Subroutines and Control Abstraction

Control abstraction is association of a name with a program fragment that performs an operation, and is thought of its purpose or function, rather than its implementation.

Subroutine is a principal mechanism for control abstraction.

Procedure is a subroutine that does not return a value.

Function is a subroutine that returns a value.

In-Line Expansion

Recall the overheads of stack-based calling, i.e. space allocation, branch delays from call and return, maintaining link to referenced stack frame, saving/restoring registers.

As an alternative, many language implementations (e.g. C, C++, Ada) allow certain subroutines to be **expanded in-line** at the point of call.

- A copy of the called routine becomes a part of the caller (no actual subroutine call)

```
//C++
inline int max (int a, int b) {
  return a > b ? a : b;
}
…
a = max(x,y);
  is expanded to
a = x > y ? x : y;
```

- Code size will increase as body of subroutine appears at every call site within the caller.
- It is a hint and the compiler may ignore it.
- Not an option in general case for recursive subroutines, or only expand one level at each call site.

Parameter Passing

```
int max (int a, int b)
{ … }
…
x = max(y+5, 10);
```

Parameter names that appear in the declaration of a subroutine are **formal parameters**.

Expressions that are passed to a subroutine in a particular call are **actual parameters** or **arguments**.

Parameter Passing Modes

We will talk about three common modes.

For a language with a **value model of variables**

- **Call by value**
- **Call by reference**

For a language with a **reference model of variables**

- **Call by sharing**

Common Parameter Passing Modes for Value Model of Variables

Call by value

Each actual parameter is assigned into the corresponding formal parameter. The two are independent. (A copy of actual parameter’s value is passed).

Call by reference

Each formal parameter is a new name for the corresponding actual parameter. The two refer to the same object. (The l-value of the actual parameter is passed.)

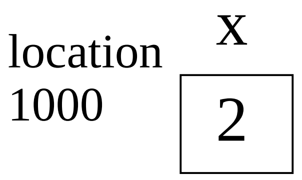
Some languages provide only one mode (e.g. C - value, Fortran - reference). Some provide both (e.g. Pascal, Ada, C++).

Call by Value (for Value Model of Variables)

Value of an actual parameter is assigned to a formal parameter. Both parameters are independent.

```
(* Pascal *)
var x : integer;        (* global *)

procedure foo(y : integer)
   y := 3;
   println(x);     (* print 2 *)
end;
…
x := 2;
foo(x);            (* Value 2 is passed. *)
println(x);        (* print 2 *)
```



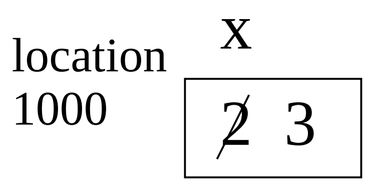
Call by Reference (for Value Model of Variables)

Formal parameter is a new name for actual parameter; the two are the same object.

If change is made on the object through the formal parameter within the subroutine, the change is visible through the actual parameter outside the subroutine.

```
(* Pascal *)
var x : integer;      (* global *)

procedure foo(var y : integer)
  y := 3;
  println(x);    (* print 3 *)
end;
…
x := 2;
foo(x);          (* Location 1000 is passed. *)
println(x);      (* print 3 *)
```



Variations in Languages with Value Model of Variables (1)

Java

- Primitive types use call by value.

C

- Always calls by value.
- Except for passing an array because of compatibility between array and pointer.
- Changes to array elements accessed through this pointer within the subroutine are visible to the caller.

```
//C
//Array a is like a pointer to the first element of array
int a[3];
…
init(a);
```

Exercise: Call by Value

Draw a picture to show the values of all variables. What does the program print at the end?

```
//C
void swap(int a, int b) {
int t = a; a = b; b = t;
}
…
int v1 = 1, v2 = 2;
swap(v1, v2);
printf("v1 is %d and v2 is %d\n", v1, v2);
```

……………………………………………………………………

Variations in Languages with Value Model of Variables (2)

C

- To **simulate** an effect of call by reference, a **pointer type** whose value is a reference to some object is used.
- The address of the variable, rather than the variable itself, must be passed explicitly as an argument to the subroutine.
  - *Actual parameter is the address of an object.*
  - *Formal parameter is a pointer which is dereferenced to modify the object.*

```
//C
void swap(int *a, int *b) {                       //pointers a,b hold address of v1,v2
int t = *a; *a = *b; *b = t;                      //modify v1,v2 values through a,b
                                                  //by using dereference operator (*)
…
int v1 = 1, v2 = 2;
swap(&v1, &v2);                                   //addrs of args are passed explicitly
printf("v1 is %d and v2 is %d\n", v1, v2);        //v1 is 2, v2 is 1
```

Variations in Languages with Value Model of Variables (3)

C++

- C++ addresses C's lack of true call by reference by using the concept of **reference variables (alias)** for its style of call by reference.
- A formal parameter is a reference parameter; it contains the address of the actual parameter, but there is no semantic difference between the two (exact same operations can be applied).

```
//C++
void swap(int &a, int &b) {
int t = a; a = b; b = t;        //a, b are int (alias of v1, v2)
}                               //not pointer to int; no dereferencing is required
…
int v1 = 1, v2 = 2;
swap(v1, v2);
cout << "v1 is " << v1 << " and v2 is " << v2 << "\n";    //v1 is 2, v2 is 1
```

When to Use Which Passing Mode

When the language provides both call by value and call by reference,

- Use call by reference when actual parameters need to be modified, or are very large.
- But call by reference requires an extra level of indirection. If the parameter is used often within the subroutine, the cost of indirection may outweigh the cost of copying the value.

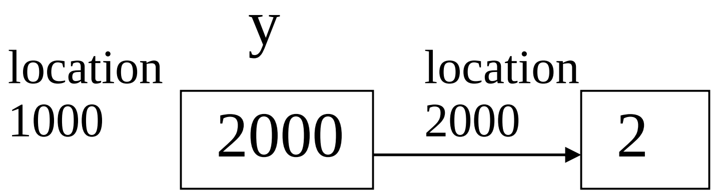
Common Parameter Passing Modes for Reference Model of Variables

Such as Smalltalk, Lisp, ML, Clu, Python, Ruby, Java’s user-defined class types.

Call by sharing

- Since an actual parameter is already a reference to an object, it is copied to the formal parameter (e.g. passing location 2000).

```
y = 2
```



- The actual parameter and formal parameter refer to (or share) the same object.
- Many people mix it up with call by value and call by reference since the value that is passed is a reference to an object.

Call by Sharing (for Reference Model of Variables) (1)

Similar to call by value, actual parameter is copied into formal parameter. But both of them are references, not values of objects.

```
//Java
//Suppose Class IntWrapper has an int attribute and methods getValue() and
setValue()

static void swap(IntWrapper x, IntWrapper y) {
  IntWrapper tmp = x;
  x = y;
  y = tmp;
}
…
IntWrapper v1 = new IntWrapper(1);
IntWrapper v2 = new IntWrapper(2);
swap(v1, v2);                           //v1 and v2 are references
System.out.println("v1 is " + v1.getValue() + " and v2 is " + v2.getValue() + "\n");
//v1 is 1, v2 is 2
```

Exercise: Call by Sharing vs. Call by Value

Given the code in the previous slide which looks similar to the swap() example of call by value, except that the variables are references not values. Draw a picture to show the values of all variables. How is it similar to and different from call by value?

```
//Java
static void swap(IntWrapper x, IntWrapper y) {
  IntWrapper tmp = x;
  x = y;
  y = tmp;
}
…
IntWrapper v1 = new IntWrapper(1);
IntWrapper v2 = new IntWrapper(2);
swap(v1, v2);
System.out.println(“v1 is ” + v1.getValue() + “ and v2 is ” + v2.getValue() + “\n”);
//v1 is 1, v2 is 2
```

Call by Sharing (for Reference Model of Variables) (2)

Similar to call by reference, if the object to which the formal parameter refers is modified, such change is visible through the actual parameter outside the subroutine.

```
//Java
//Suppose the class IntWrapper has an int attribute and methods getValue()and
setValue()

static void swap(IntWrapper x, IntWrapper y) {
  int tmp = x.getValue();
  x.setValue(y.getValue());
  y.setValue(tmp);
}
…
IntWrapper v1 = new IntWrapper(1);
IntWrapper v2 = new IntWrapper(2);
swap(v1, v2);                               //v1 and v2 are references
System.out.println(“v1 is ” + v1.getValue() + “ and v2 is ” + v2.getValue() + “\n”);
//v1 is 2, v2 is 1
```

Exercise: Call by Sharing vs. Call by Reference

Given the code in the previous slide. Draw a picture to show the values of all variables. How is it similar to call by reference?

```
//Java
static void swap(IntWrapper x, IntWrapper y) {
  int tmp = x.getValue();
  x.setValue(y.getValue());
  y.setValue(tmp);
}
…
IntWrapper v1 = new IntWrapper(1);
IntWrapper v2 = new IntWrapper(2);
swap(v1, v2);
System.out.println(“v1 is ” + v1.getValue() + “ and v2 is ” + v2.getValue() + “\n”);
//v1 is 2, v2 is 1
```

Call by Sharing (for Reference Model of Variables) (3)

Unlike call by reference, although the subroutine can change the value of the object to which the actual parameter refers, it cannot change the location to which the actual parameter refers (actual parameter cannot be changed to refer to a different object).

This is possible for call by reference.

```
//C++
void passByValue(int* p) {          //p points to the same location as p1;
  p = new int;                      //p now points to a new int value
}
void passByReference(int* &p) {     //p is a reference to p2
  p = new int;                      //p now points to a new int value
}
…
int main() {
  int* p1 = 0;                      //p1 is NULL
  int* p2 = 0;                      //p2 is NULL
  passByValue(p1);
  cout << p1 << "\n";               //0;  p1 is still NULL
  passByReference(p2);
  cout << p2 << "\n";               //0x582c58;  p2 now points to some allocated area for a new int value
}
```

Parametric Polymorphism

Dynamically typed languages can naturally support parametric polymorphism, e.g.

```
-- pseudocode
function calculate(a, b, c) => return (a + b)*c

example1 = calculate (1, 2, 3)
example2 = calculate ([1], [2, 3], 2)
example3 = calculate (‘apples ’, ‘and oranges, ’, 3)

print to_string example1      -- 9
print to_string example2      -- [1, 2, 3, 1, 2, 3]
print to_string example3      -- apples and oranges, apples and oranges, apples and oranges,
```

Statically typed languages use **explicit** parametric polymorphism (**generics**) to specify type parameters when declaring a subroutine or class.

- A **generic subroutine** can perform operation for different object types.
- A **generic class** is useful for creating containers (data abstractions that hold a collection of objects but whose operations do not care about the type of those objects).
- E.g. Ada, C++, Eiffel, Java, C#, Scala

Compiler can use these type parameters in static type checking.

Generics

//C++ Template

```
template<class item, int max_items = 100>
class queue {
    item items[max_items];
    int next_free, next_full, num_items;
public:
    queue() : next_free(0), next_full(0), num_items(0) { }
    bool enqueue(const item& it) {
        if (num_items == max_items) return false;
        ++num_items;  items[next_free] = it;
        next_free = (next_free + 1) % max_items;
        return true;
    }
    bool dequeue(item* it) {
        if (num_items == 0) return false;
        --num_items;  *it = items[next_full];
        next_full = (next_full + 1) % max_items;
        return true;
    }
};
...
queue<process> ready_list;
queue<int, 50> int_queue;
```

Compiler creates a separate copy of queue abstraction for every queue instance.

If several queue instances share the same set of type arguments, they share the same copy of code.

Exception

Exception is unexpected or unusual condition that arises during program execution.

It may be detected automatically by language implementation, or program may raise it explicitly, e.g.

- Unexpected end of file when reading input
- Reading in a string when expecting a number
- Arithmetic overflow
- Division by zero
- Subscript error
- Null pointer dereference

Exception handling facility is provided by more recent languages, e.g. Ada, C++, Java, C#, ML, Python, PHP, Ruby.

Defining Exception

In most OO languages, an exception is an instance of some predefined or user-defined class type.

An exception can be parameterized to pass information to the handler. In OO languages, parameters are the attributes of the exception class.

```
//C++
class duplicate_in_set {
public:
  duplicate_in_set(item ditem) {dup = ditem;}
  item dup;                      //element that was inserted twice
};



…
throw duplicate_in_set(d);    //raise exception, passing duplicate item d
```

Exception Handlers

Handlers are lexically bound to blocks of code.

Execution of a handler replaces the yet-to-be-completed portion of the block:

- Compensate to allow program to recover and continue, e.g. request for more space for out-of-memory exception.
- Clean up, e.g. resources in local block, and reraise exception to propagate back to a handler that can recover.
- Print helpful message, if recovery is not possible, and terminate program.

```
//C++
try {
  …
  if (something_unexpected)
    throw my_error();
  …
  cout << "everything's ok\n";
  …
} catch (my_error) {
    cout << "oops!\n";
}
```

```
//C++
try {
  …
  if (something_unexpected)
    throw my_error("oops!");
  …
  cout << "everything's ok\n";
  …
} catch (my_error e) {
    cout << e.explanation << "\n";
}
```

Exception Propagation out of a Called Subroutine

If an exception is not handled within the current subroutine, the subroutine returns abruptly and exception is raised at the point of call.

If the exception is not handled in the calling routine, it propagates back up the calling chain.

If the exception is not handled in the main program, a predefined outermost handler is invoked, and usually terminates the program.

```
//C++
try {
  …
  foo();
  …
  cout << “everything’s ok\n”;
  …
} catch (my_error e) {
    cout << e.explanation << “\n”;
}
```

```
//C++
void foo() {
  …
  if (something_unexpected)
    throw my_error(“oops!”);
  …
}
```

Exception Propagation

```
//C++
try {
    …
    //protected block of code that tries to read a file and involves many calls to stream I/O routines
    …
} catch (end_of_file e1) {
    …   //handler for end_of_file
} catch (io_error e2) {
    …   //handler for any io_error other than end_of_file
} catch (…) {
        //handler for any exception not previously named
        //… in this case is a valid C++ catch-all token
}
```

A block of code may have a list of exception handlers, and they are examined in order.

The handler matches the raised exception if it names a class from which the exception is derived, or if it is a catch-all, e.g.

- end_of_file exception will match the first handler (end_of_file is subclass of io_error).
- Other I/O errors will match the second.
- Non-I/O errors will match the third.

Exception Table

Each entry in an exception table corresponds to a protected block, and contains

- Starting and ending addresses of the protected block.
- Corresponding handlers addresses

Entries are sorted by starting address.

When an exception is raised, program counter is used as a key for search on the table, e.g. binary search, to find the handler for the current block.

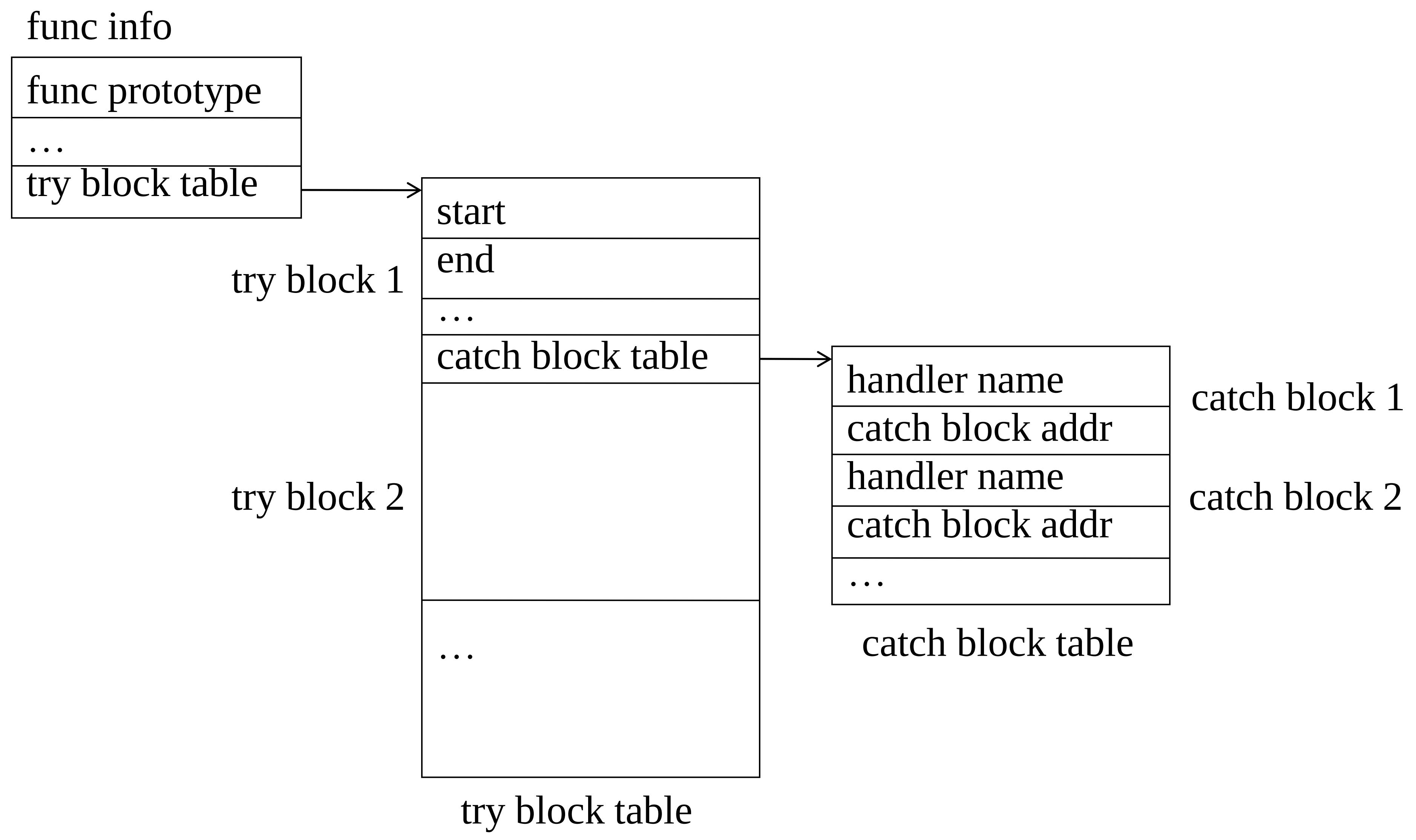
That handler checks if it matches the exception.

- If not, it performs the subroutine epilogue code and reraises the exception.
- The return address of the subroutine is used as a key for table lookup, then making the exception propagates back down the calling chain.

High run-time cost (for table lookup) especially when there are a lot of handlers in the program. But this cost is paid only when an exception occurs (which is an unusual event).

Table creation is compile-time cost.

Example of Exception Table Implementation


func info
func prototype
…
try block table
start
end
…
catch block table
try block 1
try block 2
…
try block table
handler name
catch block addr
handler name
catch block addr
…
catch block 1
catch block 2
catch block table