the value of p refer to the same memory location

dereferencing
- we use the asterisk to dereference a pointer
- it basically means to look up the value at that memory location
  - “follow the pointer and grab the value there”
- e.g.
```
int x = 25;
int *p = &x;

cout << *p << endl;   // displays the value stored in the memory location pointed to by p
cout << p << endl;    // displays the memory location of p
*p = 55;
```
- how does it dereference?
  - e.g.

```
int *p;
int num;        // step 1

num = 78;       // step 2
p = &num;       // step 3
*p = 24;        // step 4
```

	...		...		...		...
p: 1200					1800		1800
	...		...		...		...
num: 1800			78		78		24
	...		...		...		...

meanings

&p	address of p (in the e.g. above, that's 1200)
p	content of p (which is an address of some other variable)
*p	content of the memory location pointed to by p

another e.g.

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

	...
p: 1400	
	...
x: 1750	
	...

```
&p  =  1400
p   =  ?
*p  =  ?
&x  =  1750
x   =  ?

x = 50;
&p  =  1400
p   =  ?
*p  =  ?
&x  =  1750
x   =  50

p = &x;
&p  =  1400
p   =  1750
*p  =  50
&x  =  1750
x   =  50

*p = 38;
&p  =  1400
p   =  1750
*p  =  38
```

```
&x    =    1750
x     =    38
```

note

a declaration such as: int *p allocates memory only for p (NOT for *p)

consider:

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

then:

p is a pointer variable
the content of p points only to a memory location of type int
memory location x exists and is of type int

so the assignment:

```
p = &x;
```

is legal, and afterward *p is valid and meaningful

it dereferences the address in p and "returns" the contents of that memory location

initialization

recall: C++ does not automatically initialize variables

we can initialize variables ourselves however:

```
int x = 0;
int y[5] = { 0 };
```

we can also initialize pointers

```
int *p = NULL;
int *q = 0;              // these two statements mean the same thing
```

or

```
int *r;
r = NULL;
r = 0;                   // these two statements mean the same thing
```

we call this a null pointer

null pointers point to nothing!

dynamic variables

we've learned how to use pointers to manipulate data in existing memory spaces

i.e. we had to declare some other variable and set the pointer to point to it

but now we can do much more powerful things with pointers

we can allocate and deallocate memory during program execution using pointers

variables created at run time are called dynamic variables

operators (reserved words)

new

create a new dynamic variable

delete

destroy a dynamic variable

new

```
new dataType;              // allocate a variable
new dataType[intExp];      // allocate an array of variables
```

e.g.

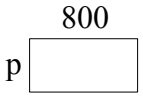
```
int *p;                    // p is a pointer of type int
char *name;                // name is a pointer of type char
string *str;               // str is a pointer of type string
```

```
p = new int;                // allocates memory of type int; stores this address in p
*p = 28;                    // stores 28 in the allocated memory

name = new char[5];         // allocates memory for an array of 5 chars and stores the base address of this array in name
strcpy(name, "John");       // stores "John" in name

str = new string;           // allocates memory of type string; stores this address in str
*str = "Hello World!";      // stores this string in the memory location pointed to by str
```

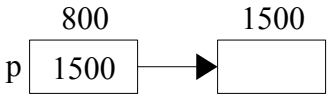
delete

e.g.

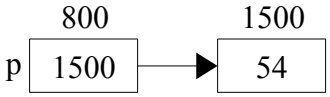
```
int *p;
```



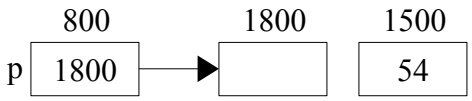

```
p = new int;
```



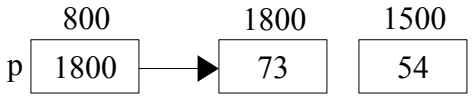

```
*p = 54;
```




```
p = new int;
```




```
*p = 73;
```



what happened to memory location 1500?

it's in limbo!

and C++ does not "collect garbage"

so we should probably free up this memory so that we can use it for other variables

delete

```
delete pointerVariable;         // deallocate a single dynamic variable
delete [] pointerVariable;      // deallocate a dynamically created array
```

e.g.

```
delete p;
delete [] name;
delete str;
```

we should probably set pointers to null after deleting any dynamic variables they point to

e.g.

```
int *p;

p = new int;
*p = 25;
```

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

now we won't have a “dangling” pointer which may cause unexpected errors
what would happen if we tried to dereference after a delete?
who knows...

operations on pointers

```
int *p, *q;

p = q;          // copies the value of q into p (memory address is copied)
if (p == q)     // evaluates to true if both point to the same memory location
    ...
if (p != q)     // the inverse
    ...
p++;            // DANGEROUS! this increments p by one int (4 bytes)
```

e.g.

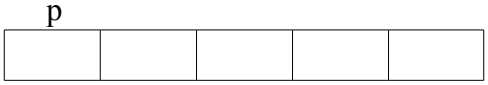
```
double *d;
char *c;

d++;            // increments d by one double (8 bytes)
c++;            // increments c by one char (1 byte)
c = c + 2;      // increments c by two chars (2 bytes)
```

dynamic arrays

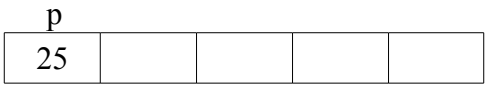
also known as anonymous arrays

e.g.

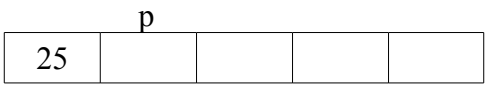
```
int *p;

p = new int[5];          // allocates 10 contiguous memory locations of type int
                         // stores the address of the first element in p
```



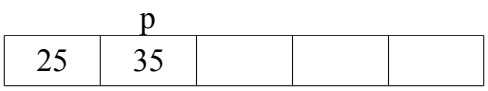

```
*p = 25;                 // stores 25 into the first memory location
```




```
p++;                     // p now points to the next element of the array
```

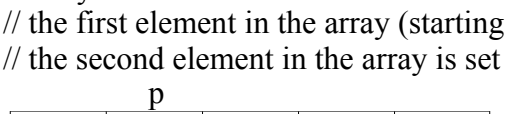



```
*p = 35;                 // stores 35 into this next memory location
```



we can also index as in normal arrays:

```
p[0] = 10;               // the first element in the array (starting at p) is now set to 10
p[1] = 20;               // the second element in the array is set to 20
```



BE CAREFUL!

pointer arithmetic is very dangerous
this is why many languages do not support pointers
or at least support them in an “unsafe” manner

recall:

```
int list[5];       // list is actually a pointer; it points to the first element in the array
```

list and &list[0] are the same thing
list is just a constant pointer (it cannot be changed)

so:

```
int list[5];
int *p;

p = list;                  // now p points to the first element in the array as well
cout << list[0];
cout << *p;
cout << *list;             // all three statements are the same!

int *q = &list[3];         // valid
```

e.g.

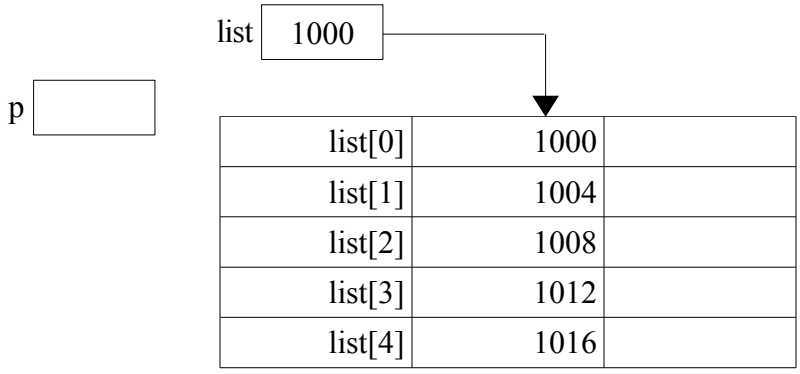
```
char color[] = “blue”;
char *color2 = “blue”;                  // both are the same except color2 is anonymous

cout << color;                  // --> blue
cout << color2;                 // --> blue
color2++;
cout << color2;                 // --> lue
color2--;
cout << *color2;                // --> b
```

e.g.

```
int *p;

p = new int[5];
for (int i=0; i<5; i++)
{
        p[i] = i * i;
        cout << p[i] << “ “;
}
cout << endl;
```

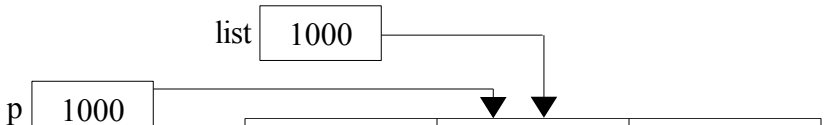
e.g.

```
int list[5];
int *p;
```



list[0]	1000	
list[1]	1004	
list[2]	1008	
list[3]	1012	
list[4]	1016	

```
list[0] = 25;
list[3] = 35;
p = list;
p[4] = -9;
p++;
*p = 33;
*p++;  // hey; dereferencing is done first!
```



list[0]	1000	25
list[1]	1004	34
list[2]	1008	
list[3]	1012	35
list[4]	1016	-9

```
cout << *(p++);        // output is?
```

functions and pointers

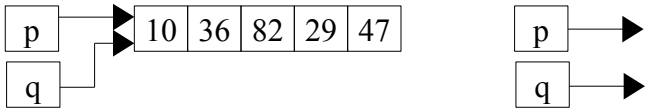
```
int* func(int*&, double*);
    the first parameter is a pointer to an int, passed by reference
    the second parameter is a pointer to a double, passed by value

int* func(int* &p, double *q)
{
    *p = 5;          // this is valid even after returning from this function
    *q = 6.22;       // this is only valid in this function

    int *r = p;      // r is a pointer to whatever p is pointing to

    return r;
}
```

shallow/deep copy and pointers

consider:

```
int *p, *q;
```

```
p = new int[5];
q = p;
delete [] p;
```



this is a shallow copy

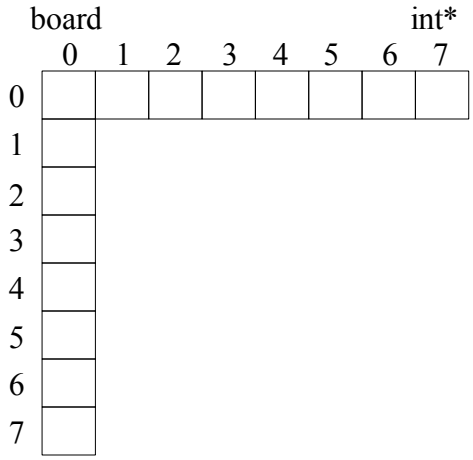
instead:

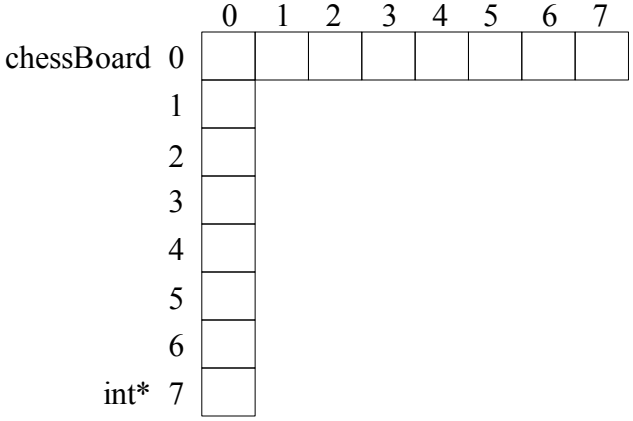
```
int *p, *q;

p = new int[5];
q = new int[5];

for (int i=0; i<5; i++)
        q[i] = p[i];

delete [] p;
```

this is a deep copy

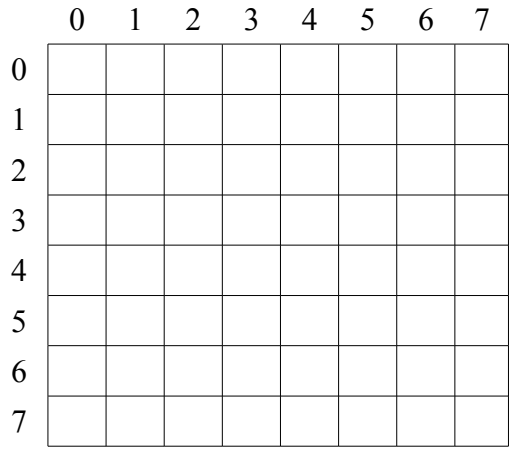
dynamic 2d arrays

```
int *board[8];                  // 2 ways of drawing this array

for (int col=0; col<8; col++)   // could use row here instead; depends on how we draw the array
        board[col] = new int[8];    // [8] can be specified at runtime
```





end result is the same:



so board is a 2d array of 8 rows and 8 columns
we can specify the number of rows at runtime, but not the number of columns
so this is truly not a dynamic 2d array

```
int **board;

board = new int*[8];                    // now int*[8] can be specified at runtime

for (int col=0; col<8; col++)
        board[col] = new int[8];        // and so can int[8] here
```

now we have a truly dynamic 2d array
so we could generate one dynamically as follows:

```
        int **board;    // wow; a pointer to a pointer!
        int rows, cols;

        cout << "How many rows? ";
        cin >> rows;
        cout << "How many columns? ";
        cin >> cols;

        board = new int*[rows];
        for (int row=0; row<rows; row++)
                board[row] = new int[cols];

        for (int row=0; row<rows; row++)
        {
                for (int col=0; col<cols; col++)
                        cout << board[row][col] << " ";
                cout << endl;
        }
```

HANDOUT pointer worksheet