

## COMP205 IMPERATIVE LANGUAGES

### 12a. ERROR HANDLING

- 1) Causes of errors
- 2) Classification of errors
- 3) Signals and exceptions

### 12b. PROGRAM COMPOSITION I

- 1) Program hierarchy
- 2) Blocks
- 3) Routines
- 4) Procedures and functions

## CAUSES OF ERRORS

- 1) **Domain/data errors**: an operation is called with data that cannot be handled by that operation, e.g. *divide by zero, integer overflow*, etc.
- 2) **Resource exhaustion**: memory errors. If not enough memory is available:
  - The Ada allocator `new` will fail.
  - The C allocator `malloc` will return `NULL`.
- 3) **Loss of facilities**: network failure, power failure, etc.

## ADA DIVIDE BY ZERO EXAMPLE

```
procedure MYADAPRO is
  P: integer:= 2;
  Q, R: integer:= 0;
begin
  R := P/Q;
end MYADAPRO;
```

```
$ myAdaPro
Ada-runtime: Exception NUMERIC_ERROR
  raised in error1.ada on line 5.
```

## C DIVIDE BY ZERO EXAMPLE

```
void main(void) {
  int p = 2, q = 0, r = 0;

  r = p/q;
}
```

```
$ myCpro
Floating exception(coredump)
```

## SIGNALS AND EXCEPTIONS

- Errors usually result in the termination of a program, i.e. it "*crashes*".
- Such an uncontrolled termination is undesirable.
- Most programming languages leave the onus of dealing with errors with the programmer.
- There are two mechanisms whereby a program can regain control after an error has been detected.
  1. Signal statements.
  2. Exception handlers.

## SIGNALS

- A **signal statement** names an error condition and a procedure. When the error condition occurs the procedure is called.
- An **error condition** is a class of error, e.g. `NUMERIC_ERROR` (Ada).
- Signal statements can be implemented as *library routines*.
- Disadvantage of signal statements is that they have no access to local variables.

## EXCEPTION HANDLER

- An **exception handler** comprises an error condition and a set of statements that can be attached to a block of code.
- If a statement in the block fails with the error condition specified in the exception handler, the set of statements in the exception handler will be executed instead of the remaining statements in the block.
- Ada has five predefined error conditions:
 

constraint_error	numeric_error
program_error	storage_error
tasking_error	
- Note: Ada also allows user-defined error conditions.  
Example:

```
TIME_UP: exception;
```

## EXAMPLE ADA EXCEPTION HANDLER

```
procedure MYADAPRO is
  P: integer:= 2;
  Q, R: integer:= 0;
begin
  R := P/Q;
exception
  when NUMERIC_ERROR =>
    put_line("NUMERIC_ERROR");
    put("Exception handler ");
    put_line("has control");
    put("P = ");put(P);new_line;
    put("Q = ");put(Q);new_line;
    put("R = ");put(R);n($ myAdaPro
end MYADAPRO;
```

```
NUMERIC_ERROR
Exception handler has control
P =      2
Q =      0
R =      0
```

## SUMMARY: ERROR HANDLING

- 1) Causes of errors
- 2) Classification of errors
- 3) Signals and exceptions

## 12(b) PROGRAM COMPOSITION I

- 1) Program hierarchy
- 2) Blocks
- 3) Routines
- 4) Procedures and functions

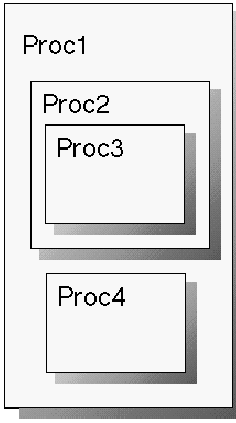
## PROGRAM HIERARCHY

- Four levels of hierarchy can be distinguished:
  - 1) Blocks.
  - 2) Routines (sub-programs).
  - 3) Modules and packages.
  - 4) Programs.
- Two other ingredients are abstract data types and generics.

## BLOCKS

- A **block** is the smallest grouping of declarations and statements.
  - In Ada it is delimited by the keywords `begin` and `end`.
  - In C using the symbols `{` and `}`.
- A block defines the **local scope** of its data items (usually variables), i.e. their "visibility".
- Variables within a block cannot be seen from outside that block.
- Such variables are referred to as **local variables**, local to a block.
- Most imperative programming languages will allow nesting of blocks (at least to some degree).

- When blocks are nested we conceive of **levels of nesting**.
- Blocks can “inherit” names declared in higher level blocks.



### ROUTINES (SUB-PROGRAMS)

- One of the most important issues that has to be addressed in the design of a programming language is the support it gives to the control of complexity.
- An important technique is the division of a program into "chunks" called routines (or sub-programs) that will allow the programmer to think at a more abstract level.
- A routine can be defined as a collection of statements and expressions that must be invoked explicitly.
- An additional advantage is that routines can be stored in libraries for reuse.
- Issues involved include:
  - 1) The method of combining routines to form a complete program.
  - 2) Parameter-passing mechanisms between routines

### PROCEDURES AND FUNCTIONS

- At its simplest a routine can be either a single:
  - 1) Procedure, or
  - 2) Function.
- The distinction between a procedure and a function is that a function returns a value while a procedure does not.
- A function must be called as part of an assignment expression.
 

```
destinationVariable =
  functionName (arguments)
```
- A function declaration must also incorporate a definition for its return value.

- Some imperative languages (Ada, FORTRAN, Pascal) make a clear distinction between procedures and functions.
- Other imperative languages (C, ALGOL 68) do not.
- The definition of a procedure (function) comprises a *head* and a *body*.
 

```
<HEAD> { <BODY> }
<HEAD> begin <BODY> end
```
- In the head the name of the routine and its *formal parameters* (arguments) are specified.
 

```
functionName( <ARGUMENTS> }
```
- In the body the necessary code is given.

- A procedure (function) is activated by a *procedure/function call* (routine invocation) which names the procedure (function) and supplies the *actual parameters*, i.e. the values which are to be assigned to the **formal parameters**.
- **Actual parameters** can be variables, constants or expressions.

### NESTING OF PROCEDURES AND FUNCTIONS

- Some imperative programming languages will allow nesting of procedures and functions.
- First introduced in Algol 60.
- A procedure/function is a **block** the same as any other.
- Languages that support the nesting of procedures/functions are sometimes referred to as *block structured languages*.
- Ada, Algol 60 and Pascal are block structured, C is not.
- Variables declared in the outermost procedure/function (block) are called **global variables** and have a **global scope**.

```

procedure EXAMPLE is
  X, Y: float;
  -----
  procedure MEAN_VALUE (X1, X2: float) is
  begin
    put("The mean is ");
    put((X1+X2)/2.0);
    new_line;
  end MEAN_VALUE;
  -----
begin
  put_line("Enter 2 real numbers ");
  get(X); get(Y);
  MEAN_VALUE(X,Y);
end EXAMPLE;

```

**Ada  
MEAN\_VALUE  
procedure**

```

procedure EXAMPLE is
  X, Y, Z: float;
  -----
  function MEAN_VALUE (X1, X2: float) return float is
  begin
    return (X1+X2)/2.0;
  end MEAN_VALUE;
  -----
begin
  put_line("Enter 2 real numbers ");
  get(X);
  get(Y);
  Z:= MEAN_VALUE(X,Y);
  put("The mean is ");
  put(Z);
  new_line;
end EXAMPLE;

```

**Ada  
MEAN\_VALUE  
function**

```

#include <stdio.h>

float meanValue(float, float);

void main(void) {
  float x, y;

  printf("Enter 2 real numbers ");
  scanf("%f%f",&x,&y);

  printf("\nThe mean is %f\n",meanValue(x,y));
}

float meanValue(float x1, float x2){
  return (x1+x2)/2.0;
}

```

**C  
meanValue  
function**

### ORDER OF PROCEDURE DECLARATIONS

- In **block structured languages** (where procedures and functions may be nested) the order in which procedures are declared also effects their visibility.
- Ada (and other block structured languages such as Pascal) require that any use of an identifier must be preceded by its declaration.
- Thus if two procedures are declared at the same level, their order of declaration will dictate "who can call whom".
- The procedure currently being declared cannot see any following procedures.
- This can be overcome by using an **incomplete declaration**.
- Ordering is not significant in C.

```

#include <stdio.h>

void meanValue(float, float);

void main(void) {
  float x, y;

  printf("Enter 2 real numbers ");
  scanf("%f%f",&x,&y);

  meanValue(x,y);
}

void meanValue(float x1, float x2) {
  printf("\nThe mean is %f\n", (x1+x2)/2.0);
}

```

**C  
meanValue  
procedure**

### SUMMARY: PROGRAM COMPOSITION I

- 1) Program hierarchy
- 2) Blocks
- 3) Routines
- 4) Procedures and functions