# Creating associations/links to other objects

```
#include <iostream>
#include <string>
using namespace std;

class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", married to: ";
        os << rhs.spouse.name;
        return os;
    }
public:
    Person(const string& name) : name(name) {}
    bool marries(Person& rhs) {
        spouse = rhs;
        return true;
    }
private:
    string name;
    Person& spouse;
};

int main() {
    Person john("John");
    Person mary("Mary");
    john.marries(mary);
    cout << john << '\n';
    return 0;
}
```

**Won't compile** - recall that references:

- Cannot be changed, i.e. always referring to same object (except in ranged for loops)
  - Thus they must be initialized at same instance of declaration

**What do we use then?**

Containment (i.e. classes containing objects)?

# Creating associations/links to other objects

```
#include <iostream>
#include <string>
using namespace std;

class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs) {
        os << "Name: " << rhs.name << ", married to: ";
        os << rhs.spouse.name;
        return os;
    }
public:
    Person(const string& name) : name(name) {}
    bool marries(Person& rhs) {
        spouse = rhs;
        return true;
    }
private:
    string name;
    Person& spouse;
};

int main() {
    Person john("John");
    Person mary("Mary");
    john.marries(mary);
    cout << john << '\n';
    return 0;
}
```

**Won't compile** - recall that references:

- Cannot be changed, i.e. always referring to same object (except in ranged for loops)
  - Thus they must be initialized at same instance of declaration

**What do we use then?**

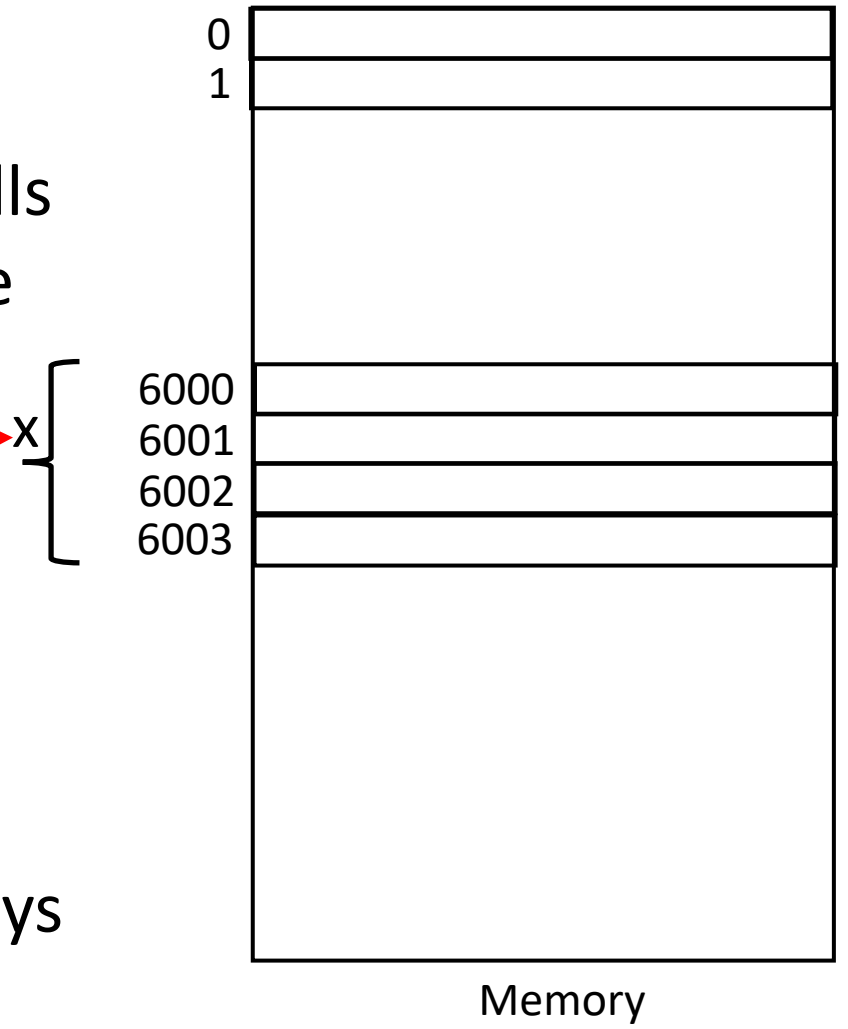
Containment (i.e. classes containing objects)?

- Won't work, can't have an object inside itself

# Pointers

- As stated in previous lectures, declaring a variable allocates a number of memory cells (or bytes), and assigning them a name (the name of the variable).
- Memory is organized into bytes (or cells), where each has a unique address.
- When a variable is allocated a certain number of bytes in memory, they are always contiguous, for example:

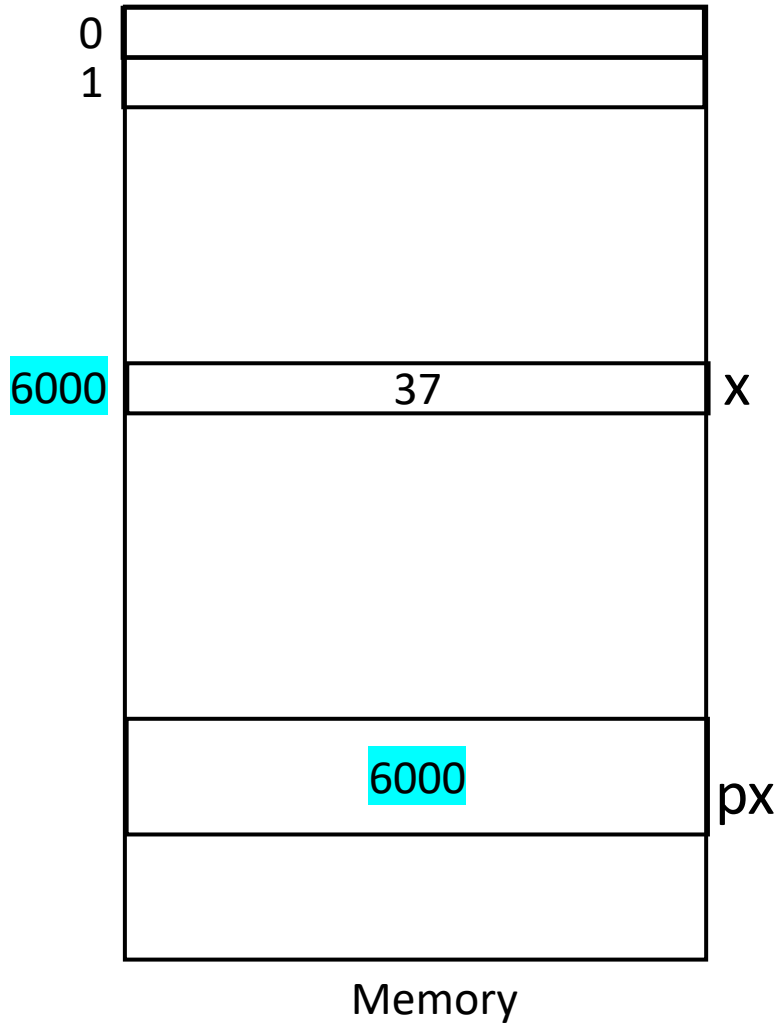
```
int x;
```



# Pointers (cont.)

- Every time your program is loaded, it may occupy a different area of memory, and hence the address allocated to the same variable may be different every time you run your program .

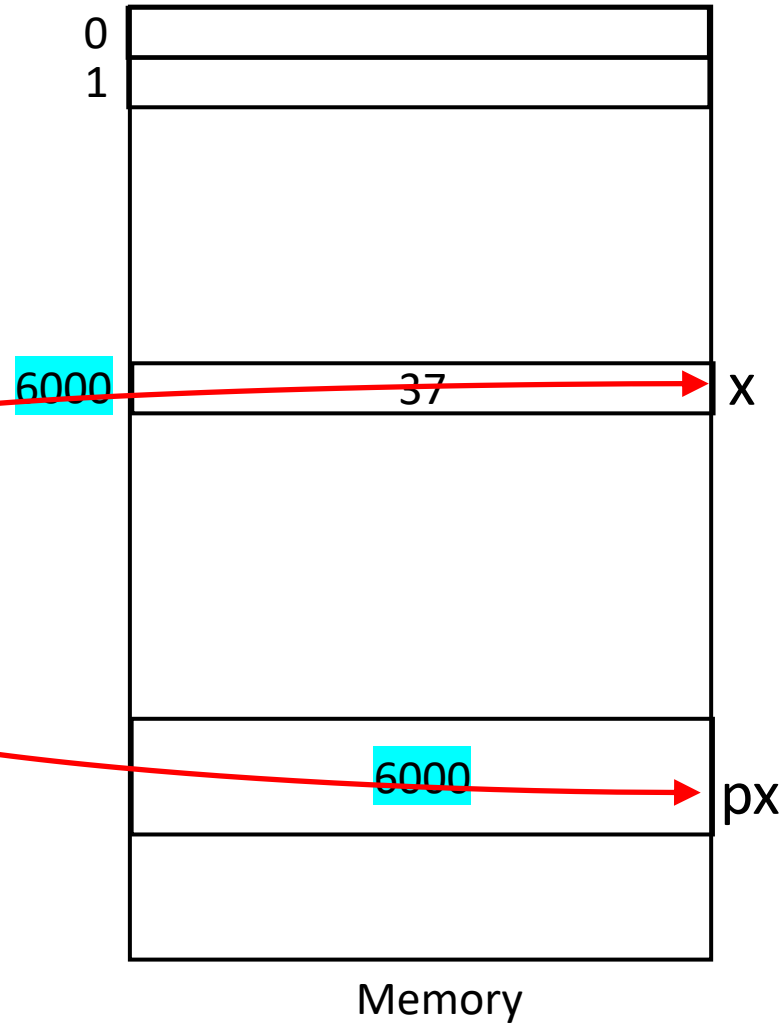
- In many cases, your program may need to know the memory address of your variable and may also need to access a variable using its address instead of its name.



# Pointers – The address of operator &

- You can access the address of a variable using the & operator, e.g.

```
char x = 37;
char *px;
px = &x
```




In the above example:

- & is the address-of operator
- char* is the declaration of a pointer

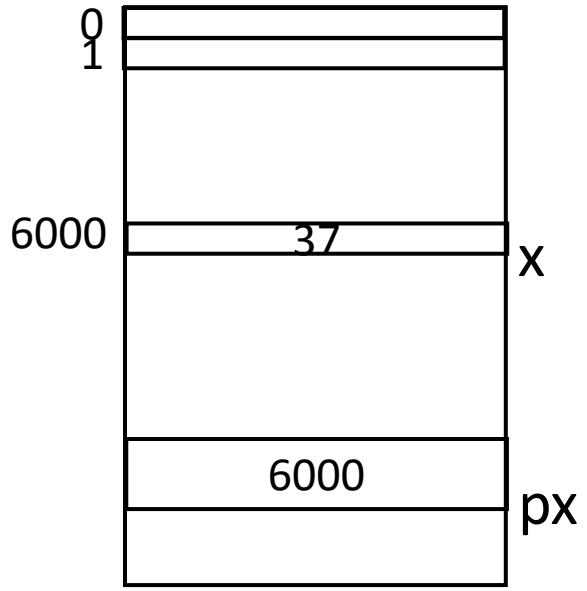
# Pointers – The dereference operator *

The dereference operator * allows reads or writes to a variable using its pointer instead of its name, for example:

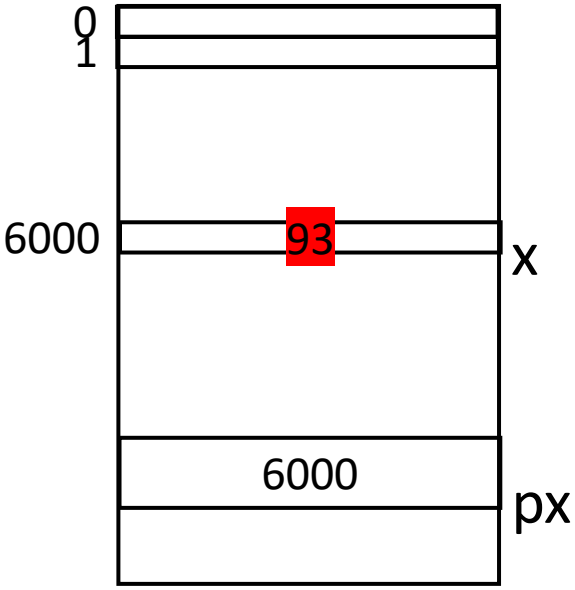
```
*px = 93;
```

change the contents of variable x from 37 to 93. Hence it is equivalent to the statement:

```
x=93;
```




Memory before the statement *px=93;



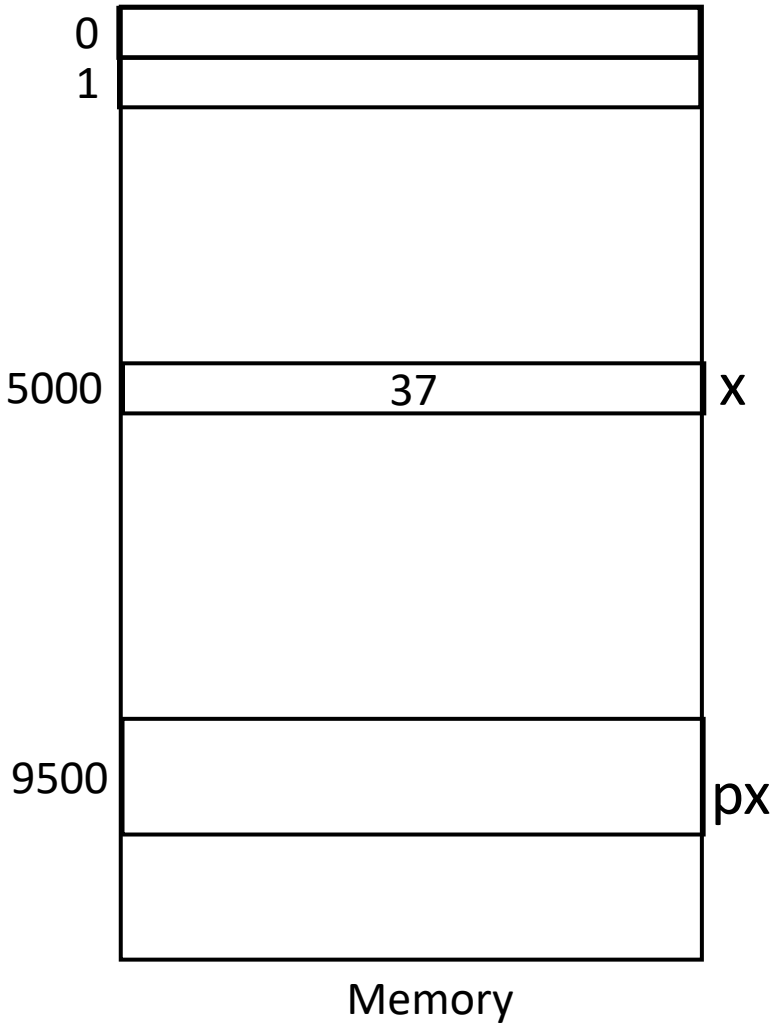

Memory after the statement *px=93;

# Pointers – cont.

```
int x = 37;
Int* px = &x;
```

Which one of the following statements evaluates to true?

```
(x==5000)
(x==37)
(&x==9500)
(&x==5000)
(px==9500)
(px==37)
(*px==9500)
(*px==37)
```

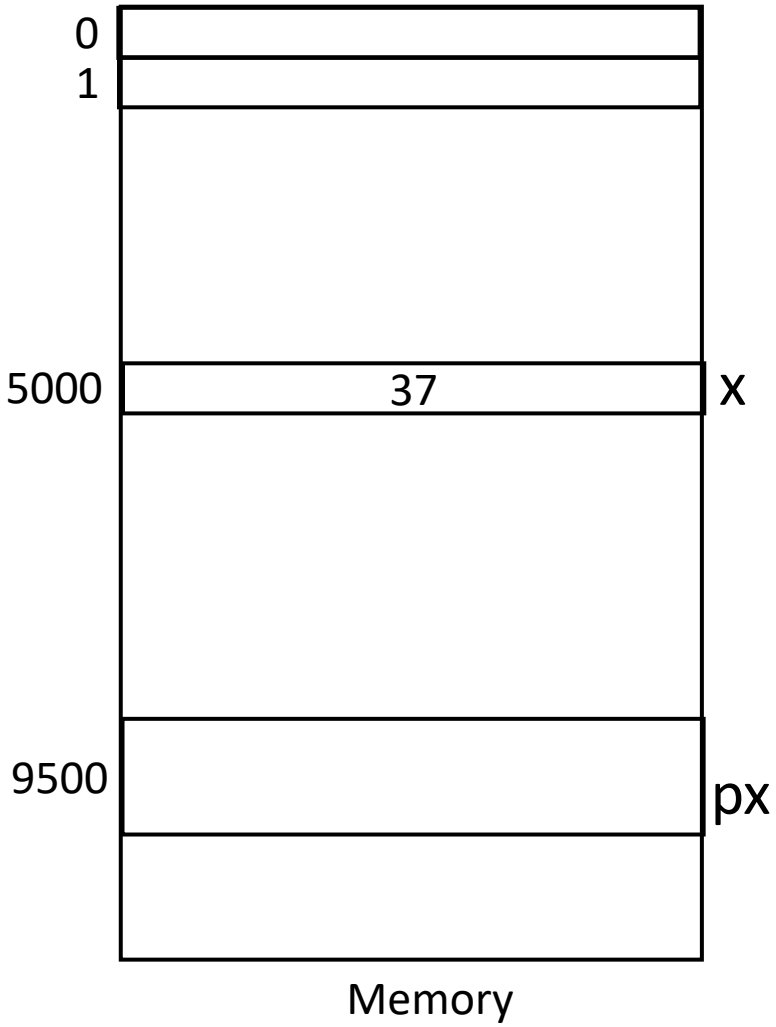


# Pointers - Example:

```
int x = 37;
px = &x  ;
```

Which one of the following statements evaluates to true?

```
(x==5000)          false
(x==37)            true
(&x==9500)         false
(&x==5000)         true
(px==9500)         false
(px==37)           false
(*px==9500)        false
(*px==37)          true
```



# Pointers - declaration

```
int *p1;
char *p2;
double *p3;
```

- Note that the asterisk (*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator seen a bit earlier. They are simply two different things represented with the same sign.

```
int * x,y;
```

- In the previous line, `x` is declared as a pointer, but `y` is declared as an `int`.

# Pointers – example 1

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

# Pointers – example 1

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

# Pointers – example 2

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 =
                     //value pointed to by p1
  p1 = p2;           // (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

# Pointers – example 2

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 =
                     //value pointed to by p1
  p1 = p2;           // (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

# Pointers – nullptr

- Always a good idea to initialize pointers to a known value
- If no known object, then initialize to nullptr (which is 0)

```
int *x = nullptr;
```

- Then we can test for pointer validity before we deference it:

```
int y=15;
int *x=&y;
if(x!=nullptr)    // test before deref.
    (*x)++;
```

# The `this` pointer

- `this`: predefined pointer available to a class's member functions
- Always points to the instance (object) of the class whose function is being called
- Is passed as a hidden argument to all member functions
  - All except **static member functions** (static members are not a part of this course)

# pointers – cont.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    int x = 17;
    cout << &x << endl; // address-of operator

    // Create p (ptr to x) + print p and deferred p
    int* p = &x;
    cout << "pointer is: " << p << " value is: " << *p << endl;

    // Assign derefed p into y + print
    int y = *p;
    cout << "y: " << y << endl;

    // can we deref x?
    //cout << *x << endl;

    // set value of x to 42 USING p
    *p = 42;
    cout << "The value of x is " << x << endl;
}
```

```
0000004D4D52FDA0
pointer is: 0000004D4D52FDA0 value is: 17
y: 17
The value of x is 42
```

```
Cannot deref an int
```

# pointer to const & constant pointers

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    int x = 17;
    // Define a constant int
    const int a = 6;

    //int* q = &a; // won't work
    const int* q = &a;
    q = &x;
    // *q = 14; // compilation error

    // Here we are declaring a pointer that can only be set /
    // initialized ONCE. Just like any other constant thing.
    int* const r = &x;
    cout << "r: " << r << ' ' << *r << endl;

    int z = 64;
    //r = &z;  // cannot re-assign 'r', it's a const pointer
    *r = 64; // r can be used to modify x.

    // Here we have a pointer that can only ever point at what it was
    // initialized to, and also cannot be used to change the value
    // there.
    const int* const s = &x;
    cout << "s: " << s << ' ' << *s << endl;
}
```

```
r: 00000068E23EFD90 17
s: 00000068E23EFD90 64
```

Despite assuming address of x (x is mutable), q is still a pointer to const → (deref q) is still a const

**Can we point at it with a non-const pointer?**

- No. Why not? Because if we could, then that pointer could be used to change the int.

**Solution - Pointer to const**

- It is a "compile-time" protection, not a runtime protection.
- C++ does not check at runtime if the assignment would be legal based on the type of the left-hand side.
- A pointer that is not pointing to const is therefore free to modify what it is pointing at.

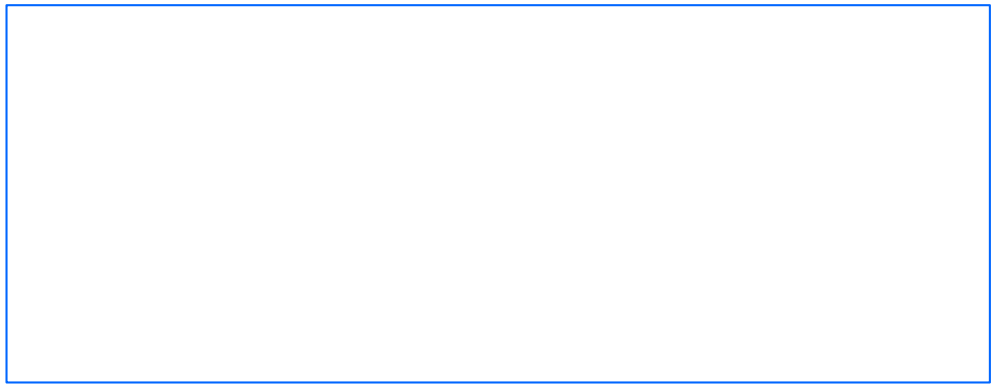
Unlike a reference, pointers can be modified and made to point to something else

**Constant pointer** – can only point to one thing and that cannot change!

# const pointers –cont.

```
#include <iostream>
using namespace std;

struct Thing { int val; };

void foo(const Thing& theThing) {
    // Thing* p = &theThing; // fails to compile
    const Thing* p = &theThing; // Has to promise to respect constness
    // p->val = 17; // "obviously" won't work but...
}
void bar(Thing theThing) {
    // Ok for pointer to const to point to something non-const
    theThing.val = 28; // But can with theThing
}

int main() {
    Thing aThing{42}; // Note the curly braces, just for the jollies.

    foo(aThing);
    cout << "After calling foo(): " << aThing.val << endl;

    bar(aThing);
    cout << "After calling bar(): " << aThing.val << endl;
}
```

?



# const pointers –cont.

```cpp
#include <iostream>
using namespace std;

struct Thing { int val; };

void foo(const Thing& theThing) {
    // Thing* p = &theThing; // fails to compile
    const Thing* p = &theThing; // Has to promise to respect constness
    // p->val = 17; // "obviously" won't work but...
}
void bar(Thing theThing) {
    // Ok for pointer to const to point to something non-const
    theThing.val = 28; // But can with theThing
}

int main() {
    Thing aThing{42}; // Note the curly braces, just for the jollies.

    foo(aThing);
    cout << "After calling foo(): " << aThing.val << endl;

    bar(aThing);
    cout << "After calling bar(): " << aThing.val << endl;
}
```

```
After calling foo(): 42
After calling bar(): 42
```

# const pointers –cont.

```
#include <iostream>
using namespace std;

struct Thing { int val; };

void foo(const Thing& theThing) {
    // Thing* p = &theThing; // fails to compile
    const Thing* p = &theThing; // Has to promise to respect constness
    // p->val = 17; // "obviously" won't work but...
}
void bar(Thing& theThing) {
    // Ok for pointer to const to point to something non-const
    theThing.val = 28; // But can with theThing
}

int main() {
    Thing aThing{42}; // Note the curly braces, just for the jollies.

    foo(aThing);
    cout << "After calling foo(): " << aThing.val << endl;

    bar(aThing);
    cout << "After calling bar(): " << aThing.val << endl;
}
```

```
After calling foo(): 42
After calling bar(): 28
```