9.1 Pointers

- Memory address of variable tells where variable is, not its name
- Call-by-ref a form of pointer

To declare:
- Cannot use integer even though address is num
- Must say it is a pointer of X type
  - `int *p`
  - Place asterisk in front of name

Accessing:
- `*p` - value of addr pointed to by p
- `p` - addr pointed to by p

Assignment:
- `p = &v` - get addr of v, store in p
- `*p = 42` - store 42 in addr pointed to
- `*` is dereferencing operator
- `&` is "address-of" (referencing) operator
- `p1 = p2` - works when both are pointers
- `*p1 = *p2` - assign values, not addr

Example:

\[
\begin{align*}
\text{Before} & & \text{After} \\
p1 & \rightarrow \square & p1 & \rightarrow \square \\
p2 & \rightarrow \square & p2 & \rightarrow \square \\
\end{align*}
\]

\[
\begin{align*}
\text{Before} & & \text{After} \\
p1 & \rightarrow 84 & p1 & \rightarrow 99 \\
p2 & \rightarrow 99 & p2 & \rightarrow 99 \\
\end{align*}
\]
Creating new memory chunks
allocate memory from computer
"anonymous" because has no name, just pointer
int *pl;
pl = new int;
also called dynamic variables
new is C++ method of allocating
C uses function calls
if not enough mem, program will terminate
Example
int *pl, *p2;
pl = new int;
*p1 = 42;
p2 = pl;
*p2 = 53;
pl = new int;
*p1 = 88;
Can also call constructors
pl = new int(17);
sets value of mem chunk to 17
Memory Management
"freestore" - mem that can be allocated
can be exhausted by too many news
free dynamic vars no longer used w/delete
delete p
also important to delete before assigning
pointer a new addr, mem leak
dangling pointer - pointer has no addr
after delete is called
do not use *p
Static vs Dynamic Variables

dynamic - created by new allocated/freed while program is running
ordinary vars are not static however
static is special keyword
we will return to its concept w/ classes

call ordinary vars "automatic vars"
they are a subset of dynamic vars
that are created/deleted by
the restrictions of their scope

Using typedef
typedef is a sort of alias
can be used for any datatype
Example:

    typedef int* Int_Ptr;
    Int_Ptr p1, p2;

helps avoid accidentally forgetting *
    int *p1, p2; // p2 normal int
    int* p1, p2; // p2 normal int
    int *p1, *p2; // correct
    Int_Ptr p1, p2; // correct

Syntax:
    typedef <known-type> <alias>; j

9.2 Dynamic Arrays
size determined while program is running
instead of when written like Ch 7
an array var is actually a pointer
[size] tells how much mem to alloc
[size] tells how much mem to alloc
array var points to 1st element
can assign array vars to pointers

Example:

```c
int a[10];
int *p;
p = a;  // but cannot reassign a
p[1] & a[1] both access 2nd element
```

Can use [] w/ pointers if pointer points to array
pointer simply becomes base addr

Ordinary arrays cannot be assigned like ptrs
a = p;  // illegal

Creating & Using Dynamic Arrays
in Ch 7, dealt w/ issue of unknown
size using partially filled arrays
wasteful of memory

Cannot grow beyond maximum
dynamic arrays are "just right" size
give arg to new to say "want an array"

```c
int *p;
p = new int[10];
```

Can use var for size too

```c
int size = 5;
int *p;
p = new int[5];
```

must also give new arg to delete
lets delete know its an array
delete [] p;  

w/o [], delete only frees space

used by first element
do not call new again w/o first delete
    can create mem leak
after delete, can call new again

Pointer Arithmetic
operates on addresses, not numbers

Example:

double *p;
p = new double[10];
cout << *(d + 1) << " " << d[i];
this would output same element

d+1 evals to "add one offset"
d+1 = d + 1 * size-of_double
d+i = d + i * size-of_double

can only add or subtract

Multidimensional Dynamic Arrays
multidimensional are arrays of arrays

eg pointers to pointers

typedef int* IntPtr;
IntPtr *m = new IntPtr[3];
for (int i=0; i<3; i++)
    m[i] = new int[4];
for (int i=0; i<3; i++)
    delete [] m[i]; // inner array
delete [] m; // outer array