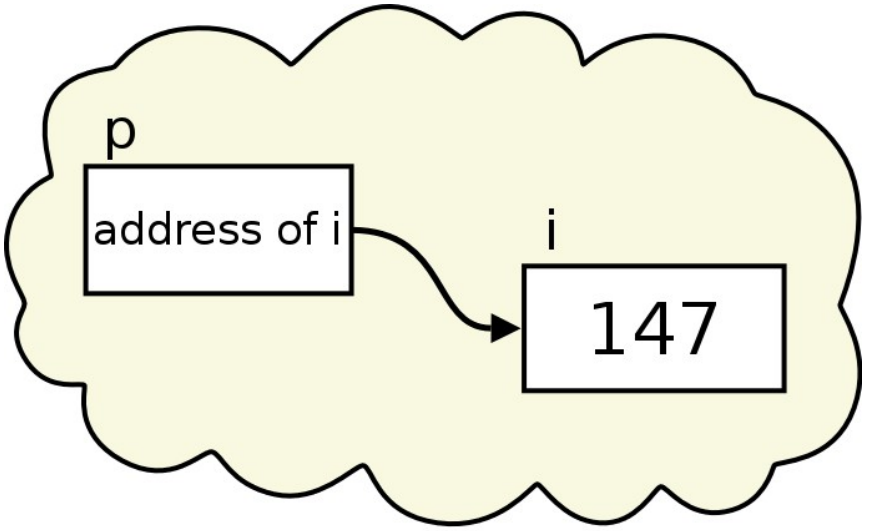
CS2141 – Software Development using C/C++

Pointers

What is a pointer?

- A *pointer* is simply a variable that stores the memory address of some other value
- All pointers on a given machine are the same size since all memory addresses are the same size

```
int i = 147;
int * p = &i;
```



Pointers in Java

- Does Java have pointers? Recall the Java program from before:

```
public class TestClass {
   public int value;

   public static void main( String[] args ) {
      TestClass obj1 = new TestClass( );
      TestClass obj2;
      obj1.value = 12;
      obj2 = obj1;
      obj1.value = 18;
      System.out.println( “obj1 value “ + obj1.value );
      System.out.println( “obj2 value “ + obj2.value );
   }
}
```

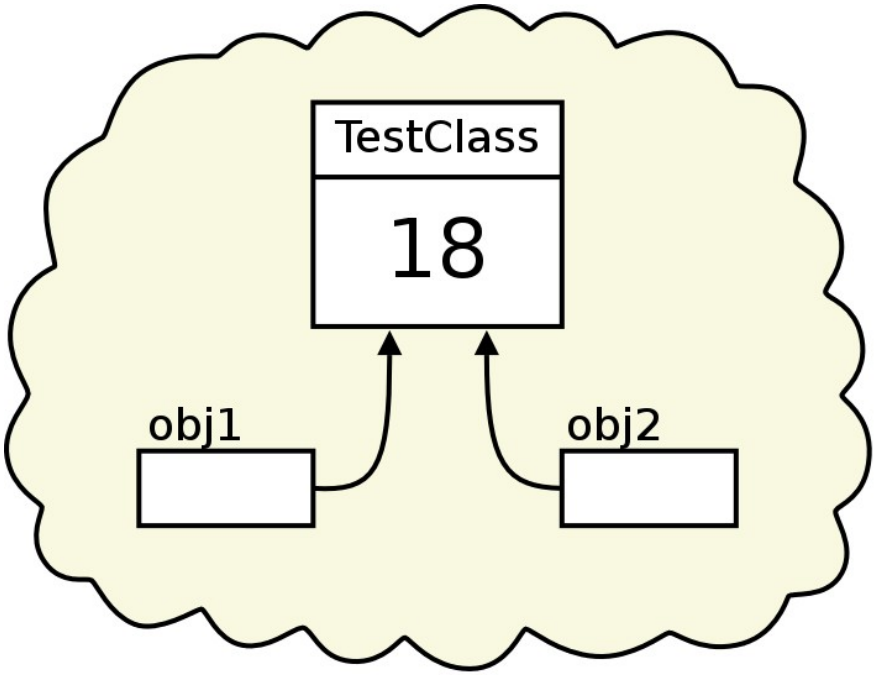
Pointers in Java cont.

- And the C++ program from before:

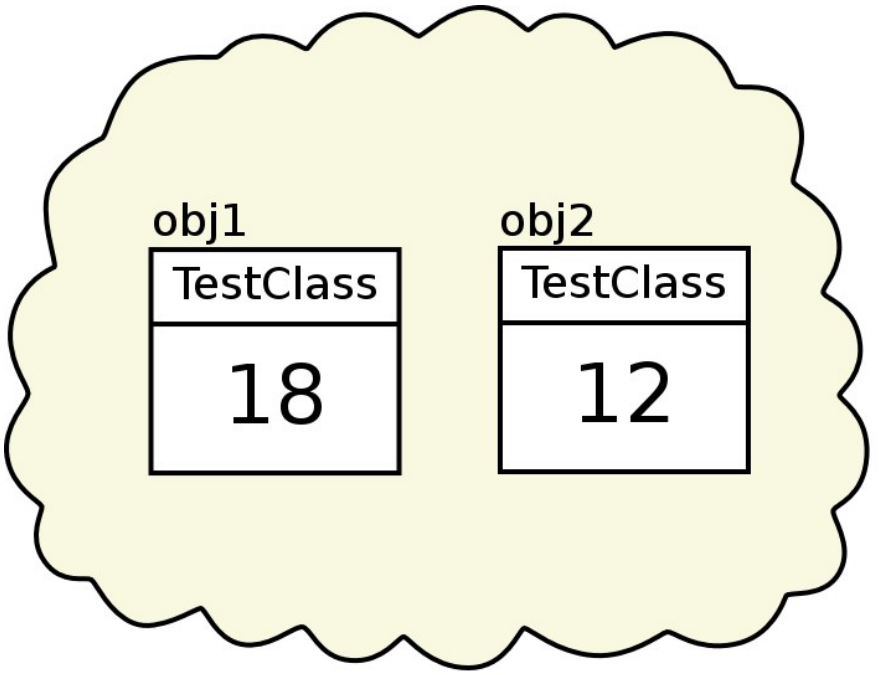
```
class TestClass {
   public:
   int value;
};

int main( ) {
   TestClass obj1;
   TestClass obj2;
   obj1.value = 12;
   obj2 = obj1;
   obj1.value = 18;
   cout << “obj1 value “ << obj1.value << endl;
   cout << “obj2 value “ << obj2.value << endl;
}
```

Pointers in Java cont.

- Remember the results:



Java (object references)



C++ (object values)

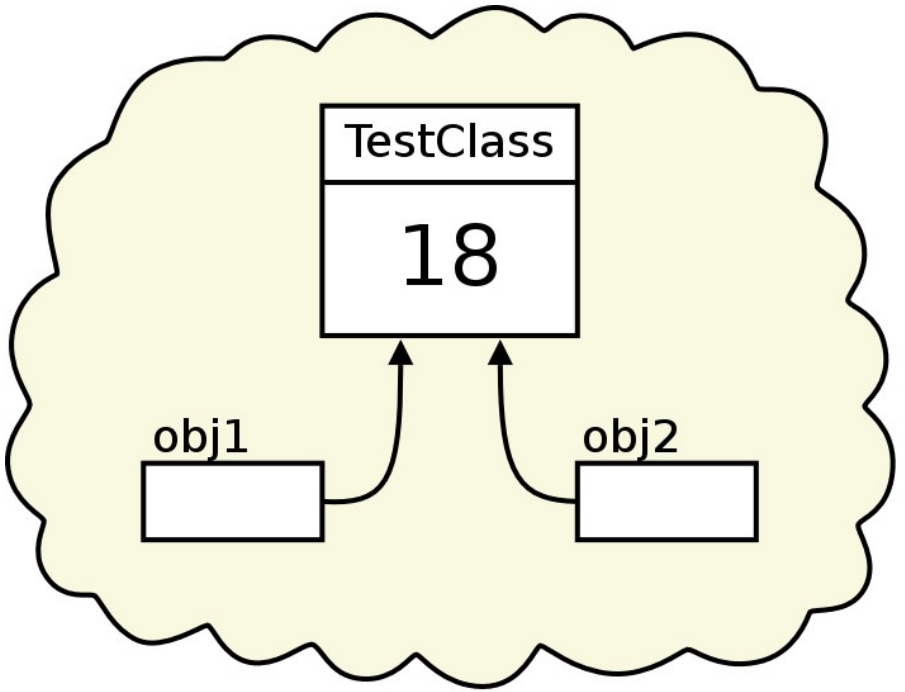
Pointers in Java cont.

- Now consider this C++ program:

```
class TestClass {
   public:
   int value;
};

int main( ) {
   TestClass * obj1 = new TestClass( );
   TestClass * obj2;
   obj1->value = 12;
   obj2 = obj1;
   obj1->value = 18;
   cout << “obj1 value “ << obj1->value << endl;
   cout << “obj2 value “ << obj2->value << endl;
}
```

Pointers in Java cont.

- When run, it works like the Java version, with **`obj1`** and **`obj2`** pointing at the same object:



So does Java have pointers?

Declaring a Pointer

- A pointer is declared using the data type of the value it will point at and an asterisk:

```
float * fp;  // pointer to a float
```

- A *null pointer* is a pointer value that does not refer to any memory location
  - A pointer can be made null by assigning zero to it:

```
char * s = 0;
```

  - Since a pointer is either null (zero) or non-null (not zero), it can be used as a boolean like an integer

Assigning Values to Pointers

- Three ways to assign pointers:
  - Using new, which returns a pointer:

```
TestClass * obj1 = new TestClass( );
```

  - Copying an existing pointer:

```
TestClass * obj2 = obj1;
```

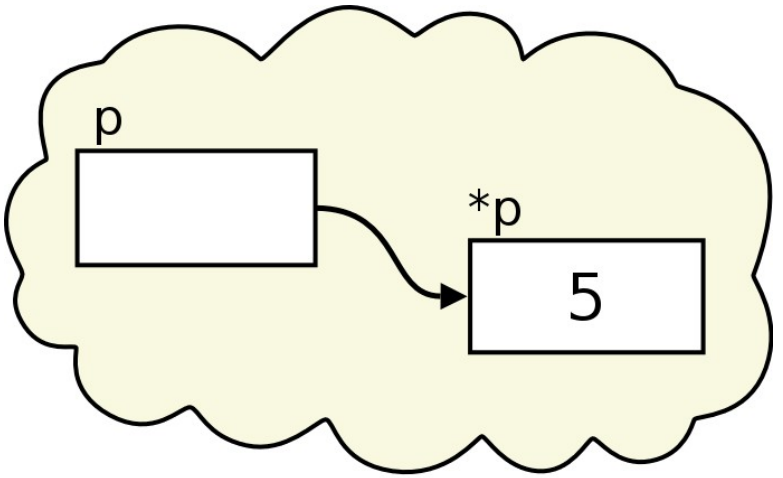
  - The *address-of operator* (**&**) is used to get the memory address of an existing value:

```
TestClass obj3;
obj2 = &obj3;
```

Dereferencing a Pointer

- A pointer can be *dereferenced* to access the value it points at. There are several ways to do this:
  - The * operator – if a variable **p** is holding the address of a value, then ***p** is the value pointed at by **p**

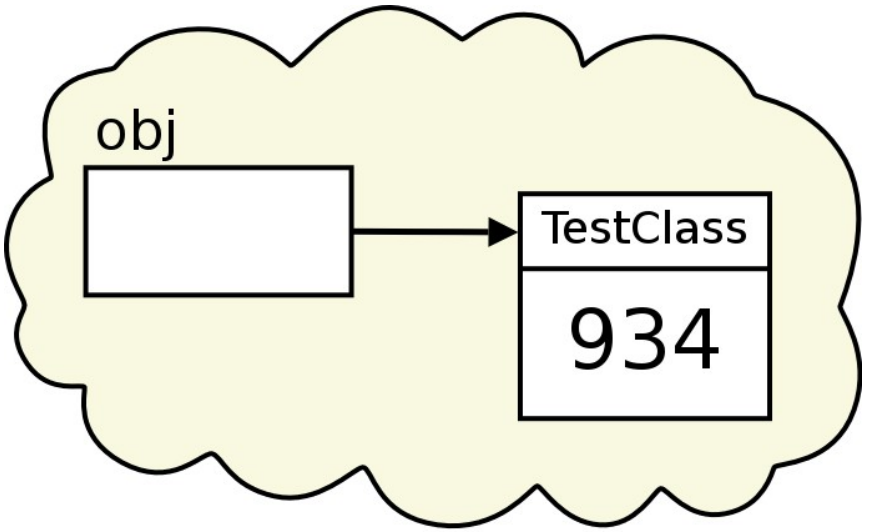
```
// Reserve space for a new
// integer and have p point
// at that space
int * p = new int;

// Set the value of the
// integer p points at to 5
*p = 5;
```



Dereferencing a Pointer cont.

- A pointer to a class can combine dereferencing and member field access using the pointer operator **->**.

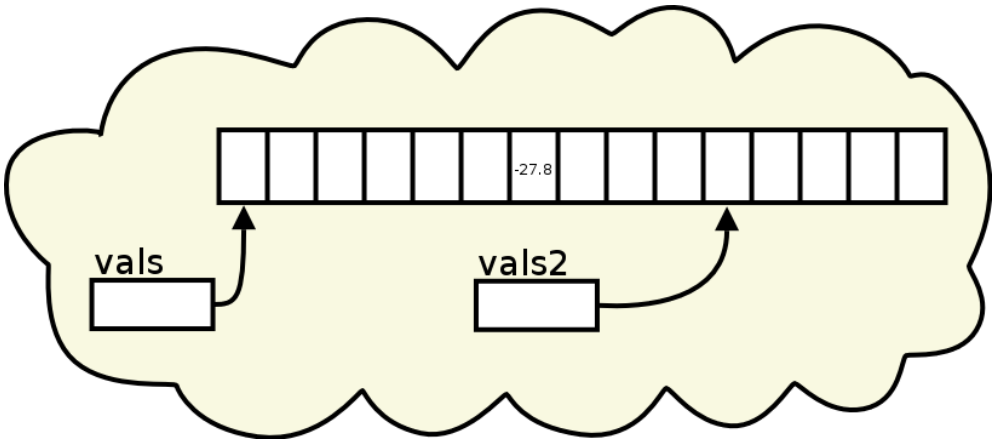
```
// Declare a pointer to
// a TestClass object
TestClass * obj;

// Allocate a TestClass
// object and set obj
// to point at it
obj = new TestClass( );

// Set the value field
// of the TestClass object
// obj points at to 934
obj->value = 934;
```



Dereferencing a Pointer cont.

- A pointer to an array can be subscripted to access array elements (more on pointers and arrays later):

```
float * vals;
vals = new float[15];
vals[6] = -27.8;
```



- An integer can be added to or subtracted from a pointer to yield a new pointer:

```
float * vals2 = vals + 10;
```

Pointer Operations

- Pointers to primitive data types should only be used in two operations; comparing for equality (or inequality) and dereferencing:

```
int * p = new int;
int * q = new int;
if( p == q )  // Decide if p and q point
   *p = 5;    // to the same location
else          // if( p != q )
   *q = 6;
```

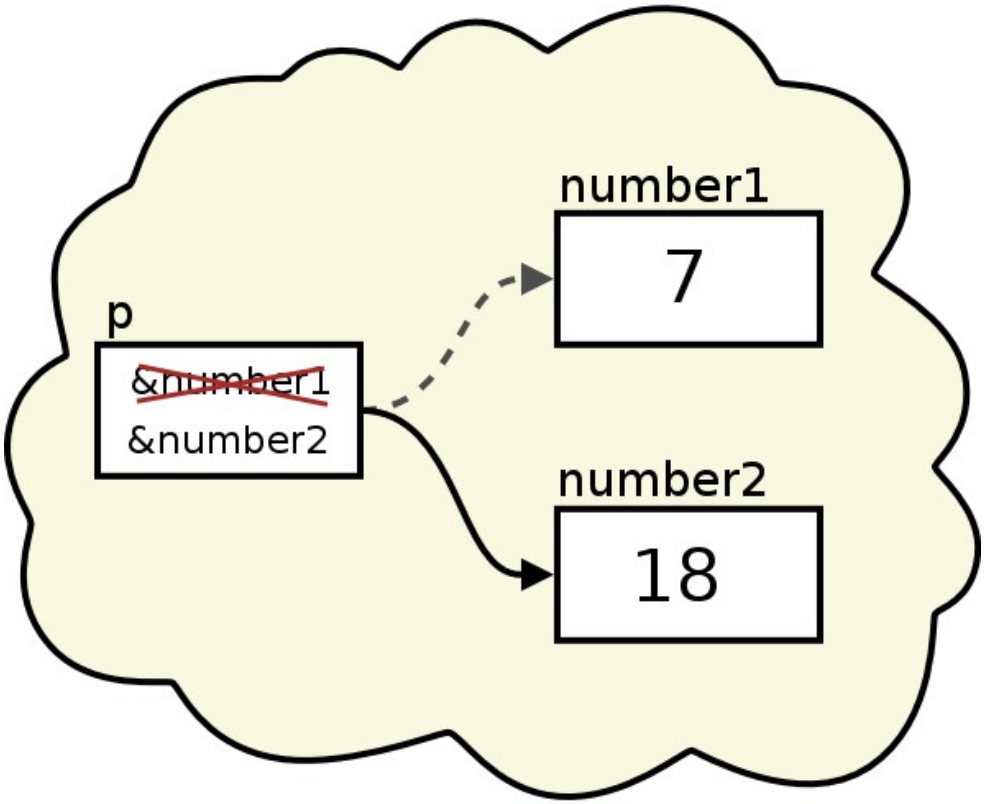
- Other operations are possible, but they don't make much sense:

```
if( p < q ) ...      // What??
```

Reassigning Pointers

- The location a pointer points at can be changed with another assignment:

```
int number1 = 7;
int number2 = 18;
int * p;
p = &number1;

...

p = &number2;
```



Using Pointers

- Dereferenced values can be used in any operation (including math):

```
*p = *p + number1;
```

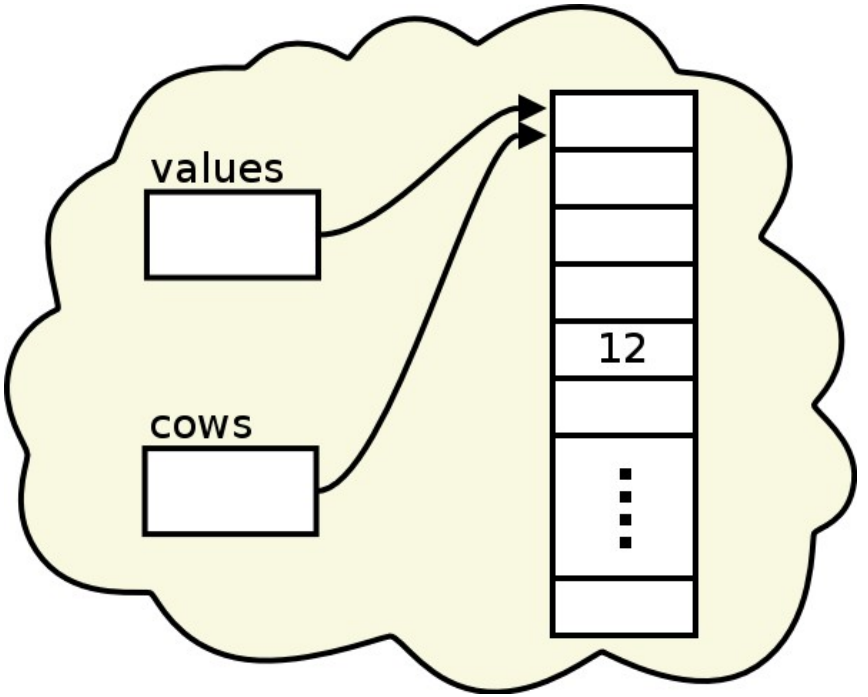
- Modifying a pointer is **not** the same as modifying the value it points at:

```
p = &number1; // Modifying the pointer.
*p = 42;      // Modifying the value
              // p points at.
```

Pointers and Arrays

- Pointers and arrays are very closely related
  - Any pointer can be subscripted
  - An array variable can be assumed to be a pointer

```
int values[100];
int * cows = values;

// These are the same:
cows[4] = 12;
values[4] = 12;
*(cows + 4) = 12;
*(values + 4) = 12;
```



Pointers and Arrays cont.

- Subscripts are never checked for range.
  - The following subscripts are all legal (the compiler will not complain), but incorrect:

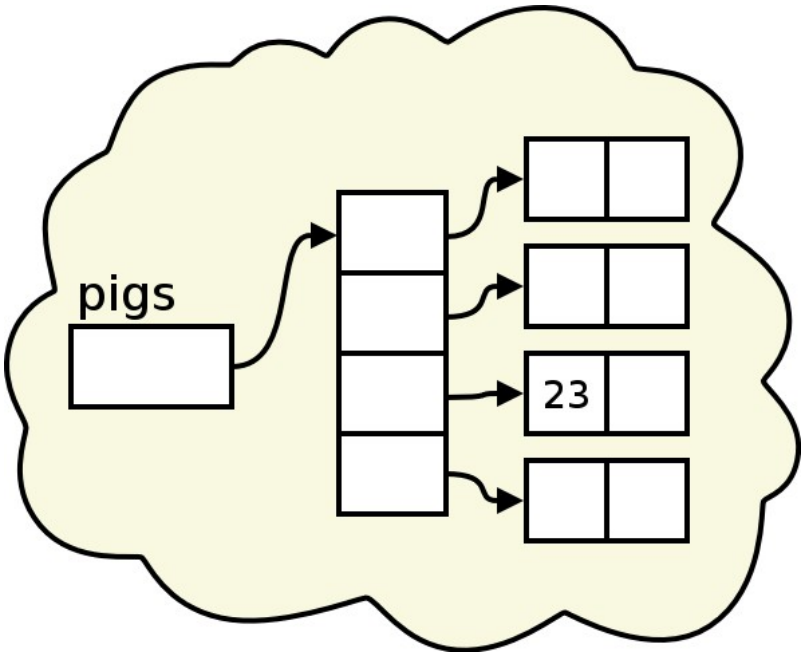
```
cows[250] = 83;
values[-27] = 42;
TestClass * obj1 = new TestClass( );
obj1[5].value = 8;
```

  - Rarely is there a need for an out of range subscript.
  - If it happens, it is usually a programming error.
  - The same thing can happen with pointer arithmetic.

Pointer to Pointers

- Multiple dereferences of the same variable are possible and sometimes convenient.
- An example is a multi-dimensional array:

```
int i;

int ** pigs = new int*[4];
for( i = 0; i < 4; i++ )
   pigs[i] = new int[2];

pigs[2][0] = 23;
```



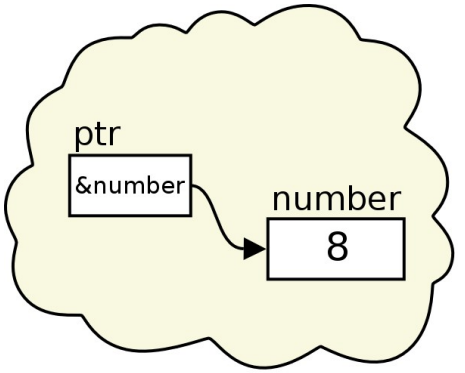
Global Pointers to Local Variables

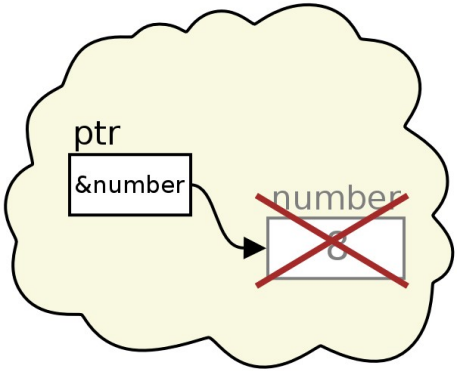
- Global pointers to local variables are a bad idea:

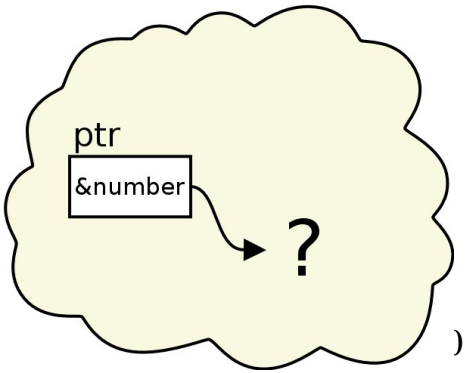
```
int * ptr;     // A global pointer

void set( ) {
    int number;
    number = 8;
    ptr = &number;
}

void use( ) {
    double value;
    value = 30;          // Assign a value.
    value += *ptr;       // Use the value
}                        // ptr points at.
```

Global Pointers to Local Variables cont.

- The code will probably fail:
  - When **`set`** is run, **`ptr`** is set to point at the local variable **`number`**.
  - Since it is a local variable, **`number`** is destroyed when **`set`** returns, but **`ptr`** still points at the memory location where **`number`** was.
  - **`use`** then uses the value **`ptr`** points at, but that value will very likely not be the one the programmer expected.







Pointers and `const`

- A pointer to a constant:

```
int number = 15;
const int * ptr1 = &number;
*ptr1 = 25;       // not allowed
```

- A constant pointer:

```
int * const ptr2 = &number;
*ptr2 = 32;       // allowed
ptr2 = ptr1;      // not allowed
```

- A constant pointer to a constant:

```
const int * const ptr3 = &number;
```

`void` pointers

- A `void *` pointer can point to anything
  - Can be used much like the `Object` type in Java
  - Any pointer can be converted into a `void *` pointer
  - Converting `void *` pointer back requires a cast:

```
double real;
double * rptr = &real;

void * gptr = rptr;

double * rptr2;
rptr2 = (double *)gptr;  // Converting back
```

Function Pointers

- A pointer can also point to a function:

```
int next_n( int n ) {
return n + 1;
}

// Declaring a function pointer fun_ptr
int (*fun_ptr)( int );

// Assigning it to point to next n
fun_ptr = &next_n;

// Or
fun_ptr = next_n;
```

Function Pointers cont.

- Using fun_ptr:

```
int x;

// Using fun_ptr
x = (*fun_ptr)( x );

// Or
x = fun_ptr( x );
```

- Do not use pointers to member functions.
  - The syntax is extremely obscure.
  - Such pointers are very rarely needed.