Variables, Memory and Pointers

- A variable is a named piece of memory
  - The name stands in for the *memory address*

```
int num;
```

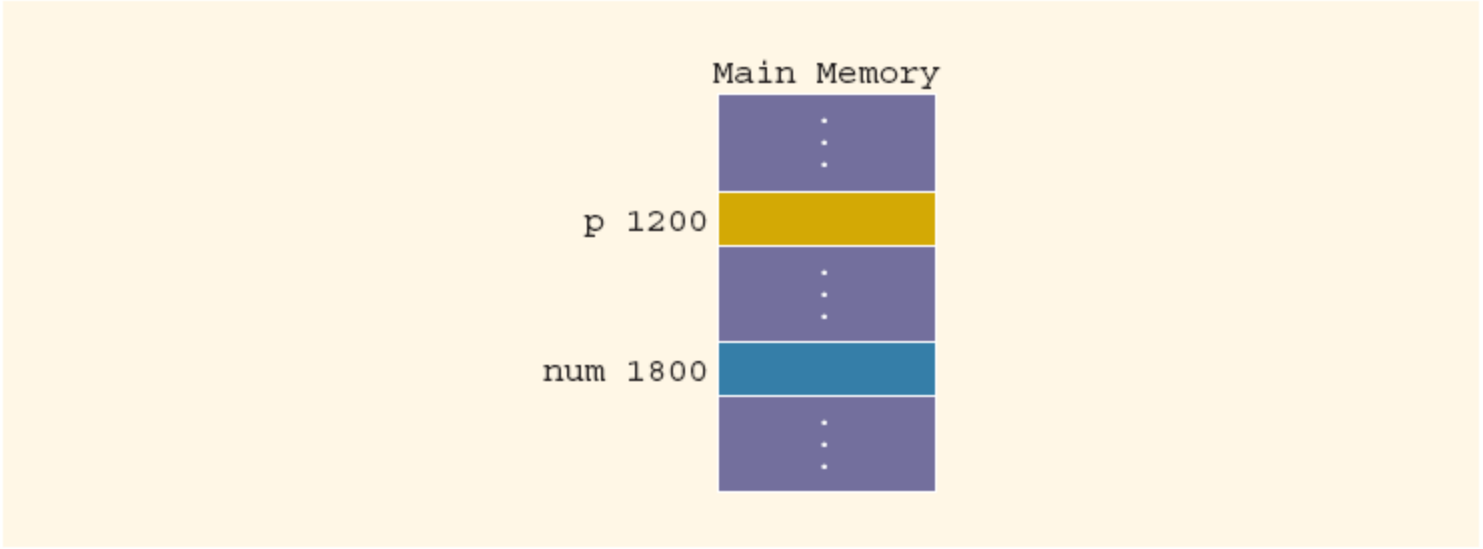


FIGURE 13-1 Main memory, p, and num

Variables, Memory and Pointers

- When a value is assigned to a variable, it is stored at that address in memory

```
num = 78;
```

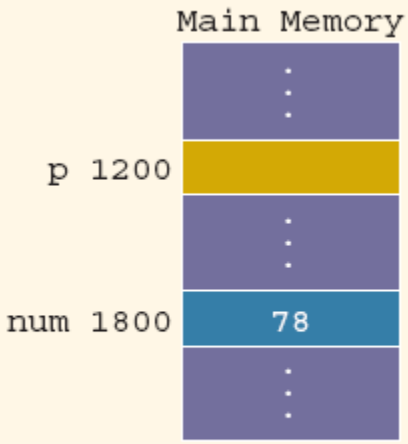


FIGURE 13-2 num after the statement num = 78; executes

Variables, Memory and Pointers

- A *pointer* is a variable that holds the address of another variable
  - It is declared in terms of the type of variable it points at:

```
int *p;
```

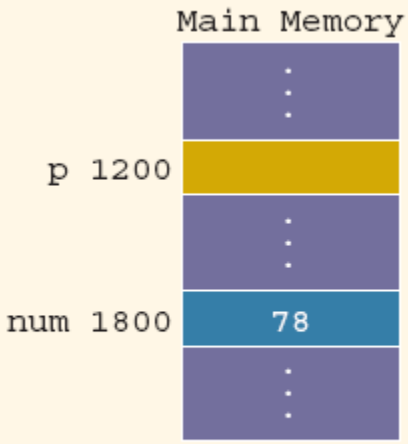


FIGURE 13-2 num after the statement num = 78; executes

Address-of operator

- The operator `&` returns the address of a variable
  - It can then be assigned to a pointer

```
p = &num;
```

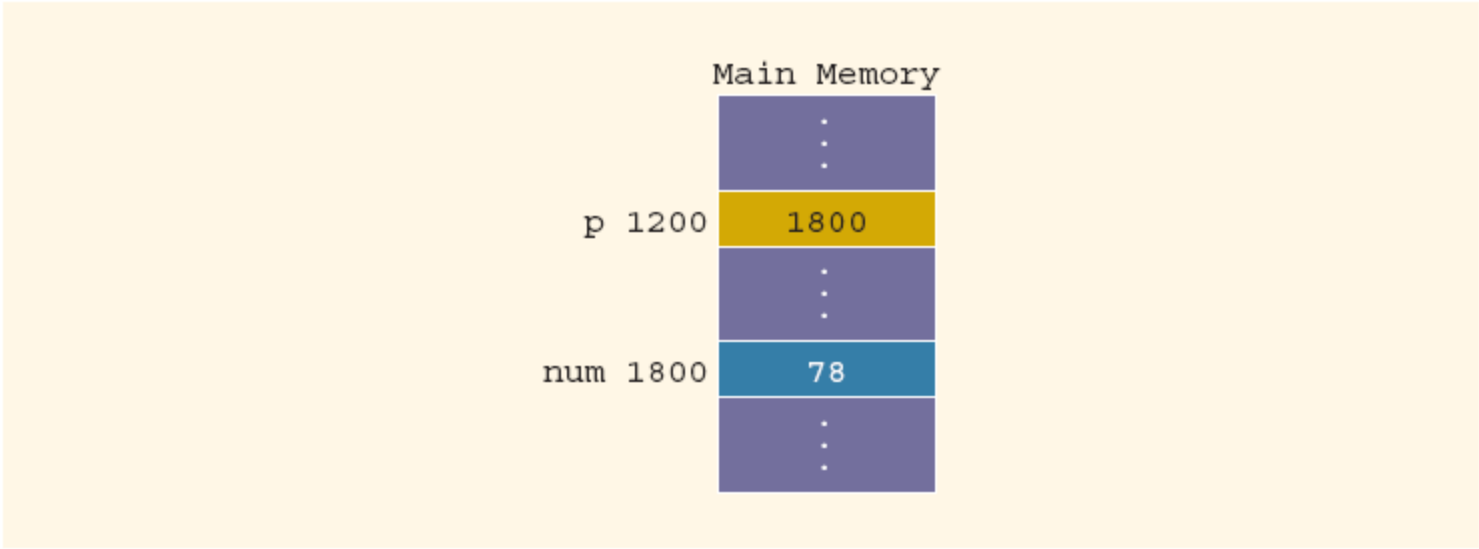


FIGURE 13-3 p after the statement p = # executes

Dereference operator

- The operator * takes an address (a pointer) and returns the location in memory being pointed to
  - Can only be applied to a pointer

```
*p = 24;
```

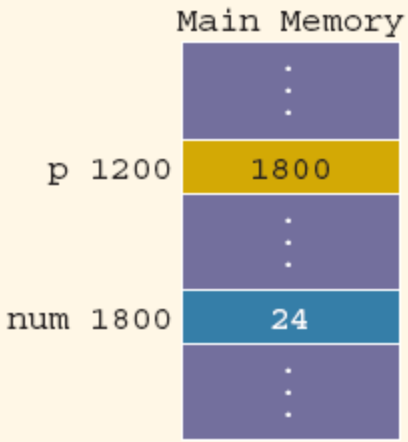
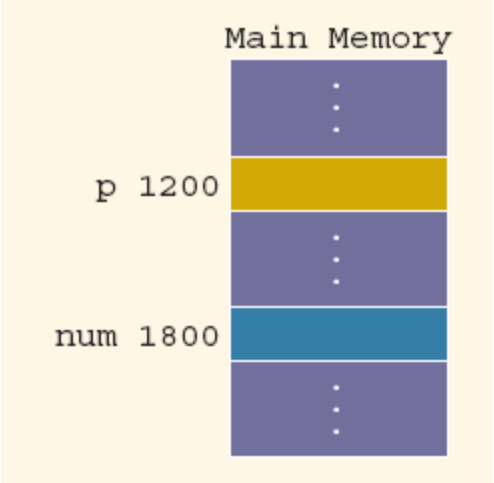


FIGURE 13-4 *p and num after the statement *p = 24; executes

Example Values

- Assuming the memory layout provided, after this code executes:

```
int num;
int *p;
num = 50;
p = &num;
```



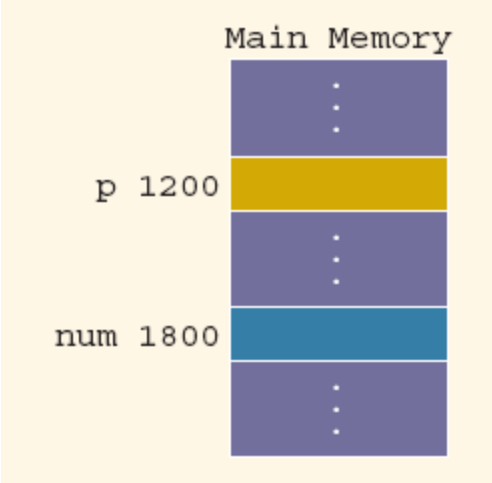
- What are the values of these expressions?

```
&num = ?
num = ?
&p = ?
p = ?
*p = ?
```

Example Values

- Assuming the memory layout provided, after this code executes:

```
int num;
int *p;
num = 50;
p = &num;
```



- What are the values of these expressions?

```
&num = 1800
num = 50
&p = 1200
p = 1800
*p = 50
```

Assigning Pointers

- Pointers can be assigned to pointers of the same type

```
int x, *p, *q;
x = 50;
p = &x;
q = p;
```

- The value of *q is?

Assigning Pointers

- Pointers can be assigned to pointers of the same type

```
int x, *p, *q;
x = 50;
p = &x;
q = p;
```

- The value of *q is 50

Comparing Pointers

- Be careful of the difference between comparing two pointers and comparing their values:

```
int x = 50, y = 50, *p, *q;
p = &x;
q = &y;
```

- `*q == *p` evaluates to?
- `q == p` evaluates to?

Comparing Pointers

- Be careful of the difference between comparing two pointers and comparing their values:

```
int x = 50, y = 50, *p, *q;
p = &x;
q = &y;
```

- `*q == *p` evaluates to `true`
- `q == p` evaluates to `false`

Pointers and Classes

- A pointer to an object is no different than a pointer to any other type of variable

```
class Album
{
public:
  Album();
  string title;
  string artist;
};

Album myFavorite;
Album *album_ptr;
album_ptr = &myFavorite;
```

Pointers and Classes

- The public members of that object are accessed by a combination of dereference (*) and membership (.):

```
(*album_ptr).artist = "Zamfir";
```

- Since this is done a lot, there is an easier syntactic shortcut:

```
album_ptr->artist = "Zamfir";
```

The Null Pointer

- In addition to variable addresses and other pointers, a pointer can be assigned to the *null pointer*
  - Either the number `0` or the constant `NULL`
  - Used to indicate an invalid pointer (pointing to nothing)
  - Dereferencing a null pointer causes a hard error (segmentation fault)

```
Album *album_ptr = 0;
album_ptr = NULL;

album_ptr->artist = "Zamfir";     // error
```

Pointers and functions

- Stack variables are tied to a particular function
  - Allocated in the *stack frame* for that function
- Heap memory is not!
  - Pointers to the heap are valid no matter what function you are in
  - Pointers are passed by value (copied), but still point to the same heap memory
  - Changes made using a pointer persist:

```
void change_title( Album * p )
{
    p->title = "A better title";  // changes heap memory
}
```

Dynamic Memory Allocation

- Local variables and parameters are allocated on the *stack*
- Stack variables are:
  - Allocated by the compiler before the program is run
  - Pre-determined size, even arrays

```
// stack object
Album myAlbum;

// stack variable
Album *ap;

// stack array
Album collection[100];
```

Dynamic Memory Allocation

- Pointers give us a mechanism for dynamically allocating memory on demand
  - Allocated as needed at run-time (during execution)
  - Size determined at time of allocation
  - De-allocated when no longer useful

Operator `new`

- A pointer variable is allocated on the stack:

```
Album *myAlbum = 0;
```

  - But there is no space for the actual data

```
(*myAlbum).artist = "Zamfir"; // error!
```

- The `new` operator allocates memory for a specified type of variable and returns the address

```
myAlbum = new Album;
// now write to newly allocated memory
(*myAlbum).artist = "Zamfir";
```

  - This memory is **not named**
  - If the pointer is moved, we have no way to get back to that piece of memory

Operator `new`

- `new` can also be used to dynamically allocate memory space for arrays (can be variable size!)

```
double *scores;
int x;
cout << "How many scores?" << endl;
cin >> x;
scores = new double[x];
```

- Could be an array of objects, of course

```
Album *collection;
collection = new Album[36];
```

- A pointer can point at a variable OR an array!
  - And can be used with [] if it is pointing at an array

```
scores[4] = 6.7;
```

Operator `new` and constructors

- Dynamically allocated objects are constructed just like stack allocated objects
  - Allocate memory (from stack vs. heap)
  - Run appropriate constructors
- Syntax:

```
Album album( "Zamfir", "Greatest Hits" );
```

  - Vs.

```
Album *ap;
ap = new Album( "Zamfir", "Greatest Hits" );
```

- Arrays of objects are always default constructed!
  - Same for heap allocation as for stack allocation
  - Can't pass in parameters either way

Operator `new`

- How about a dynamic array of pointers?

```
Album **collection;
collection = new Album*[36];
```

- Why would you do such a thing?!
  - The pointers don't take up unnecessary space
  - You can set them to NULL to indicate "no album here"

```
for( int i=0; i<counter; i++ ) {
  collection[i] = NULL;
}
```

  - Imagine a better `Add` method:

```
collection[counter] = new Album( title );
counter++;
```

Operator `delete`

- Dynamic memory allocation runs the risk of running out of memory during program execution
  - If `new` cannot allocate the requested piece of memory, it fails and terminates the program
- To avoid this, programs need to de-allocate dynamic memory when they are done with it
  - This is done with the operator `delete`, called on a pointer to a dynamically allocated piece of memory

```
int *p, *q;
q = new int;
p = new int[x];
delete q;
delete [] p;
```

Memory Leaks

- If a pointer to dynamically allocated memory is moved, that memory location cannot be found again

```
int *p;
p = new int[1024];
p = new int[518];
p = new int[20435];
p = 0;
```

  - This means it cannot be `delete`d, and is lost to the program
  - This is referred to as a *memory leak*
  - If a leak occurs in a repeating part of the program, then no matter how small it is the program will eventually run out of memory
    - This matters a lot in server programs that run for months or years