# CS4212 - Compiler Design

# Exceptions and OOL

# Outline

- Exceptions
  - Checked
  - Unchecked
  - `finally` clauses
- Object Oriented Languages
  - Polymorphism
  - Inheritance

# General Facts about Exceptions

- Useful for error handling
- Two parts:
  - `try` statement with `catch`/`finally` clauses
  - `throw`/`raise` statement: execution jumps to the `catch` clause that can handle the exception
- Without function calls: similar to a labeled `break`
- With function calls: non-local returns

# Exception Example

[C]

```
try {
   …
   throw exc ;
   …
} catch (Exc1 e) {
   …
} catch (Exc2 e) {
   …
} finally {
   …
}
```

```
{ struct Exc { exc_type t ; … } E ;
  E.t = no_exception ;
  …
  E.t = exc_type ;
  // Fill E with more relevant data
  goto catch_clauses ;
  …
catch_clauses:
  switch ( E.t ) {
  case exc1_type:
    // catch Exc1 code
    goto finally_clause ;

  case exc2_type:
    // catch Exc2 code
    goto finally_clause ; // may be optimized

  case no_exception:
  finally_clause:
    // finally code
    break ;
  }
}
```

# Function Calls

- Java distinguishes between checked and unchecked (runtime)
- Checked Exceptions
  - return individually from each called function
  - return structure contains possibly thrown exception
  - the caller must check return structure
    - no exception: proceed as usual
    - exception raised: re-raise
      - fill up return structure and return immediately

# Function Calls

- Unchecked exceptions
  - `try` statement saves processor "state", including the *current stack configuration*
  - Keep a global variable whose type is a structure to record the currently thrown exception
  - `throw` statement
    - fills up global variable with exception details
    - restores "state", performing a long return into the try statement
  - global variable helps select correct `catch` clause

# Unchecked Exception - Refinement



- `try` statements may be nested
- saved state encoded as element to be placed on *exception stack*
- each `try` pushes current state on exception stack
- `throw`: restores state at top of exception stack
- if restored `try` cannot handle current exception, re-`throw`
- before re-throwing, execute `finally`, if one exists

# Example

```
int f() {
  …
  try {
    …
    g() ;
    …
  } catch (E1 e) {
    …
  }
  …
  return … ;
}

int g() {
  …
  h() ;
  …
  return … ;
}
```

```
int h() {
  …
  try {
    …
    E1 e1 ;
    …
    throw e1 ;
    …
    E2 e2 ;
    …
    throw e2 ;
    …
  } catch (E2 e2) {
    …
  } finally {
    …
  }
  …
  return … ;
}
```

# Checked Exc: Translation of f in C

```
int f() {
  …
  try {
    …
    g() ;
    …
  } catch (E1 e) {
    …
  }
  …
  return R ;
}
```

```
struct E { exc_type T ; exc_val V ;
           int retVal ; } ;

struct E f() {
  …
  { struct E gRet ;
    …
    gRet = g() ;
    switch ( gRet.T ) {
    case E1 :
      … // handle exception E1
      goto finally ;
    case NOEXCEPTION :
    finally :
      … // finally block, if any, otherwise empty
      break ;
    default : // other exception, rethrow
      return gRet ;
  }
  …
  struct fRet ;
  fRet.T = NOEXCEPTION ;
  fRet.retVal = R ;
  return fRet ;
}
```

# Checked Exc: Translation of g in C

```
int g() {
  …
  h() ;
  …
  return R ;
}
```

```
struct E g() {
  …
  struct E hRet ;
  hRet = h() ;
  if ( hRet.T != NOEXCEPTION )
    return hRet ;
  …
  struct E gRet ;
  gRet.T = NOEXCEPTION ;
  gRet.RetVal = R ;
  return gRet ;
}
```

# Checked Exc: Translation of h in C [C]

```
int h() {
  …
  try {
    …
    E1 e1 ;
    …
    throw e1 ;
    …
    E2 e2 ;
    …
    throw e2 ;
    …
  } catch (E2 e2) {
    …
  } finally {
    …
  }
  …
  return R ;
}
```

```
struct E h() {
  …
  { struct E hRet ;
    …
    hRet.T = E1 ; hRet.V = …  ; goto catch ;
    …
    hRet.T = E2 ; hRet.V = … ; goto catch ;
    …
  catch:
    switch ( hRet.T ) {
    case E2 :
        … // handle E2
        hRet.T = NOEXCEPTION ;
        goto finally :
    default:
    finally:
        …
        if ( hRet.T != NOEXCEPTION ) return hRet ;
        break ;
    }
  …
  hRet.T = NOEXCEPTION ;
  hRet.RetVal = R ;
  return hRet ;
}
```

# `setjmp/longjmp` in C

- `int setjmp(jmp_buf env)`
  - Sets up the local `jmp_buf` buffer and initializes it for the jump.
  - Saves the program's calling environment in the environment buffer `env`.
  - Direct invocation: `setjmp` returns 0.
  - Return from call to `longjmp`: returns nonzero.
- `void longjmp(jmp_buf env, int value)`
  - Restores context of environment buffer `env`.
  - The value specified by value is passed from `longjmp` to `setjmp`.
  - Program execution continues as if the corresponding invocation of `setjmp` had just returned.
  - `value != 0` -> `setjmp` returns 1 ; otherwise returns value.

# Example

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");         // prints
    longjmp(buf,1);             // jumps back to where setjmp was called - 
making setjmp now return 1
}

void first(void) {
    second();
    printf("first\n");          // does not print
}

int main() {
    if ( ! setjmp(buf) ) {
        first();                // when executed, setjmp returns 0
    } else {                    // when longjmp jumps back, setjmp returns 1
        printf("main\n");       // prints
    }

    return 0;
}
```

# Unchecked Exceptions: f

```
int f() {
  …
  try {
    …
    g() ;
    …
  } catch (E1 e) {
    …
  }
  …
  return R ;
}
```

g remains the same!

```
extern struct E exception ;
extern jmp_buf push(), pop() ;

int f() {
  …
  if ( setjmp(push()) ) { // try
    …
    g() ;
    …
    pop();
  } else {
     switch ( exception.T ) {
        case E1 :
          …
          exception.T = NOEXCEPTION ;
          goto finally ;
        default:
        finally:
          … // may be empty
          if ( exception.T != NOEXCEPTION )
            longjmp(pop()) ;
     }
  }
  …
  return R ;
}
```

# Unchecked Exceptions: h

[A]

```
int h() {
  …
  try {
    …
    E1 e1 ;
    …
    throw e1 ;
    …
    E2 e2 ;
    …
    throw e2 ;
    …
  } catch (E2 e2) {
    …
  } finally {
    …
  }
  …
  return R ;
}
```

```
int h() {
  …
  if (setjmp(push)) {
    …
    exception.T = E1 ;
    exception.V = e1 ;
    longjmp(pop()) ;
    …
    exception.T = E2 ;
    exception.V = e2 ;
    longjmp(pop()) ;
    …
    pop();
  } else {
      switch ( exception.T ) {
         case E2 : // handle E2
           …
           exception.T = NOEXCEPTION ;
           goto finally ;
         default:
         finally:
           …
           if ( exception.T != NOEXCEPTION )
             longjmp(pop()) ;
      }
  }
  …
  return R ;
}
```

# Discussion

- Checked exceptions
  - easier to optimize, more compositional
  - more amenable to debugging
  - less flexible: require intermediate functions (like g) to be aware that an exception may pass through
- Unchecked exceptions
  - more flexible: intermediate functions need not changed
  - less amenable to debugging
  - more difficult to optimize

# Object Oriented Languages

- Encapsulation and Data Hiding
  - static checking of access attributes
  - name mangling
- Polymorphism
  - type checking
  - name mangling
- Inheritance
  - tables of pointers to functions

# Encapsulation and Data Hiding

- Each class translated into a structure
  - data fields are copied into the structure
  - for each method reserve a pointer to that method
- Acces modifiers become part of type system
- Data hiding enforced through static checking

# Scoping

- Global scope: heap
- Class scope: heap
- Object scope: structure
- Method scope: stack
- Local scope: stack
- Care must be taken in stacking up frames in correct order

# Data Access and Polymorphism

- class scopes, return and argument types are used to mangle the name of a static data member, or of a method.
- Ex: `int class1::f(char)` translated into

  `int class1_f_ic(char)`
- Global namespace contains all static references
  - Name mangling ensures that each reference is unique
  - No name clashes should be possible

# An OO Example

```
class Speaker {
  void say(String msg) { System.out.println(msg) ; }
}

class Lecturer extends Speaker {
  void lecture(String msg) {
    say(msg) ;
    say("You should be taking notes") ;
  }
}

class ArrogantLecturer extends Lecturer {
  void say(String msg) { super.say("It is obvious that" + msg) ; }
}

…

(new ArrogantLecturer).lecture("The sky is blue") ;
```

# Speaker: C Equivalent

```
struct speaker {
       void (*say)(struct speaker * self, char* msg) ;
} ;

void speaker_say(struct speaker * self, char* msg) {
     printf("%s\n",msg) ;
}

void init_speaker(struct speaker *p) {
       p->say = speaker_say ;
}

struct speaker * make_speaker () {
       struct speaker * retVal = malloc(sizeof(struct speaker));
       init_speaker(retVal) ;
       return retVal ;
}
```

# Lecturer: C Equivalent

```
struct lecturer {
       void (*say)(struct speaker * self, char* msg) ;
       void (*lecture)(struct lecturer * self, char* msg) ;
} ;

void lecturer_lecture(struct lecturer * self, char * msg){
     self->say(self,msg) ;
     self->say(self,"You should be taking notes") ;
}

void init_lecturer(struct lecturer *p) {
       init_speaker(p) ;
       p->lecture = lecturer_lecture ;
}

struct lecturer * make_lecturer() {
       struct lecturer * retVal = malloc(sizeof(struct lecturer));
       init_lecturer(retVal) ;
       return retVal ;
}
```

# Arrogant Lecturer: C Equivalent

```
struct alecturer {
       void (*say)(struct speaker * self, char* msg) ;
       void (*lecture)(struct lecturer * self, char* msg) ;
       void (*super_say)(struct speaker * self, char* msg) ;
};

void alecturer_say(struct alecturer * self, char * msg){
     char * p = malloc(200) ;
     *p = '\0' ;
     strcat(p,"It is obvious that " ) ;
     strcat(p,msg) ;
     self->super_say(self,p) ;
}

void init_alecturer(struct alecturer *p) {
       init_lecturer(p) ;
       p->super_say = p->say ;
       p->say = alecturer_say ;
}

struct alecturer * make_alecturer() {
       struct alecturer * retVal = malloc(sizeof(struct alecturer));
       init_alecturer(retVal) ;
       return retVal ;
}
```

# Main function

```
int main() {
    struct alecturer * p = make_alecturer() ;
    p->lecture(p,"The sky is blue") ;
}
```

# Discussion

- Each class
  - compiled into a structure
  - data members remain the same
  - methods
    - pointer to function member
    - translated into function
- Each method
  - compiled into a function with a mangled name
  - first argument is pointer to self
  - always called via the corresponding pointer

# Constructors

- Split into two parts
  - allocation
  - intialization
    - call initializer of extended class
- Return a pointer to the structure representing the object
  - method calls have the pattern

```
p->m(p,...) ;
```

# Class Extension

- Corresponding structure has same prefix as extended class
  - allows polymorphism
- overloaded methods of superclass are saved into separate fields of the structure
  - they can still be called if needed

# Optimizations

- Pointers to functions
  - class wide table instead of object memebership
- Detect methods that are never overloaded and compile direct calls for them