

- They are objects that look and feel like pointers, but are smarter.
- look like pointers = have the same interface that pointers do

- smarter = do things that regular pointers don't

- They are objects that look and feel like pointers, but are smarter.
- look like pointers = have the same interface that pointers do
 - they need to support pointer operations like dereferencing (`operator*`) and indirection (`operator ->`).
 - an object that looks and feels like something else is called a **proxy object**, or just **proxy**
- smarter = do things that regular pointers don't

- They are objects that look and feel like pointers, but are smarter.
- look like pointers = have the same interface that pointers do
 - they need to support pointer operations like dereferencing (`operator*`) and indirection (`operator ->`).
 - an object that looks and feels like something else is called a **proxy object**, or just **proxy**
- smarter = do things that regular pointers don't
 - ... such as memory management

- Simplest example: `auto_ptr`, included in the standard C++ library.
 - header: `<memory>`

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                {delete ptr;}
    T& operator*()             {return *ptr;}
    T* operator->()            {return ptr;}
    // ...
};
```

INTERMISSION: THE EXPLICIT KEYWORD

- This keyword is a declaration specifier that can only be applied to in-class constructor declarations
 - An explicit constructor cannot take part in implicit conversions
 - It can only be used to explicitly construct an object

```
class C {
public:
    int i;
    explicit C(const C&) {}
    explicit C(int i) {}
    C() { i = 0; } };

class C2 {
public:
    int i;
    explicit C2(int i) {} };

C f(C c) {
    c.i = 2;
    // call to copy constructor
    return c;
}

void f2(C2) {}

void g(int i) {
    f2(i);
    // try the following line instead
    // f2(C2(i));
}

int main()
{
    C c, d;
    d = f(c); // c is copied
}
```

THE EXPLICIT KEYWORD (CONT'D)

- explicit on a constructor with multiple arguments has no effect, since such constructors cannot take part in implicit conversions
- however, explicit will have an effect if a constructor has multiple arguments and all but one of the arguments has a default value.

EXAMPLE OF USE

Instead of:	We then use:
<pre>void foo() { MyClass* p(new MyClass); p->DoSomething(); delete p; }</pre>	<pre>void foo() { auto_ptr<MyClass> p(new MyClass); p->DoSomething(); } • p now cleans up after itself.</pre>

WHY USE: LESS BUGS

Automatic cleanup. They clean after themselves, so there is no chance you will forget to deallocate.

Automatic initialization. You all know what a non-initialized pointer does. The default constructor now does the initialization to zero for you.

Dangling pointers. As stated many times before, dangling pointers are evil:

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
// p->DoSomething(); // We don't do that, p is dangling
p = 0; // we do this instead
q->DoSomething(); // Ouch, q is still dangling!
```

- Smart pointers can set their content to 0 once copied, e.g.

```
template <class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs) {
    if (this != &rhs) {
        delete ptr; ptr = rhs.ptr; rhs.ptr = 0;
    }
    return *this;
}
```

DANGLING POINTERS (CONT'D)

- The simplistic strategy to “change ownership” may not be suitable; other strategies can be implemented:
 - **Deep copy** the source into the target.
 - **Transfer ownership** by letting `p` and `q` point to the same object but transfer the responsibility for cleaning up from `p` to `q`.
 - **Reference counting**.
 - **Copy on write**: use reference counting as long as the pointer is not modified, and just before it gets modified copy it and modify the copy.
- Each strategy has advantages and disadvantages and are suitable for certain kind of applications.

WHY USE: EXCEPTION SAFETY

- Our old, simple example...

```
void foo() {
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

- ...generates a memory leak (at best!)

WHY USE: EXCEPTION SAFETY

- Our old, simple example...

```
void foo() {
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

- ...generates a memory leak (at best!) whenever `DoSomething` throws an exception.

WHY USE: EXCEPTION SAFETY

- Our old, simple example...

```
void foo() {
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

- ...generates a memory leak (at best!) whenever `DoSomething` throws an exception.
- We could of course take care of it by hand (pretty awkward; imagine now that you have some loops threw in for good measure):

```
void foo() {
    MyClass* p(new MyClass);
    try { p = new MyClass; p->DoSomething(); delete p; }
    catch (...) {delete p; throw; }
}
```

- With smart pointers we could let `p` clean up by itself.

WHY USE: GARBAGE COLLECTION

- C++ typically lacks garbage collection.
- But this can be implemented using smart pointers.
 - Simplest form of garbage collection: reference counting.
 - Other, more sophisticated strategies can be implemented in the same spirit.
 - * Remember, we can also have static variables in a class, and other goodies.

WHY USE: EFFICIENCY

- If the object pointed at does not change, there is no need to copy it. 'Nuff said.
 - copying takes both time and space
 - copy on write is our friend here
 - C++ strings are typically implemented in this manner

```
string s("Hello");
string t = s;      // t and s point to the same buffer of characters
t += " there!";   // a new buffer is allocated for t before
                  // appending, so that s is unchanged.
```

WHY USE: STL CONTAINERS

- STL containers (such as `vector`) store objects by value.
 - So you cannot store objects of a derived type.

```
class Base { /*...*/ };
class Derived : public Base { /*...*/ };
```

```
Base b;   Derived d;
vector<Base> v;
```

```
v.push_back(b); // OK
v.push_back(d); // no cake
```

WHY USE: STL CONTAINERS

- STL containers (such as `vector`) store objects by value.
 - So you cannot store objects of a derived type.

```
class Base { /*...*/ };
class Derived : public Base { /*...*/ };
```

```
Base b;   Derived d;
vector<Base> v;
```

```
v.push_back(b); // OK
v.push_back(d); // no cake
```

- You go around this by using **pointers**.

```
vector<Base*> v;
```

```
v.push_back(new Base);    // OK
v.push_back(new Derived); // OK
```

```
// obligatory cleanup, disappears when using smart pointers
for (vector<Base*>::iterator i = v.begin(); i != v.end(); ++i)
    delete *i;
```

- **But.**
 - STL containers do a lot behind our back
 - * in particular they delete (copy, etc.) objects without us noticing (e.g., when resizing themselves).
 - All the copies in a container **must thus be equivalent.**
 - The standard smart pointer and the ownership transfer pointers cannot be used safely in an STL container.

- The simplest smart pointer `auto_ptr` is probably suited for **local variables.**
- A **copied pointer** is usually useful for **class members.**
 - Think copy constructor and you think deep copy.
- Due to their nature **STL containers** require garbage collected pointers (e.g., **reference counting**).
- Whenever you have **big objects** you are probably better off using **copy on write** pointers.

- These notes are based on
 - <http://ootips.org/yonat/4dev/smart-pointers.html>
 - The Web page contains implementations and further readings.