

Breaking Problems Down

This section of notes shows you how to break down a large problem into smaller modules that are easier to implement and manage.

James Tam

Designing A Program: Top-Down Approach

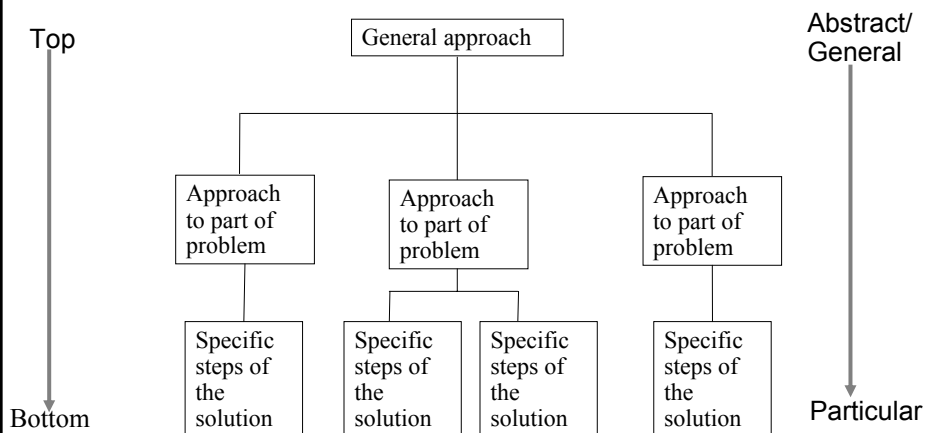
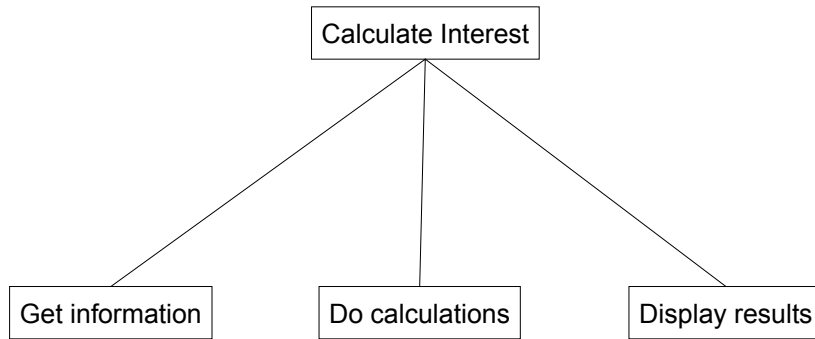


Figure extracted from Computer Science Illuminated by Dale N. and Lewis J.

James Tam

Top Down Approach: Programming



James Tam

Decomposing Problems Via The Top Down Approach

Approach

- Breaking problem into smaller, well defined parts (modules)
- Making modules as independent as possible (loose coupling)

Pascal implementation of program modules

- Procedures
- Functions

James Tam

Using Functions And Procedures In Pascal

Definition

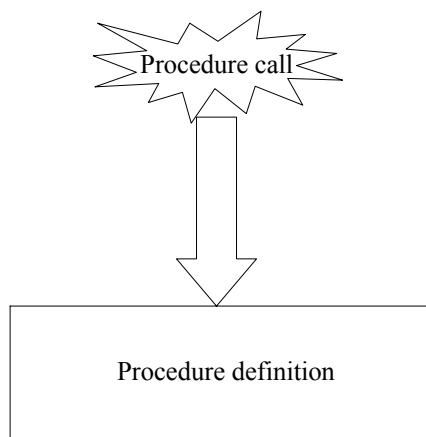
- Indicating what the function or procedure will do when it runs

Call

- Getting the function or procedure to run

James Tam

Procedures (Basic Case – No Parameters)



James Tam

Defining Procedures (Basic Case – No Parameters)

Format:

```
procedure name;  
begin  
    (* Statements of the procedure go here *)  
end; (* End of procedure name *)
```

Example:

```
procedure displayInstructions;  
begin  
    writeln ('The statements in this module will typically give a');  
    writeln ('high level overview of what the program as a');  
    writeln ('whole does');  
end; (* End of procedure displayInstructions *)
```

James Tam

Where To Define Modules (Procedures)

Header

Declarations

```
const  
Procedure and function definitions  
:
```

Statements

```
begin  
  
end.
```

James Tam

Calling A Procedure (Basic Case – No Parameters)

Format:

name;

Example:

displayInstructions;

James Tam

Where To Call Modules (Procedures)

It can be done most anywhere in the program

Header

Declarations

```
const  
Procedure and function definitions  
:
```

Statements

```
begin  
Calling the module: This example  
end.
```

James Tam

Procedures: Putting Together The Basic Case

The full version of this example can be found in
Unix under
/home/231/examples/modules/firstExampleProcedure.p

```
program firstExampleProcedure (output);  
  
procedure displayInstructions;  
begin  
    writeln ('The statements in this module will  
typically give a');  
    writeln ('high level overview of what the program  
as a');  
end;
```

James Tam

Procedures: Putting Together The Basic Case

The full version of this example can be found in Unix under
/home/231/examples/modules/firstExampleProcedure.p

```
program firstExampleProcedure (output);
```

```
procedure displayInstructions;  
begin  
    writeln ('The statements in this module will typically give a');  
    writeln ('high level overview of what the program as a');  
    writeln ('whole does');  
end; (*Procedure displayInstructions *)
```

```
begin  
    displayInstructions;  
    writeln('Thank you, come again!');  
end. (* Program *)
```

**Procedure
definition**

**Procedure
call**

James Tam

Defining Local Variables

Exist only for the life the module

Format:

```
procedure name;  
var  
  (* Local variable declarations go here *)  
begin  
  :  
end;
```

Example:

```
procedure proc;  
var  
  num1 : integer;  
begin  
  : :  
end;
```

James Tam

Defining Local Variables: Putting It All Together

The full version of this example can be found in Unix under
`/home/231/examples/modules/secondExampleProcedure.p`

```
program secondExampleProcedure (output);  
procedure proc;  
var  
  num1 : integer;  
begin  
  var num2 : integer;  
  num1 := 1;  
  num2 := 2;  
  writeln(num1, ' ', num2);  
end;  
begin  
  var num1 : integer;  
  num1 := 10;  
  writeln(num1);  
  proc;  
  writeln(num1);  
end.
```

James Tam

Defining Local Variables: Putting It All Together

The full version of this example can be found in Unix under
/home/231/examples/modules/secondExampleProcedure.p

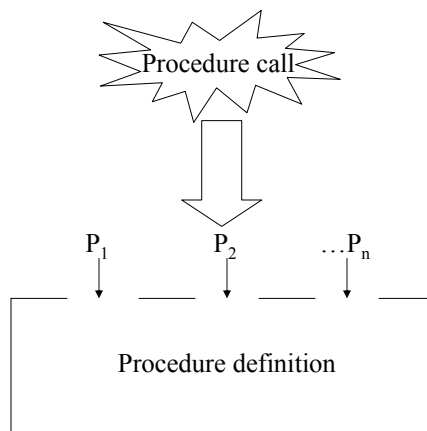
```
program secondExampleProcedure (output);  
procedure proc;  
var  
  num1 : integer;  
begin  
  var num2 : integer;  
  num1 := 1;  
  num2 := 2;  
  writeln(num1, ' ', num2);  
end;  
begin  
  var num1 : integer;  
  num1 := 10;  
  writeln(num1);  
  proc;  
  writeln(num1);  
end.
```

**Local variable:
procedure 'proc'**

**Local variable:
main module**

James Tam

Procedures With Parameters



James Tam

Defining Modules (Procedures) With Parameters

Format:

```
procedure name (Name of parameter 1 : type of parameter 1;  
               Name of parameter 2 : type of parameter 2;  
               :  
               :  
               Name of parameter n : type of parameter n);  
begin  
    (* Statements of the procedure go here *)  
end;
```

Example:

```
procedure celciusToFahrenheit (celciusValue : real);  
var  
    fahrenheitValue : real;  
begin  
    fahrenheitValue := 9 / 5 * celciusValue + 32;  
    writeln('temperature in Celsius: ', celciusValue:0:2);  
    writeln('temperature in Fahrenheit: ', fahrenheitValue:0:2);  
end; (* Procedure celciusToFahrenheit *)
```

James Tam

Calling Modules (Procedures) With Parameters

Format:

```
name (Name of parameter 1, Name of parameter 2...Name of  
      parameter n);
```

Example:

```
celciusToFahrenheit (celciusValue);
```

James Tam

Procedures: Putting Together The Case Of Procedures With Parameters

The full version of this example can be found in Unix under
/home/231/examples/modules/temperatureConverter.p

```
program temperatureConverter (input, output);

procedure celsiusToFahrenheit (celsiusValue : real);
var
  fahrenheitValue : real;
begin
  fahrenheitValue := 9 / 5 * celsiusValue + 32;
  writeln('Temperature in Celsius: ', celsiusValue:0:2);
  writeln('Temperature in Fahrenheit: ', fahrenheitValue:0:2);
end; (* Procedure celsiusToFahrenheit *)
```

James Tam

Procedures: Putting Together The Case Of Procedures With Parameters

The full version of this example can be found in Unix under
/home/231/examples/modules/temperatureConverter.p

```
program temperatureConverter (input, output);
```

```
procedure celsiusToFahrenheit (celsiusValue : real);
var
  fahrenheitValue : real;
begin
  fahrenheitValue := 9 / 5 * celsiusValue + 32;
  writeln('Temperature in Celsius: ', celsiusValue:0:2);
  writeln('Temperature in Fahrenheit: ', fahrenheitValue:0:2);
end; (* Procedure celsiusToFahrenheit *)
```

**Procedure
definition**



James Tam

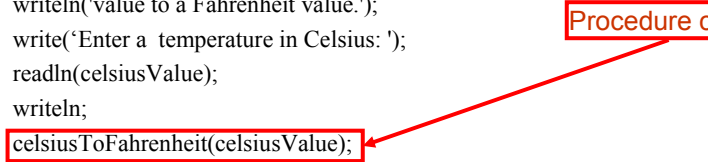
Procedures: Putting Together The Case Of Procedures With Parameters (2)

```
begin
  var celsiusValue : real;
  writeln;
  writeln('This program will convert a given temperature from a Celsius');
  writeln('value to a Fahrenheit value. ');
  write('Enter a temperature in Celsius: ');
  readln(celsiusValue);
  writeln;
  celsiusToFahrenheit(celsiusValue);
  writeln('Thank you and come again. ');
end. (* Program *)
```

James Tam

Procedures: Putting Together The Case Of Procedures With Parameters (2)

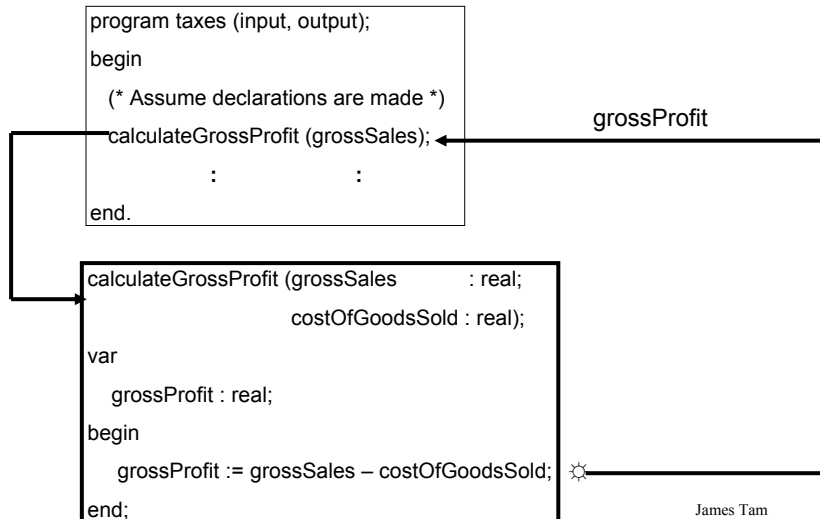
```
begin
  var celsiusValue : real;
  writeln;
  writeln('This program will convert a given temperature from a Celsius');
  writeln('value to a Fahrenheit value. ');
  write('Enter a temperature in Celsius: ');
  readln(celsiusValue);
  writeln;
  celsiusToFahrenheit(celsiusValue);
  writeln('Thank you and come again. ');
end. (* Program *)
```



James Tam

Retaining Information From A Module (Function Or Procedure) After The Module Has Ended

For example: producing an income statement

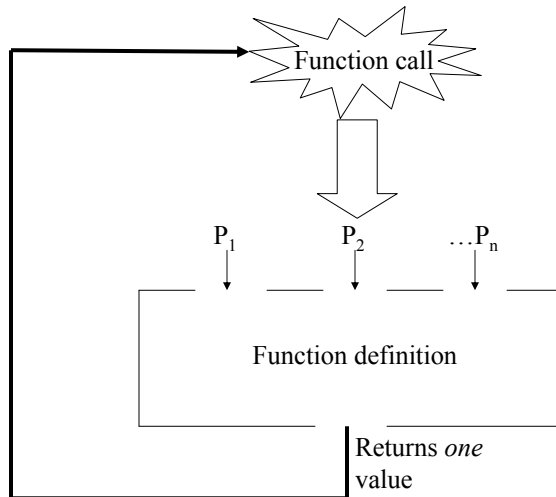


Retaining Information From A Module (Function Or Procedure) After The Module Has Ended (2)

Methods:

- **Return a value with a function**
- Pass parameters into the procedure as variable parameters (rather than as value parameters)

Functions



James Tam

Defining Functions

Format:

```
function name (Name of parameter 1 : type of parameter 1;  
              Name of parameter 2 : type of parameter 2;  
              :  
              :  
              Name of parameter n : type of parameter n):  
    return type;
```

begin

(* Statements of the function go here *)

```
    :  
    :  
    name := expression; (* Return value *)
```

end;

Example:

```
function calculateGrossIncome (grossSales      : real;  
                              costOfGoodsSold : real) : real;
```

begin

```
    calculateGrossIncome := grossSales - costOfGoodsSold;
```

end;

James Tam

Defining Functions

Format:

```
function name (Name of parameter 1 : type of parameter 1;  
              Name of parameter 2 : type of parameter 2;  
              :  
              :  
              Name of parameter n : type of parameter n):  
  return type;  
  
begin  
  (* Statements of the function go here *)  
  :  
  :  
  name := expression; (* Return value *)  
end;
```

Example:

```
function calculateGrossIncome (grossSales : real;  
                              costOfGoodsSold : real) : real;  
begin  
  calculateGrossIncome := grossSales - costOfGoodsSold;  
end;
```

**Return: Often the last
statement in the function**

James Tam

Calling Functions

Format:

```
variable := name of function;
```

```
variable := name of function (name of parameter 1, name of parameter  
                              2...name of parameter n);
```

Example:

```
grossIncome := calculateGrossIncome (grossSales, costOfGoodsSold);
```

James Tam

Functions: Putting It All Together

The full version of this example can be found in Unix under
/home/231/examples/modules/financialStatements.p

```
program financialStatments (input, output);

function calculateGrossIncome (grossSales      : real;
                              costOfGoodsSold : real) : real;

begin
  calculateGrossIncome := grossSales - costOfGoodsSold
end;

function calculateNetIncome (grossIncome : real;
                             expenses    : real) : real;

begin
  calculateNetIncome := grossIncome - expenses;
end;
```

James Tam

Functions: Putting It All Together

The full version of this example can be found in Unix under
/home/231/examples/modules/financialStatements.p

```
program financialStatments (input, output);
```

Function definitions

```
function calculateGrossIncome (grossSales      : real;
                              costOfGoodsSold : real) : real;

begin
  calculateGrossIncome := grossSales - costOfGoodsSold
end;
```

```
function calculateNetIncome (grossIncome : real;
                             expenses    : real) : real;

begin
  calculateNetIncome := grossIncome - expenses;
end;
```

James Tam

Functions: Putting It All Together (2)

```
procedure produceIncomeStatement;
var
  grossSales      : real;
  costOfGoodsSold : real;
  grossIncome     : real;
  expenses        : real;
  netIncome       : real;
begin
  write('Enter gross sales $');
  readln(grossSales);
  write('Enter cost of the goods that were sold $');
  readln(costOfGoodsSold);
  write('Enter corporate expenses $');
  readln(expenses);

  grossIncome := calculateGrossIncome (grossSales, costOfGoodsSold);

  netIncome := calculateNetIncome (grossIncome, expenses);
```

James Tam

Functions: Putting It All Together (2)

```
procedure produceIncomeStatement;
var
  grossSales      : real;
  costOfGoodsSold : real;
  grossIncome     : real;
  expenses        : real;
  netIncome       : real;
begin
  write('Enter gross sales $');
  readln(grossSales);
  write('Enter cost of the goods that were sold $');
  readln(costOfGoodsSold);
  write('Enter corporate expenses $');
  readln(expenses);

  grossIncome := calculateGrossIncome (grossSales, costOfGoodsSold);

  netIncome := calculateNetIncome (grossIncome, expenses);
```

Function calls

James Tam

Functions: Putting It All Together (3)

```
(* Procedure produceIncomeStatement continued *)  
writeln;  
writeln('Gross sales $':26, grossSales:0:2);  
writeln('Less: cost of goods sold $':26, costOfGoodsSold:0:2);  
writeln('Gross income $':26, grossIncome:0:2);  
writeln('Less: expenses $':26, expenses:0:2);  
writeln('Net income $':26, netIncome:0:2);  
writeln;  
end; (* End of procedure produceIncomeStatement *)
```

James Tam

Functions: Putting It All Together (4)

```
procedure intro;  
begin  
  writeln;  
  writeln('This program will produce an income statement based upon your');  
  writeln('gross sales figures, the cost of the goods that you sold and  
  writeln('your expenses.');  writeln;  
end;
```

James Tam

Functions: Putting It All Together (5)

```
(* Start of main program *)  
begin  
  intro;  
  produceIncomeStatement;  
  writeln("Thank you, come again!");  
end. (* End of entire program. *)
```

James Tam

Retaining Information From A Module (Function Or Procedure) After The Module Has Ended

Methods:

- Return a value with a function
- **Pass parameters into the procedure as variable parameters (rather than as value parameters)**

James Tam

Passing Parameters As Value Parameters

Previous examples

```
procedureName (p1: parameter type);
```

```
procedureName (p1 : parameter type);  
begin  
end;
```

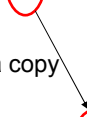
James Tam

Passing Parameters As Value Parameters

Previous examples

```
procedureName (p1 parameter type);
```

Pass a copy



```
procedureName (p1 : parameter type);  
begin  
end;
```

James Tam

Passing Parameters As Variable Parameters

Example coming up

```
procedureName (p1: parameter type);
```

```
procedureName (var p1 : parameter type);  
begin  
end;
```

James Tam

Passing Parameters As Variable Parameters

Example coming up

```
procedureName (p1 parameter type);
```

Pass a
variable

```
procedureName (var p1 : parameter type);  
begin  
end;
```

James Tam

Procedure Definitions When Passing Parameters As Variable Parameters

Format:

```
procedure name (var Name of parameter 1 : type of parameter 1;  
                var Name of parameter 2 : type of parameter 2;  
                :  
                :  
                var Name of parameter n : type of parameter n);  
  
begin  
    (* Statements of the procedure go here *)  
end;
```

Example:

```
procedure tabulateIncome (    grossSales      : real;  
                            costOfGoodsSold : real;  
                            var grossIncome  : real;  
                            expenses        : real;  
                            var netIncome    : real);  
  
begin  
    grossIncome := grossSales - costOfGoodsSold;  
    netIncome  := grossIncome - expenses;  
end;
```

James Tam

Calling Procedures With Variable Parameters

It's the same as calling procedures with value parameters!

Format:

```
name (name of parameter 1, name of parameter 2...name of parameter n);
```

Example:

```
tabulateIncome(grossSales,costOfGoodsSold,grossIncome,expenses,  
               netIncome);
```

James Tam

Passing Variable Parameters: Putting It All Together

The full version of this example can be found in Unix under
`/home/231/examples/modules/financialStatements2.p`

```
program financialStatments (input, output);

procedure getIncomeInformation (var grossSales      : real;
                               var costOfGoodsSold : real;
                               var expenses       : real);

begin
  write('Enter gross sales $');
  readln(grossSales);
  write('Enter the cost of the goods that were sold $');
  readln(costOfGoodsSold);
  write('Enter business expenses $');
  readln(expenses);
end; (* End of procedure getIncomeInformation *)
```

James Tam

Passing Variable Parameters: Putting It All Together (2)

```
procedure tabulateIncome (  grossSales      : real;
                           costOfGoodsSold : real;
                           var grossIncome  : real;
                           expenses       : real;
                           var netIncome   : real);

begin
  grossIncome := grossSales - costOfGoodsSold;
  netIncome  := grossIncome - expenses;
end; (* End of procedure tabulateIncome *)
```

James Tam

Passing Variable Parameters: Putting It All Together (3)

```
procedure displayIncomeStatement (grossSales      : real;
                                costOfGoodsSold : real;
                                grossIncome     : real;
                                expenses        : real;
                                netIncome       : real);

begin
  writeln;
  writeln('INCOME STATEMENT':40);
  writeln('Gross sales $':40, grossSales:0:2);
  writeln('Less: Cost of the goods that were sold $':40, costOfGoodsSold:0:2);
  writeln('Equals: Gross Income $':40, grossIncome:0:2);
  writeln('Less: Business Operating Expenses $':40, expenses:0:2);
  writeln('Equals: Net income $':40, netIncome:0:2);
  writeln;
end; (* End of displayIncomeStatement *)
```

James Tam

Passing Variable Parameters: Putting It All Together (4)

```
procedure produceIncomeStatement;

var
  grossSales      : real;
  grossIncome     : real;
  costOfGoodsSold : real;
  expenses        : real;
  netIncome       : real;

begin
  getIncomeInformation(grossSales, costOfGoodsSold, expenses);
  tabulateIncome(grossSales, costOfGoodsSold, grossIncome, expenses, netIncome);
  displayIncomeStatement
    (grossSales, costOfGoodsSold, grossIncome, expenses, netIncome);
end; (* End of procedure produceIncomeStatement *)
```

James Tam

Passing Variable Parameters: Putting It All Together (5)

```
procedure intro;  
begin  
  writeln;  
  writeln('This program will produce an income statement based upon your');  
  writeln('gross sales figures, the cost of the goods that you sold and');  
  writeln('your expenses.');
```

```
  writeln;
```

```
end.;
```

James Tam

Passing Variable Parameters: Putting It All Together (6)

```
(* Begin main program *)  
begin  
  intro;  
  produceIncomeStatement;  
  writeln('Thank you, come again!');
```

```
end. (* End of main program *)
```

James Tam

Functions Vs. Variable Parameters

Functions: *Exactly one value is returned by the function.*

```
function calculateGrossIncome (grossSales      : real;
                              costOfGoodsSold : real) : real;
begin
  calculateGrossIncome := grossSales - costOfGoodsSold;
end;
```

Variable parameters: *One or parameters may be modified in the module*

```
procedure tabulateIncome ( grossSales      : real;
                          costOfGoodsSold : real;
                          var grossIncome  : real;
                          expenses        : real;
                          var netIncome   : real);
begin
  grossIncome := grossSales - costOfGoodsSold;
  netIncome   := grossIncome - expenses;
end;
```

James Tam

Scope

It determines when a part of a program (constant, variable, function, procedure) is available for use in that program.

e.g., variables or constants must first be declared before they can be referred to or used.

```
begin
  var num: integer;
  num := 10;
  :      :
end.
```

James Tam

Scope

It determines when a part of a program (constant, variable, function, procedure) is available for use in that program.

e.g., variables or constants must first be declared before they can be referred to or used.

```
begin
  var num: integer;
  num := 10;
  :      :
end.
```

Declaration

Usage

James Tam

Scope

It determines when a part of a program (constant, variable, function, procedure) is available for use in that program.

e.g., variables or constants must first be declared before they can be referred to or used.

```
begin
  var num: integer;
  num := 10;
  :      :
end.
```

Comes into scope

Scope of num

Goes out of scope

James Tam

Global Scope

Global scope: After declaration, the item (constant, variable, function or procedure) can be accessed anywhere in the program.

```
program exampleProgram;
```

```
Declarations here have global scope
```

```
procedure proc;
```

```
var
```

```
Declarations with local scope
```

```
begin
```

```
end;
```

```
begin
```

```
Declarations with local scope
```

```
end.
```

James Tam

Global Scope (2)

When an identifier (constant, variable, function or procedure) is encountered the compiler will:

- First check in the local scope
- Check the global scope if no matches can be found locally

For example:

```
program exampleProgram;
```

```
var
```

```
num : integer;
```

2) Check global scope

```
procedure proc;
```

```
var
```

```
num : integer;
```

1) Check local scope

```
begin
```

```
num := 1;
```

Reference to an identifier

```
end;
```

```
begin
```

```
: :
```

```
end.
```

James Tam

First Scoping Example

The full version of this program can be found in Unix under:
`/home/231/examples/modules/scope1.p`

```
program scope1 (output);
const
  SIZE = 10;
var
  num1 : integer;
  ch   : char;
procedure proc1;
var
  num2 : real;
  num3 : real;
begin
  writeln('In proc1');
end;
begin
end.
```

James Tam

Second Scoping Example

The full version of this program can be found in Unix under:
`/home/231/examples/modules/scope2.p`

```
program scope2 (output);
var
  num : integer;
  ch  : char;
procedure proc1;
var
  ch : char;
begin
  ch := 'b';
  writeln('In proc1');
  writeln ('num=', num, ' ch=', ch);
  writeln;
end;
```

James Tam

Second Scoping Example (2)

```
procedure proc2(numProc2: integer);
var
  num : integer;
begin
  writeln('In proc2');
  num := 2;
  numProc2 := 20;
  writeln('num=', num, ' ch=', ch, ' numProc2=', numProc2);
  writeln;
  proc1;
end;
```

James Tam

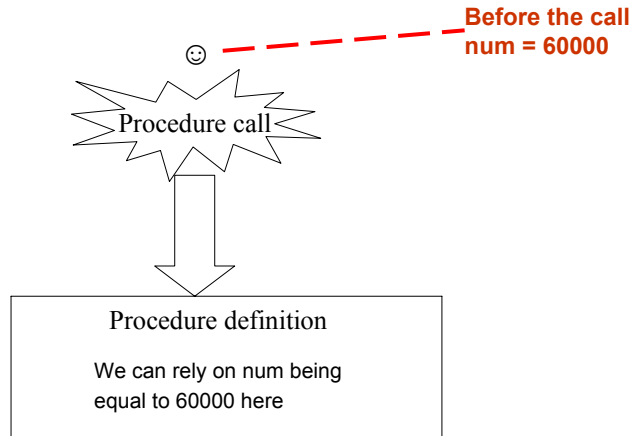
Second Scoping Example (3)

```
begin
  var numLocal : integer;
  num := 1;
  ch := 'a';
  numLocal := 10;
  writeln;
  proc2(numLocal);
  writeln('In main program');
  writeln('num=', num, ' ch=', ch, ' numLocal=', numLocal);
end.
```

James Tam

Preconditions

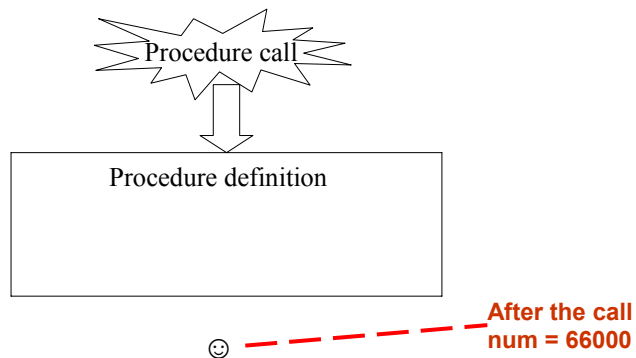
Describe what should be true before a statement is executed
e.g., What will be the value of a variable before a procedure call.



James Tam

Postconditions

Describe what should be true after a statement is executed
e.g., What will be the value of a variable after a procedure call.



James Tam

Preconditions And PostConditions

Relative: One procedure's postcondition can be another procedure's precondition

e.g.,

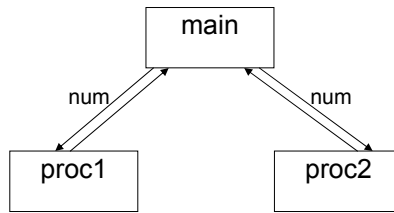
begin

var num: integer;

proc1(num);

proc2(num);

end.



James Tam

Preconditions And PostConditions

Assertions: Making assumptions about what is the state of (a part of) the program at a certain point.

☺

procedure getAge (var age : integer);

begin

write('How old are you (1-113 years)? ');

readln(age);

end;

☺

☺

function calculateAgeModifier (age : integer): integer;

begin

if (age >= 1) AND (age <= 25) then

calculateAgeModifier := age * 2;

else if (age >= 26) AND (age <= 65) then

calculateAgeModifier := age * 3;

else if (age >= 66) AND (age <= 113) then

calculateAgeModifier := age * 4;

else

calculateAgeModifier := 0;

end;

No
precondition
on 'age'

Post condition:
'age' is 1 - 113

Precondition:
'age' is 1 - 113

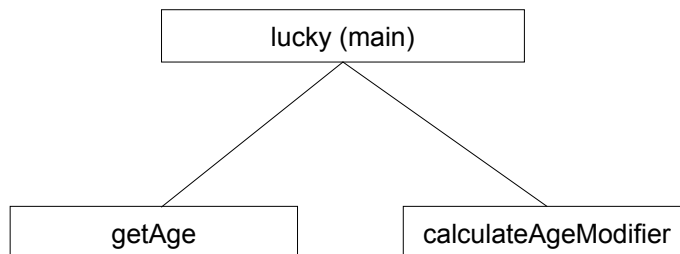
James Tam

Testing Modules

- Making sure the function or procedure does what it is supposed to do e.g., checking if calculations are correct.
- Ties into the top-down approach to design
 - 1) Outline the structure of the program with skeletons (empty modules)
 - 2) As modules are implemented test each one as appropriate
 - 3) Fix the bugs and add the working module to the program.

James Tam

Outline Of The Lucky Number Program



James Tam

Code Skeleton For The Lucky Number Generator

```
program Lucky (input, output);

procedure getAge (var age : integer);
begin

end;

function calculateAgeModifier (age : integer): integer;
begin
  calculateAgeModifier := 0;
end;

begin
  var age          : integer;
  var ageModifier : integer;
  getAge (age);
  ageModifier := calculateAgeModifier(age);
end.
```

James Tam

Implementation Of Procedure “getAge”

```
procedure getAge (var age : integer);
begin
  write('How old are you (1-113 years)? ');
  readln(age);
end;
```

James Tam

Testing Procedure “getAge”

Testing simply involves checking the input:

```
(* In the main procedure *)
getAge(age);
writeln('After getAge, age=', age);
```

James Tam

Implementing Function “calculateAgeModifier”

```
function calculateAgeModifier (age : integer): integer;
begin
  if (age >= 1) AND (age <= 25) then
    calculateAgeModifier := age * 2
  else if (age >= 26) AND (age <= 65) then
    calculateAgeModifier := age * 3
  else if (age >= 66) AND (age <= 113) then
    calculateAgeModifier := age * 4
  else
    calculateAgeModifier := 0;
end;
```

James Tam

Testing Function “calculateAgeModifier”

```
(* Testing in the main procedure calculateAgeModifier*)
ageModifier := calculateAgeModifier(0);
if (ageModifier <> 0) then
  writeln('Error if age < 1');

ageModifier := calculateAgeModifier(114);
if (ageModifier <> 0) then
  writeln('Error if age > 113');

ageModifier := calculateAgeModifier(20);
if (ageModifier <> 40) then
  writeln('Error if age 1 - 25');

ageModifier := calculateAgeModifier(40);
if (ageModifier <> 120) then
  writeln('Error if age 26 - 65');
```

James Tam

Testing Function “calculateAgeModifier” (2)

```
ageModifier := calculateAgeModifier(70);
if (ageModifier <> 280) then
  writeln('Error if age 66 - 113');
```

James Tam

Why Use Modular Design

Drawback

- Complexity – understanding and setting up inter-module communication may appear daunting at first
- Tracing the program may appear harder as execution appears to “jump” around between modules.

Benefit

- Solution is easier to visualize
- Easier to test the program
- Easier to maintain (if modules are independent)

James Tam

You Should Now Know

How to break a programming problem down into modules

What is the difference between a procedure and a function

What is the difference between a value parameter and variable parameter

How to define and call program modules (procedures and functions)

Variables and scope

- What is a local variable
- What is a global variable
- What is the scope of a procedure or function

What are preconditions and post-conditions

How to test functions and procedures

James Tam