Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 20

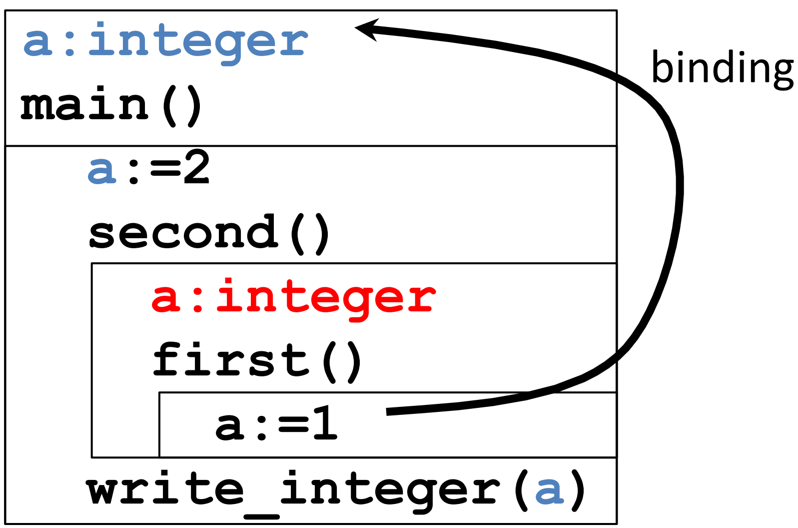
- Dynamic scoping and its implementation
  - Association lists
  - Central Reference Tables
- Subroutines as parameters: Deep and Shallow binding
- Subroutines as result: unlimited extent

Dynamic Scoping

- Scope rule: the "current" binding for a given name is the one encountered most recently **during execution**
- Typically adopted in (early) functional languages that are interpreted
- Perl v5 allows you to choose scope method for each variable separately
- With dynamic scope:
  - Name-to-object bindings *cannot* be determined by a compiler in general
  - Easy for interpreter to look up name-to-object binding in a stack of declarations
- Generally considered to be "a bad programming language feature"
  - Hard to keep track of active bindings when reading a program text
  - Most languages are now compiled, or a compiler/interpreter mix
- Sometimes useful:
  - Unix environment variables have dynamic scope

Effect of Static Scoping

Program execution:

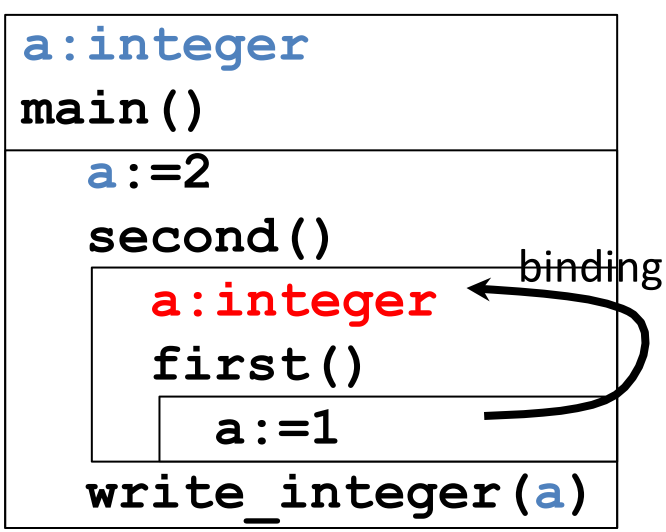


Program prints "1"

- The following pseudo-code program demonstrates the effect of scoping on variable bindings:
-
```
a:integer
procedure first(){
   a:=1}
procedure second(){
   a:integer
   first()}
procedure main(){
   a:=2
   second()
   write_integer(a)}
```

Effect of Dynamic Scoping

Program execution:



Program prints "2"

- The following pseudo-code program demonstrates the effect of scoping on variable bindings:
-
```
a:integer
procedure first(){
  a:=1}
procedure second(){
  a:integer
  first()}
procedure main(){
  a:=2
  second()
  write_integer(a)}
```

Binding depends on execution

Dynamic Scoping Problems

- In this example, function **`scaled_score`** probably does not do what the programmer intended: with dynamic scoping, **`max_score`** in **`scaled_score`** is bound to **`foo`**'s local variable **`max_score`** after **`foo`** calls **`scaled_score`**, which was the most recent binding during execution:

```
max_score:integer   -- maximum possible score

function scaled_score(raw_score:integer):real{
  return raw_score/max_score*100
  ...}

procedure foo{
  max_score:real := 0  -- highest percentage seen so far
  ...
  foreach student in class
    student.percent := scaled_score(student.points)
    if student.percent > max_score
       max_score := student.percent
}
```

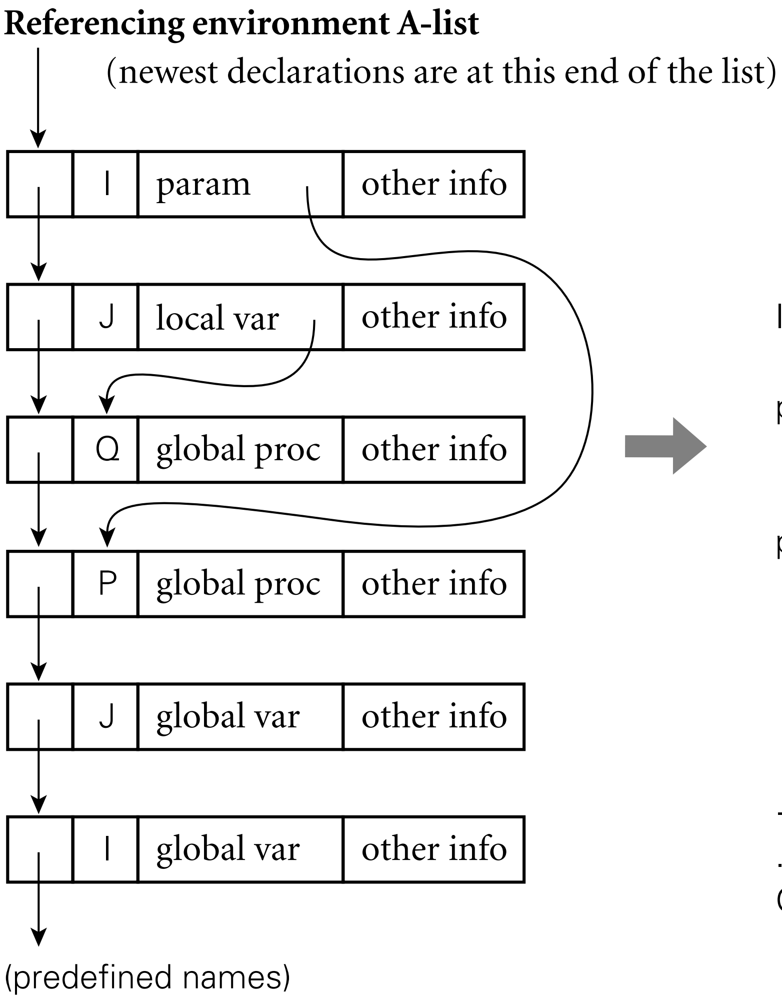
Dynamic Scope Implementation with Bindings Stacks

- Each time a subroutine is called, its local variables are pushed on a stack with their name-to-object binding
- When a reference to a variable is made, the stack is searched top-down for the variable's name-to-object binding
- After the subroutine returns, the bindings of the local variables are popped
- Different implementations of a binding stack are used in programming languages with dynamic scope, each with advantages and disadvantages

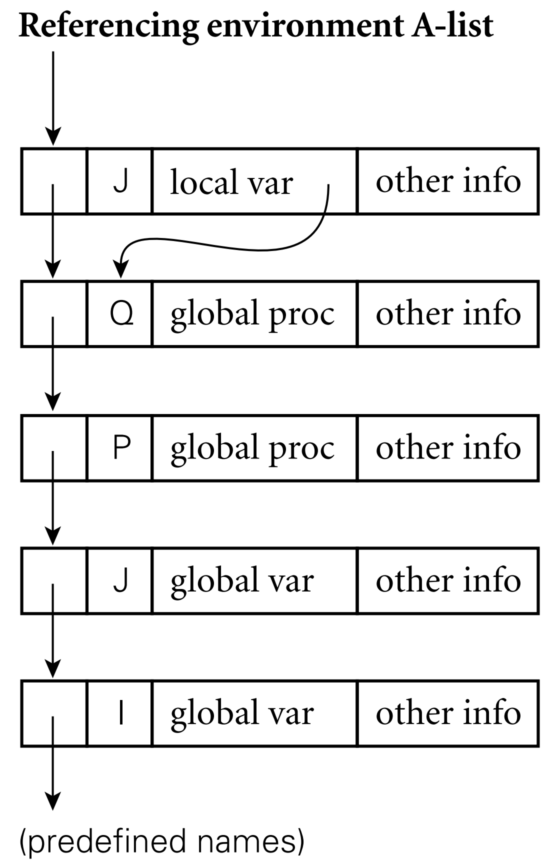
Dynamic Scoping with Association Lists (A-lists)

- List of bindings maintained at runtime
- Bindings are pushed on *enter_scope* and popped on *exit_scope*
- Look up: walks down the list till the first entry for the given name
- Entries in the list include information about types
- Used in many implementations of LISP
- Sometimes the A-list is accessible from the program, providing *reflexive* features
- Look up is inefficient

A-lists: an example




```
I, J : integer

procedure P (I : integer)
        . . .

procedure Q
        J : integer
        . . .
        P (J)
        . . .

-- main program
. . .
Q
```



A-list after entering P in the exection of Q

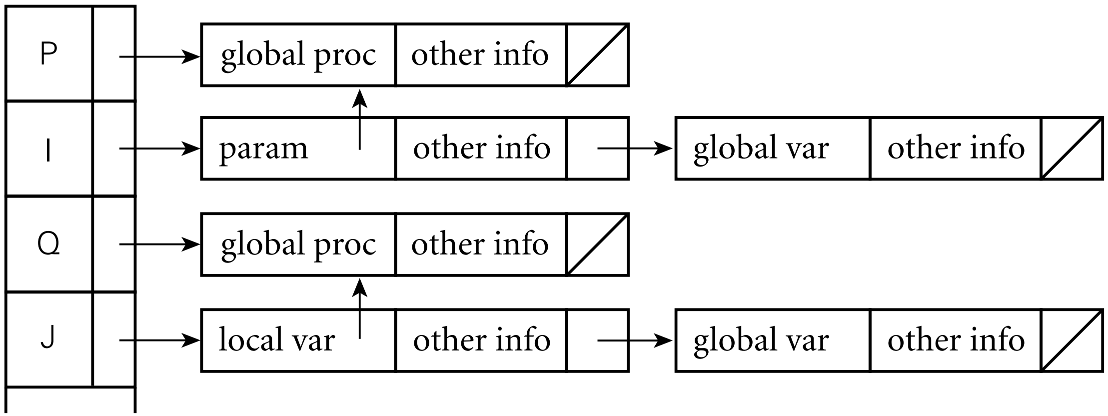
A-list after exiting P

Central reference tables

- Similar to LeBlanc&Cook hash table, but stack of scopes not needed (and at runtime!)
- Each name has a slot with a stack of entries: the current one on the top
- On *enter_scope* the new bindings are pushed
- On *exit_scope* the scope bindings are popped
- More housekeeping work necessary, but faster access than with A-lists

Central reference table

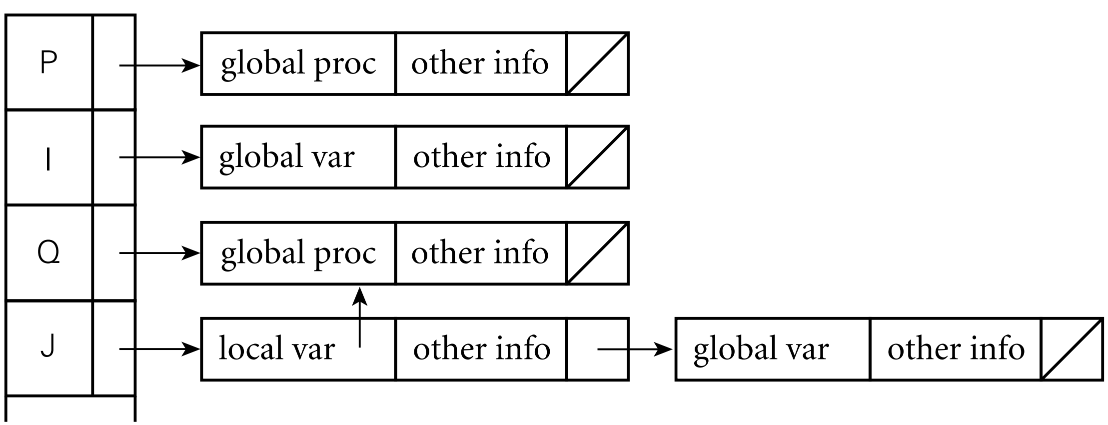
(each table entry points to the newest declaration of the given name)



(other names)

CRT after entering P in the exection of Q

Central reference table



(other names)

CRT after exiting P

```
I, J : integer

procedure P (I : integer)
        . . .

procedure Q
        J : integer
        . . .
        P (J)
        . . .

–– main program
. . .
Q
```

First, Second, and Third-Class Subroutines

- ***First-class object***: an object entity that can be passed as a parameter, returned from a subroutine, and assigned to a variable
  - Primitive types such as integers in most programming languages
- ***Second-class object***: an object that can be passed as a parameter, but not returned from a subroutine or assigned to a variable
  - Fixed-size arrays in C/C++
- ***Third-class object***: an object that cannot be passed as a parameter, cannot be returned from a subroutine, and cannot be assigned to a variable
  - Labels of goto-statements and subroutines in Ada 83
- Functions in Lisp, ML, and Haskell are unrestricted first-class objects
- With certain restrictions, subroutines are first-class objects in Modula-2 and 3, Ada 95, (C and C++ use function pointers)

Scoping issues for first/second class subroutines

- Critical aspects of scoping when
  - Subroutines are passed as parameters
  - Subroutines are returned as result of a function
- Resolving names declared **locally** or **globally** in the passed/returned subroutine is obvious
  - **Global** objects are allocated statically (or on the stack, in a fixed position)
    - Their addresses are known at compile time
  - **Local** objects are allocated in the activation record of the subroutine
    - Their addresses are computed as *base of activation record + statically known offset*

What about the Referencing Environment?

- If a subroutine is passed as an argument to another subroutine, when are the static/dynamic scoping rules applied? That is, what is the *referencing environment* of a subroutine passed as an argument?
    1) When the reference to the subroutine is first created (i.e. when it is passed as an argument)
    2) Or when the argument subroutine is called
- That is, what is the *referencing environment* of a subroutine passed as an argument?
    – Eventually the subroutine passed as an argument is called and may access non-local variables which by definition are in the referencing environment of usable bindings
- The choice is fundamental in languages with dynamic scope: **deep binding (1)** vs **shallow binding (2)**
- The choice is limited in languages with static scope

Effect of Deep Binding in Dynamically-Scoped Languages

Program execution:

```
main(p)
  bound:integer
  bound := 35
  show(p,older)
    bound:integer
    bound := 20
    older(p)
      return p.age>bound
    if  return value is true
      write(p)
```

Deep binding

Program prints persons older than 35

- The following program demonstrates the difference between deep and shallow binding:

```
function older(p:person):boolean
  return p.age > bound

procedure show(p:person,c:function)
  bound:integer
  bound := 20
  if c(p)
    write(p)

procedure main(p)
  bound:integer
  bound := 35
  show(p,older)
```

Effect of Shallow Binding in Dynamically-Scoped Languages

- The following program demonstrates the difference between deep and shallow binding:

```
function older(p:person):boolean
  return p.age > bound

procedure show(p:person,c:function)
  bound:integer
  bound := 20
  if c(p)
    write(p)

procedure main(p)
  bound:integer
  bound := 35
  show(p,older)
```

Program execution:

```
main(p)
   bound:integer
   bound := 35
   show(p,older)
      bound:integer
      bound := 20
      older(p)
         return p.age>bound
      if  return value is true
         write(p)
```

Shallow binding

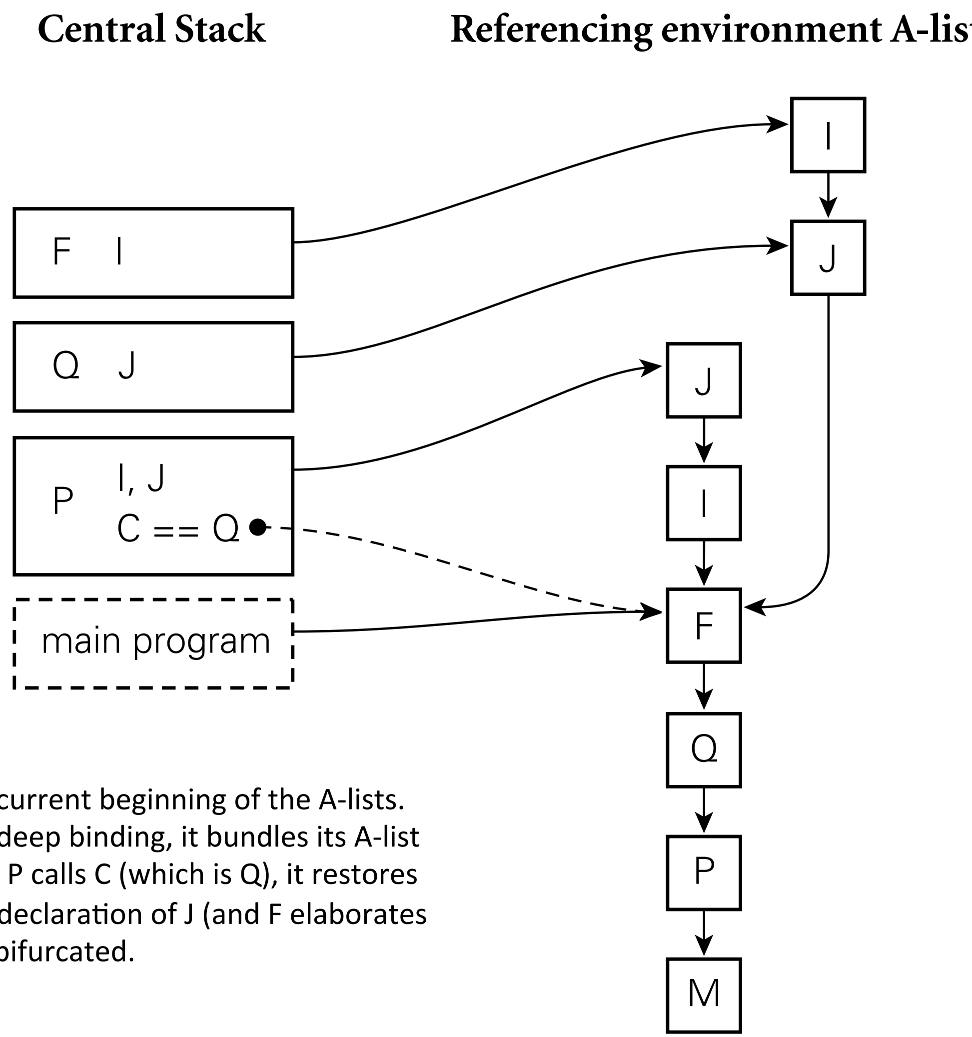
Program prints persons older than 20

Implementing Deep Bindings with Subroutine Closures

- Implementation of *shallow binding* obvious: look for the last activated binding for the name in the stack
- For *deep binding,* the referencing environment is bundled with the subroutine as a *closure* and passed as an argument
- A subroutine closure contains
  - A pointer to the subroutine code
  - The current set of name-to-object bindings
- Possible implementations:
  - With Central Reference Tables, the whole current set of bindings may have to be copied
  - With A-lists, the head of the list is copied

Closures in Dynamic Scoping implemented with A-lists

```
procedure P (procedure C)
    declare I, J
    call C

procedure F
    declare I

procedure Q
    declare J
    call F

-- main program
    call P(Q)
```



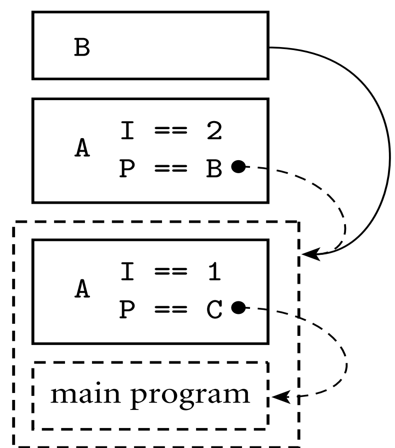
Each frame in the stack has a pointer to the current beginning of the A-lists. When the main program passes Q to P with deep binding, it bundles its A-list pointer in Q's closure (dashed arrow). When P calls C (which is Q), it restores the bundled pointer. When Q elaborates its declaration of J (and F elaborates its declaration of I), the A-list is temporarily bifurcated.

Deep/Shallow binding with **static** scoping

- Not obvious that it makes a difference. Recall:
- **Deep binding**: the scoping rule is applied when the subroutine is passed as an argument
- **Shallow binding**: the scoping rule is applied when the argument subroutine is called
- In both cases non-local references are resolved looking at the static structure of the program, so refer to the same binding declaration
- **But in a recursive function the same declaration can be executed several times: the two binding policies may produce different results**
- No language uses shallow binding with static scope
- Implementation of deep binding easy: just keep the static pointer of the subroutine in the moment it is passed as parameter, and use it when it is called

Deep binding with **static scoping**: an example in Pascal

```
program binding_example(input, output);

procedure A(I : integer; procedure P);

    procedure B;
    begin
        writeln(I);
    end;

begin (* A *)
    if I > 1 then
        P
    else
        A(2, B);
end;

procedure C; begin end;

begin (* main *)
    A(1, C);
end.
```

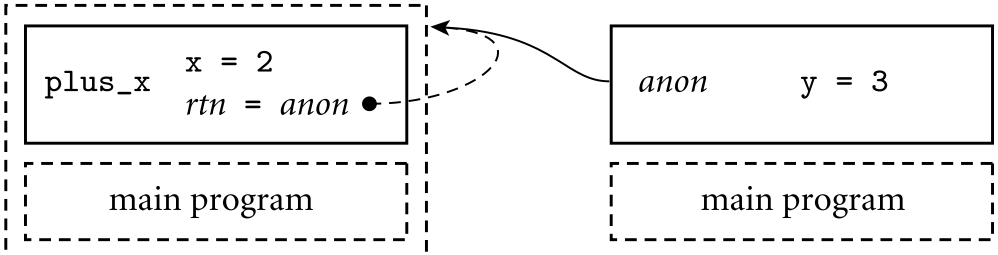


When B is called via formal parameter P, two instances of I exist. Because the closure for P was created in the initial invocation of A, B's static link (solid arrow) points to the frame of that earlier invocation. B uses that invocation's instance of I in its writeln statement, and the output is a 1. With **shallow binding** it would print 2.

Returning subroutines

- In languages with first-class subroutines, a function **f** may declare a subroutine **g**, returning it as result
- Subroutine **g** may have non-local references to local objects of **f**. Therefore:
  - **g** has to be returned as a *closure*
  - the activation record of **f** cannot be deallocated

```
(define plus-x (lambda (x)
    (lambda (y) (+ x y))))
...
(let ((f (plus-x 2)))
    (f 3))        ; returns 5
```



- **`(plus-x 2)`** returns an **anonymous function** which refers to the local **x**

First-Class Subroutine Implementations

- In functional languages, local objects have *unlimited extent*: their lifetime continue indefinitely
  - Local objects are allocated on the heap
  - *Garbage collection* will eventually remove unused objects
- In imperative languages, local objects have *limited extent* with stack allocation
- To avoid the problem of dangling references, alternative mechanisms are used:
  - C, C++, and Java: no nested subroutine scopes
  - Modula-2: only outermost routines are first-class
  - Ada 95 "containment rule": can return an inner subroutine under certain conditions

Object closures

- Closures (i.e. subroutine + non-local enviroment) are needed only when subroutines can be nested
- Object-oriented languages without nested subroutines can use objects to implement a form of closure
  - a method plays the role of the subroutine
  - instance variables provide the non-local environment
- Objects playing the role of a function + non-local enviroment are called **object closures** or **function objects**
- Ad-hoc syntax in some languages
  - In C++ an object of a class that overrides **operator()** can be called with functional syntax

Object closures in Java and C++

```
interface IntFunc {                              //Java
       public int call(int i);
}
class PlusX implements IntFunc {
       final int x;
       PlusX(int n) { x = n; }
       public int call(int i) { return i + x; }
}
...
IntFunc f = new PlusX(2);
System.out.println(f.call(3));      // prints 5
```

```
class int_func {                             // C++
   public:
       virtual int operator()(int i) = 0;
};
class plus_x : public int_func {
       const int x;
   public:
       plus_x(int n) : x(n) { }
       virtual int operator()(int i) { return i + x; }
};
...
plus_x f(2);          // f is an instance of plus_x
cout << f(3) << "\n";                 // prints 5
```