Computing with functions
Overview

• Most programs that we have seen treat objects as abstract data types
  – they define both state and behavior
  – the state is primary
  – the behavior describes how to manipulate the state

• Sometimes, it is useful to treat behavior as more important than state
A classic example

- Many programming languages have sort functions as part of their libraries
- It is usually useful to be able to specify a comparison function as an argument to the sort function
Another example

- Suppose we want a generic linear-search function
- We have seen how to make the function independent of the data structure being searched
- What about making it independent of the search criterion as well?
Searching for a particular value

• The `find` function looks for the first element with a given value:

```cpp
template<class It, class X>
It find(It begin, It end, const X& x)
{
    while (begin != end && *begin != x)
        ++begin;
    return begin;
}
```

• How can we generalize the search criterion?
Generalizing the search criterion

- We want something to which we can hand a sequence element and get an answer: yes or no
- It seems to make sense for that something to be a function whose input is an $X$ and output is a `bool`
Rewriting the search function

```cpp
template<class It, class X>
It find2(It begin, It end, bool (*f)(X))
{
    while (begin != end && !f(*begin))
        ++begin;
    return begin;
}
```

• Can we generalize it even more?
Further generalization

• We need not insist that `f` be a function. It can be any appropriate type:

```cpp
template<class It, class F>
It find_if(It begin, It end, F f)
{
    while (begin != end && !f(*begin))
        ++begin;
}
```

• How might `f` be anything other than a function?
Function objects

• In C++, we can call any object as if it were a function, provided that the object has `operator()` defined.

• In other words, if `obj` is an object, then `obj(x)` means `obj.operator()(x)`.

• Of course, `obj` has to be of a type with `operator()` defined.

• We call such objects function objects.
In other words...

- The `find_if` function will accept any function or function object as its third argument.
- It will call the function (or call the `operator()` member of the object) to test each element of the sequence.
Example

• Find the first white-space character in a string:
  \texttt{find(s.\ begin\n(), s.\ end(), isspace)}
A more interesting example

• Suppose that b and e are iterators that delimit a sequence, and we want to find the first element that is >10
• We might write a function
  
  ```
  bool gt10(int x) { return x > 10; }
  ```

• and then call
  
  ```
  find(b, e, gt10)
  ```

• But what if we want to find the first element that is >n?
Doing it the hard way

```cpp
int xx;
bool gt_xx(int x) { return x > xx; }

• and then, we might say
  
  xx = n;
  ...find_if(b, e, gt_xx) ...

• This approach is ugly!

• Why?
Why the approach is ugly

• It relies on a global variable
• To use it, you must
  – set the state explicitly (by assigning to the variable), and then
  – call the function
• In effect, the function relies on hidden state
How to clean it up

• Bind the state and the function together into a function object:

```cpp
class gt_n {
public:
    gt_n(int n0): n(n0) { }
    bool operator()(int x)
    {
        return x > n;
    }
private:
    int n;
};
```
Using class `gt_n`

- To find the first element >10:
  ```cpp
  find_if(b, e, gt_n(10))
  ```
- To find the first element >x:
  ```cpp
  find_if(b, e, gt_n(x))
  ```
- In both cases, global variables are unnecessary
It might be nice if...

- Another way to get rid of the global variable would be to make it local:
  
  ```
  {
    int n;
    bool gt_n(int x) { return x > n; }
    ...find_if(b, e, gt_n) ...
  }
  
  - But C++ doesn’t allow this technique
  - Why not?
Nested functions

• Programming languages of the Algol and Pascal family generally allow nested functions
• C and C++ do not
• The reason has to do with ease of implementation: While a function is executing, it sees only its own local variables and all global variables
Function objects simulate nested functions

- If a function could be nested inside another, you would be able to get at the inner function’s local variables, or those of the outer function(s), or global variables.
- A member function can get at its local variables, or its object’s members, or global variables.
Generating function objects

- Our `gt_n` type lets us create function objects that encapsulate comparison with a particular value.

- It would be tricky to do that even with nested functions (because it needs GC):

```cpp
bool (*gt_n(int n))(int)
{
    bool f(int x) { return x > n; }
    return f;
}
```
Two problems

• Allowing nested functions in a language potentially complicates the calling sequence for all functions
• Allowing functions to return nested functions as values causes trouble unless the language supports garbage collection
• C++ pushes the complexity into objects
How do other languages do it?

- Functional languages treat functions as first-class values:
  
  ```
  find_if(b, e, (fn x => x > n))
  ```

- Pure object-oriented languages (Smalltalk, Java) don’t have functions as separate entities at all
Function objects are objects

- Because function objects are objects, we can perform computations on them.
- It is possible to write functions (and the C++ standard library includes some such functions) that make it unnecessary to define classes such as `gt_n` at all.
Some sample library functions

- Template class `greater` is defined so that `greater<T>()(x, y)` has the same value as `x > y` (and similarly for `less`, `equal_to`, ...)
- If `f` is a function object, then template function `bind1st(f, x)(y)` has the same value as `f(x, y)` (and similarly for `bind2nd`
Using greater and bind2nd

- To find the first element >n:
  
  ```cpp
  find(b, e, bind2nd(greater<T>(), n))
  ```
Making binders work

• C++ binders are a nice example of making a high-level abstraction work in a language that wasn’t designed in advance to support it
• Binders and function objects rely on a mixture of code and conventions
Function object conventions

• Every function object has a member called `result_type` that names the type of its result

• In addition,
  - if it has a single argument, it has a member named `argument_type`
  - if it has two arguments, it has members named `first_argument_type` and `second_argument_type`
template<class A, class R>
struct unary_function {
    typedef A argument_type;
    typedef R result_type;
};

template<class A1, class A2, class R>
struct binary_function {
    typedef A1 first_argument_type;
    typedef A2 second_argument_type;
    typedef R result_type;
};
template<class T> class greater:
  public binary_function<T, T, bool>
  {
    public:
    bool operator() (const T& x, const T& y) const
    {
      return x > y;
    }
  };

Making bind2nd work

- The result of `bind2nd(f, x)` has to include the values of `f` and `x`
- Therefore, it has to have a type that includes the types of `f` and `x`
- We need an auxiliary type, which we will call `binder2nd`, to do the work
Definition of `binder2nd`
Member functions of `binder2nd`

```cpp
template<class Op>
binder2nd::binder2nd(
    const Op& o,
    const typename Op::second_argument_type& v):
    op(o), value(v) {}

template<class Op>
binder2nd::result_type operator()
    (const typename Op::first_argument_type& arg)
    const
{
    return op(arg, value);
}
```
Definition of bind2nd

template<class Op, class T>
binder2nd<Op> bind2nd(const Op& op, const T& t) 
{
    return binder2nd<Op>(op,
    typename Op::second_argument_type(t));
}
The point of all this code

- Although the types are somewhat messy,
  - the classes themselves are small
  - they can be combined in useful ways
  - the techniques used to build them can be used in other contexts
- Objects can be abstractions of behavior, not just of data structures
Other relevant library functions

• If \( f \) is a (pointer to a) function, \( \text{ptr\_fun}(f) \) is the corresponding function object.

• If \( \text{pred} \) is a unary (function object) predicate, \( \text{not\_1}(\text{pred}) \) is a predicate that yields the inverse result.
Using ptr_fun and not1

- Find the first non-space character in the string s:

```cpp
find_if(s.begin(), s.end(), not1(ptr_fun(isspace)));
```
A few more examples

- Flip the sign of every element of \( x \):
  
  ```cpp
  transform(x.begin(), x.end(), x.begin(), 
             negate<typename x::value_type>());
  ```

- Replace every pointer to a null-terminated string that compares equal to “C” by a pointer to “C++”
  
  ```cpp
  replace_if(x.begin(), x.end(), 
             not1(bind2nd(ptr_fun(strcmp), "C")), 
             "C++");
  ```
Projects

• Each team will be expected to demonstrate its project
  - be prepared to answer design and process related questions
• Each team has to find appropriate computing facilities for the demonstration and schedule a mutually agreeable time
• All demonstrations during exam week
Project scheduling

• If we heard from you by 10:30 PM, April 20 (well beyond the original deadline), we accommodated your requests.
• If not, it is now your problem! Either pick an open slot or trade with another project.
Project reviews for Monday, 5/17

- All 4 slots still open
  - 10:30 - 11:30
  - 1:30 - 2:30
  - 3:00 - 4:00
  - 4:30 - 5:30
Project reviews for Tuesday, 5/18

- 10:30 Campus Calendar
- 1:30 Direct Chat
- 3:00 AT 5000
- 4:30 Sound Images
Project reviews for Wednesday 5/19

- 10:30  This space available
- 1:30  Space Dust
- 3:00  Redemption
- 5:15  Clipbook
Project reviews for Thursday, 5/20

- 10:30 This space available
- 1:30 Online Trading
- 3:00 Project Vulcan
- 4:30 Logic Studio