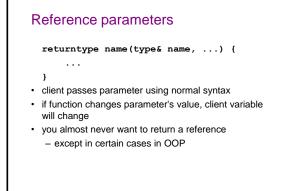
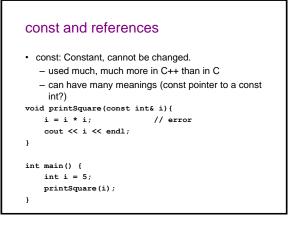


11/30 C++ intro		12/2 C++ intro		12/4	12/4 12/11 Final prep, evaluations	
12/7		12/9				
	12/15 Final					
• HW7 is	out: nev		/ due date			
HW7 isFinish la			/I due date			

References References vs. pointers • don't use * and & to reference / dereference (just & at • type& name = variable; assignment) • reference: A variable that is a direct alias for another variable. · cannot refer directly to a reference; just refers to what it refers - any changes made to the reference will affect the original to - like pointers, but more constrained and simpler syntax · a reference must be initialized at declaration - an effort to "fix" many problems with C's implementation of - int& r; // error pointers · a reference cannot be reassigned to refer to something else • Example: int x = 3, y = 5; int x = 3;int& r = x; // now use r just like any int int& r = x;r = y;// sets x == 5, r == 5 // r == 4, x == 4 · a reference cannot be null, and can only be "invalid" if it refers to • value on right side of = must be a variable, not an an object/memory that has gone out of scope or was freed expression/cast



r++;



Strings

- #include <string>
- · C++ actually has a class for strings
 - much like Java strings, but mutable (can be changed)
 - not the same as a "literal" or a char*, but can be implicitly converted

string str1 = "Hello"; // impl. conv.

Concatenating and operators

string str3 = str1 + str2;

if (str1 == str2) { // compares characters if (str1 < str3) { // compares by ABC order

- char c = str3[0];
- // first character

method	description		
append(str)	append another string to end of this one		
c_str()	return a const char* for a C++ string		
clear()	removes all characters		
compare(str)	like Java's compareTo		
<pre>find(str [, index]) rfind(str [, index])</pre>	search for index of a substring add characters to this string at given index number of characters in string adds a character to end of this string		
insert(index, str)			
length()			
push_back(<i>ch</i>)			
replace(index, len, str)	replace given range with new text substring from given start index		
<pre>substr(start [, len])</pre>			

String concatenation

• a string can do + concatenation with a string or char*, but not with an int or other type: string s1 = "hello"; string s2 = "there";

s1 = s1 + " " + s2; // ok

s1 = s1 + 42;

// error to build a string out of many values, use a stringstream

- works like an ostream (cout) but outputs data into a string
- call .str() on stringstream once done to extract it as a string #include <sstream>

stringstream stream;

stream << s1 << " " << s2 << 42;</pre> s1 = stream.str(); // ok

Libraries

#include <cmath>

library	description
cassert	assertion functions for testing (assert)
cctype	char type functions (isalpha, tolower)
cmath	math functions (sqrt, abs, log, cos)
cstdio	standard I/O library (fopen, rename, printf)
cstdlib	standard functions (rand, exit, malloc)
cstring	char* functions (strcpy, strlen)
	(not the same as <string>, the string class)</string>
ctime	time functions (clock, time)

	stack-allocated (same as C):
	type name[size];
•	heap-allocated:
	<pre>type* name = new type[size];</pre>
	- C++ uses new and delete keywords to allocate/free memory
	 arrays are still very dumb (don't know size, etc.)
	<pre>int* nums = new int[10];</pre>
	for (int $i = 0; i < 10; i++$) {
	nums[i] = i * i;
	}
	delete[] nums;

malloc vs. new

	malloc	new
place in language	a function	an operator (and a keyword)
how often used in C	often	never (not in language)
how often used in C++	rarely	frequently
allocates memory for	anything	arrays, structs, and objects
returns what	void* (requires cast)	appropriate type (no cast)
when out of memory	returns NULL	throws an exception
deallocating	free	<pre>delete (or delete[])</pre>

Exceptions

- · exception: An error represented as an object or variable.
 - C handles errors by returning error codes
 - C++ can also represent errors as exceptions that are thrown / caught
- throwing an exception with throw:
 - double sqrt(double n) {

```
if (n < 0) {
```

```
throw n; // kaboom
```

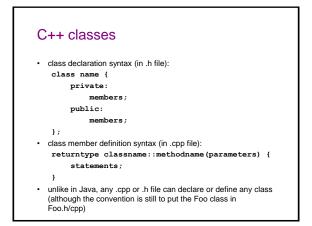
}

```
...
```

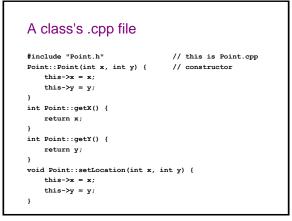
- can throw anything (a string, int, etc.)
- can make an exception class if you want to throw lots of info: #include <exception>

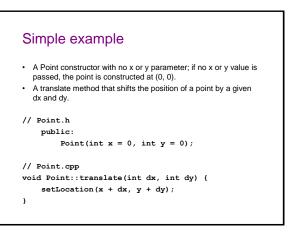
More about exceptions

- catching an exception with try/catch: try { double root = sqrt(x); } catch (double d) { cout << d << " can't be squirted!" << endl; }
 throw keyword indicates what exception(s) a method may throw - void f() throw(); // none - void f() throw(int); // may throw ints
 predefined exceptions(from std::exception):
- predefined exceptions (from std::exception): bad_alloc, bad_cast, ios_base::failure,



A class's .h file #ifndef POINT H #define POINT H class Point { private: int x; int y; // fields public: Point(int x, int y); // constructor int getX(); // methods int getY(); double distance(Point& p); void setLocation(int x, int y); }; #endif





More about constructors

- initialization list: alternate syntax for storing parameters to fields
 - supposedly slightly faster for the compiler
 - class::class(params) : field(param), ...,

field(param) {
 statements;

- 1
- Point::Point(int x, int y) : x(x), y(y) {}
- if you don't write a constructor, you get a default () constructor
 - initializes all members to 0-equivalents (0.0, null, false, etc.)

<text><list-item><list-item><text>

Constructing objects

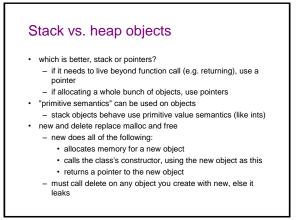
- client code creating stack-allocated object: type name(parameters);
 Point p1(4, -2);
- creating heap allocated (pointer to) object:
 type* name = new type(parameters);
 Point* p2 = new Point(5, 17);
- · in Java, all objects are allocated on the heap
- in Java, all variables of object types are references (pointers)

A client program

#include <iostream>
#include "Point.h"
using namespace std;

Client with pointers

```
#include <iostream>
#include "Point.h"
using namespace std;
int main() {
   Point* p1 = new Point(1, 2);
   Point* p2 = new Point(4, 6);
   cout << "pl is: (" << pl->getX() << ", "
        << pl->getY() << ")" << endl; // pl is: (1, 2)
   cout << "p2 is: (" << p2->getX() << ", "
        << p2->getY() << ")" << endl; // p2 is: (4, 6)
   cout << "dist : " << p1->distance(*p2) << endl;</pre>
   delete p1;
                                       // dist : 5
                // free
   delete p2;
    return 0;
}
```



Why doesn't this code change p1?

```
int main() {
    Point pl(1, 2);
    cout << pl.getX() << "," << pl.getY() << endl;
    example(p1);
    cout << pl.getX() << "," << pl.getY() << endl;
    return 0;
}
void example(Point p) {
    p.setLocation(40, 75);
    cout << "ex:" << p.getX() << "," << p.getY() << endl;
}
// 1,2
// ex:40,75
// 1,2</pre>
```

Object copying

- · a stack-allocated object is copied whenever you:
 - pass it as a parameter foo (p1) ;
 - return it return p;
 - assign one object to another p1 = p2;
- · the above rules do not apply to pointers
 - object's state is still (shallowly) copied if you dereference/assign

```
*ptr1 = *ptr2;
```

 You can control how objects are copied by redefining the = operator for your class (ugh)

Objects as parameters

We generally don't pass objects as parameters like this:

```
double Point::distance(Point p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
```

```
return sqrt(dx * dx + dy * dy);
```

```
}
```

- on every call, the entire parameter object p will be copied
- this is slow and wastes time/memory
- it also would prevent us from writing a method that modifies p

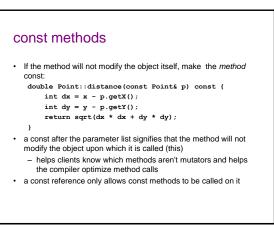
References to objects

· Instead, we pass a reference or pointer to the object:

```
double Point::distance(Point& p) {
    int dx = x - p.getX();
    int dy = y - p.getY();
    return sqrt(dx * dx + dy * dy);
}
```

- · now the parameter object p will be shared, not copied
- · are there any potential problems with this code?

const object references If the method will not modify its parameter, make it const double Point::distance(const Point& p) { int dx = x - p.getX(); int dy = y - p.getY(); return sqrt(dx * dx + dy * dy); } the distance method is promising not to modify p - if it does, a compiler error occurs - clients can pass Points via references without fear that their state will be changed



const and pointers

- const Point* p
 - p points to a Point that is const; cannot modify that Point's state
 - can reassign p to point to a different Point (as long as it is const)
- Point* const p
 - p is a constant pointer; cannot reassign p to point to a different object
 - can change the Point object's state by calling methods on it
- const Point* const p
 - p points to a Point that is const; cannot modify that Point's state
 - p is a constant pointer; cannot reassign p to point to a different object
- · (This is not one of the more beloved features of C++.)

Pointer, reference, etc.?

- How do you decide whether to pass a pointer, reference, or object? Some principles:
 - Minimize the use of object pointers as parameters.
 (C++ introduced references for a reason. They are safer and saner.)
 - Minimize passing objects by value, because it is slow, it has to copy the entire object and put it onto the stack, etc.
 - In other words, pass objects as references as much as possible; but if you *really want* a copy, pass it as a normal object.
 - Objects as fields are usually pointers (why not references?).
 - If you are not going to modify an object, declare it as const.
 - If your method returns a pointer/object field that you don't want the client to modify, declare its return type as const.

