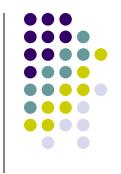
C++ March 5, 2015

Mike Spertus

mike spertus@symantec.com



Pointers



 Pointers to a type contain the address of an object of the given type or inheriting from the given type.

```
A *ap = new A;
```

Arithmetic on pointers

```
A *aap = new A[10];

*(aap + 5) is the 5<sup>th</sup> element of the array

It doesn't add 5 to the address, but adds enough to get to the

fifth element (starting from 0)
```

Dereference with *

$$A a = *ap;$$

-> is an abbreviation for (*_).
 ap->foo(); // Same as (*ap).foo()

• If a pointer is not pointing to any object, you should make sure it is nullptr ap = nullptr; // don't point at anything if (ap) { ap->foo(); }

Smart pointers that own their data



- unique_ptr<T> points to a T object. Its destructor deletes the object
- Is the following correct?
 - unique_ptr<char> = new char[10];





- No!
- We need to delete the array with delete[], but unique_ptr<char> uses delete
- With ordinary pointers, we manually remember it's pointing to an array and then cause the correct deletion function
- How can we tell the unique_ptr to automatically use the correct operator delete?
- Answer: Use a unique_ptr to an array
 - unique ptr<char[]> = new char[10];

How to transfer ownership into an out of a unique_ptr



- unique_ptr<A> ap(new A); // ap owns the new A
 // delete old A and own new A
 ap.reset(new A);
- Suppose we have the following functions

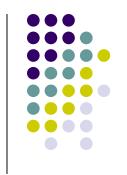
```
void f1(A *); // f1 will not del
void f2(A *); // f2 is responsible for deleting
// Funcs returning new obj should
// return a unique_ptr
unique_ptr<A> Afactory();
f1(ap.get()); // Pass the pointer
// Pass ptr and release ownership
f2(ap.release());
ap = Afactory();
// You can't copy unique pointers because that
// would create ambiguous ownership. You
// move them instead
unique ptr<A> ap2 = move(ap);
```

make_unique



- Since using owning raw pointers is error-prone, we prefer not to ever manually call new and delete
- Instead, we want creation functions that return owning smart pointers
- make_unique creates a new object and returns a rvalue unique ptr
- auto ap = make_unique<A>(1, 2, 3);
 is the same as
 unique_ptr<A> ap(new A(1, 2, 3));
 but you are never exposed to new, delete or
 raw pointers
- Best practice: Always use an owning pointer

shared_ptr



- Just like unique_ptr only can have multiple pointers to same object
- Once all of the pointers go away, the object is deleted
- Can create with make_shared
- A difference from unique ptr
 - Can copy (That's how you get multiple owners)
- Don't create two shared_ptr's from the same object
 - You'll get two reference counts!
 - Advanced: Read about enable_shared_from_this if you run into a situation where you need to create a shared_ptr from the this pointer

Pointers to functions



- The basic idea is usually that you describe a type by how it is used
 - int *ip; // Means *ip is an int
 - int (*fp)(int, int); // *fp can be called with 2 ints
- Let's show fp in action

```
• int f(int i, int j) { ... }
fp = &f;
fp(2, 3);
// The following line only works without captures
fp = [](int i, int j) { return i + j; }
```





- Sometimes we don't know what function we want to call until runtime

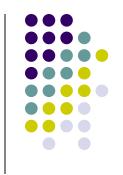
Pointers to members

```
struct A {
   int i;
   int j;
   void foo(double);
   void bar(double);
};
```

- We would like to be able to point to a particular member of A
 - Not an address because we haven't specified an A object
 - More like an offset into A objects

```
• int A::*aip = &A::i;
void (A::*afp) (double) = &A::foo;
A *ap = new A;
A a;
ap->*aip = 3; // Set ap->i to 3
(a.*afp) (3.141592); // Calls a.foo(3.141592)
```





Consider

```
vector<Animal *> zoo;
zoo.push back(new Elephant);
zoo.push back(new Zebra);
zoo.push back (new Bear);
cout << "Feeding time (f) or Bedtime (b)?"
char c;
cin >> c;
auto ap
  = c == 'f' ? &Animal::eat : &Animal::sleep;
for(auto animal : zoo) {
  animal->*ap();
```

Using with standard smart pointers



- Unfortunately, unique ptr and shared ptr don't overload operator->*(), so if we want to make the previous example delete objects when the zoo closes (or there is an exception when constructing an animal), we should modify it as shown below

References



- Like pointers but different
 - Allow one object to be shared among different variables
 - Can only be set on creation and never changed
 - Reference members must be initialized in initializer lists

```
struct A {
   A(int &i) : j(i) {}
   int &j;
};
```

Cannot be null

Not all callables can be assigned to a function pointer



- Can only assign a lambda to a function pointer if it does not have a capture list
 - See homework
- Can't assign a functor to a function pointer

std::function



- We have just discussed function pointers, but in C++, functions aren't the only thing that can be called
 - Call a function
 - Call a lambda
 - Call a functor
 - Call a member function

std::function can hold anything callable



```
struct WeightedMean {
  WeightedMean (vector < double > const & weights)
    : weights (weights) {}
  double operator()(vector<double> const &data) {
    return
      inner product(data.begin(), data.end(),
                     weights.begin(), 0.0)
      / accumulate(weights.begin(), weights.end(), 0.0);
  vector<double> weights;
};
function<double(vector<double> const &)> averager
  = WeightedMean(\{1.5, 3.6, 4.2\}); // OK
cout << "The average home price is ";</pre>
cout << averager(getHomePrices()) << endl;</pre>
```

You can even put a member pointer in a std::function



 It acts like a function whose first argument is the "this" pointer (or even a reference).

Often you can choose between a std::function and a template



In the below code, tmpl apply and fn apply can be used similarly

```
template<typename Callable>
double
tmpl apply(Callable c, vector<double> const &data)
  return c(data);
double
fn apply(function<double(vector<double> const &)> c, vector<double> const
&data)
  return c(data);
void f()
  tmpl apply(mean, \{1.7, 2.3\}); // OK
  fn apply(mean, \{1.7, 2.3\}); // OK
  tm\overline{p}1 apply (WeightedMean({1.2, 3.4}), {1.7, 2.3}) // OK
  fn apply (WeightedMean (\{1.2, 3.4\}), \{1.7, 2.3\}) // OK
```

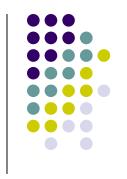
See HW

Standard exception types



- Even though technically, you can throw exceptions of any type, you should always have your exceptions inherit from std::exception, std::runtime_error, or std::logic_error
- Another good best practice: Throw by value but catch by reference
- Remember, don't use exception specifications
 - But note that C++11 introduces noexcept keyword (beyond the scope of this quarter, but will come back to it next quarter)





 Tuples are a generalization of std::pair to any number of fields

```
pair<string, int> si = make_pair("str", 2);
// di will be a tuple<double,int, char>
auto di = make_tuple(2.5, 3, 'c');
cout << get<0>(di) // prints 2.5
cout << get<char>(di); // prints 'c' (C++14)
int three = get<1>(di);
```

Tuples



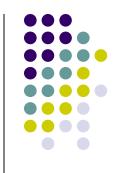
- Tuples are very useful for creating compound types on the fly
- We will implement an improved version of tuple from scratch in a few weeks

One annoyance with tuple vs pair



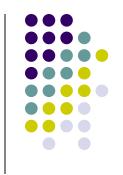
- pair<int, int> f() { return {1, 2}; // ok }
- tuple<int, int, char> f() { return {1, 2, 'u'}; // Error }
- We'll examine the reason for this and fix it in our own tuple class
- For more information, see Improving Pair and Tuple (revision 1) by Daniel Krugler
 - http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2013/n3739.html
 - Warning: This won't make sense until next quarter

Working with time



- Many programs work with time periods, but traditionally they just use integers or typedef to represent time
 - clock_t, time_t, dwMilliseconds, etc.
- This is very type-unsafe and error prone
 - For example, accidentally giving the number of milliseconds to a function that expects a number of seconds
 - Compiler won't even warn

durations



- C++ provides a chrono::duration type that represents an interval of time
- duration is actually a template class that indicates what the "clock tick" is
 - You can get the number of ticks with the count() method
- Usually, you don't use the template arguments directly (they are a little complicated), instead there are typedefs and literals for the different time units (C++14)

```
using namespace std::literals::chrono_literals;
using namespace std::chrono;
auto threeSeconds = 3s;
cout << threeSeconds.count() << " seconds" << endl;
minutes minutesInTwoHours = 2h;
// Casting milliseconds to hours is unsafe, so needs
// an explicit cast
hours h = duration cast<hours>(123456ms);
```

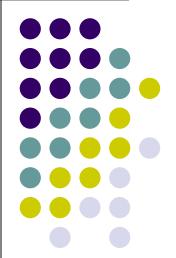
- The literal suffixes leverage a feature called "user-defined literals"
 - Later, we will learn how to create our own literals this way

time_point



- Rather than a duration, a time_point represents a particular point in time
- Again, it's easier to use standard functions to create them

Homework







- This problem consists of a series of types. Write a program that defines variables of each type set to some meaningful value (You are highly encouraged to check with a compiler). If the type is callable, the program should call it. Googling "c++ declarators" may help. Each one you get is worth 2 points.
- Example problem 1: int *
 - One possible answer: int *ip = new int;
 - Another possible answer

```
int i = 5;
int *ip = &i;
```

- Example problem 2: int &
 - One possible answer:

```
int i = 5;
int &ir(i);
```

HW 9.1 (cont)

- int *
- int &
- double
- A * (A is any appropriate class).
- char const *
- char const &
- long[7]
- int **
- int *&
- float &
- int (*) () (See http://www.newty.de/fpt/index.html)
- int (*&)()
- char *(*)(char *, char *)

HW 9.1 (cont)



See

http://www.informit.com/guides/content.aspx? g=cplusplus&seqNum=142 or the standard

```
int A::*
int (A::*) (int *)
int (A::**) (int *)
int (A::*&) (int *)
int (A::*) (double (*) (float &))
void (*p[10]) (void (*)());
```

HW 9.2



- In slide 5, the function £2 would be better with a different signature based on the best practices we've emphasized. What should it be?
- Unfortunately, even if you use unique_ptr and shared_ptr correctly, it is still possible to leak an object (i.e., the object is not deleted when you are done using it). Explain how this can happen
 - For extra credit, discuss how you might handle this