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.

Problem Solving Approaches

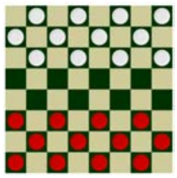
1. Bottom up
2. Top down

Bottom Up Approach To Design

1. Start implementing all details of a solution without first developing a structure or a plan.

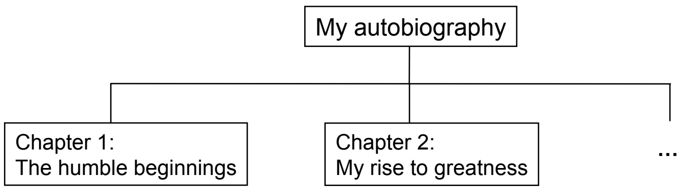
Here is the first of my many witty anecdotes, it took place in a "Tim Horton's" in Balzac..

- Potential problems:
  - (Generic problems): Redundancies and lack of coherence between sections.
  - (Programming specific problem): Trying to implement all the details of large problem all at once may prove to be overwhelming.



Top Down Design

1. Start by outlining the major parts (structure)


My autobiography
Chapter 1: The humble beginnings
Chapter 2: My rise to greatness
...

2. Then implement the solution for each part

Chapter 1: The humble beginnings

It all started seven and one score years ago with a log-shaped work station…

Top-Down Approach: Breaking A Large Problem Down

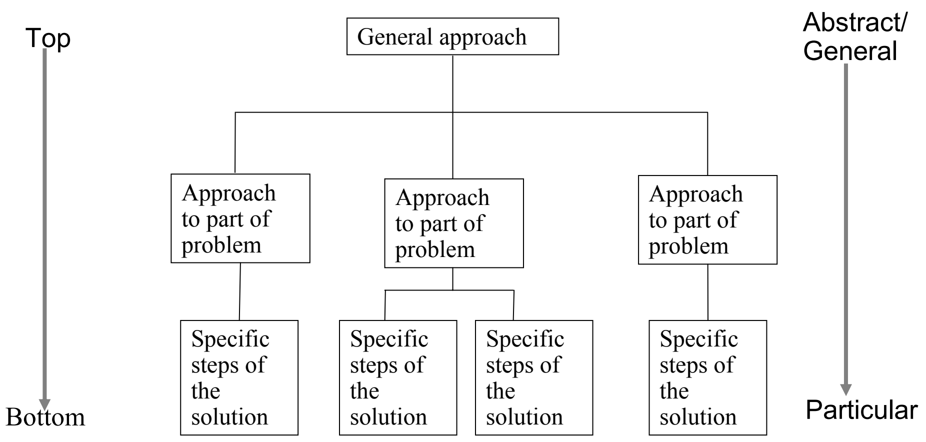
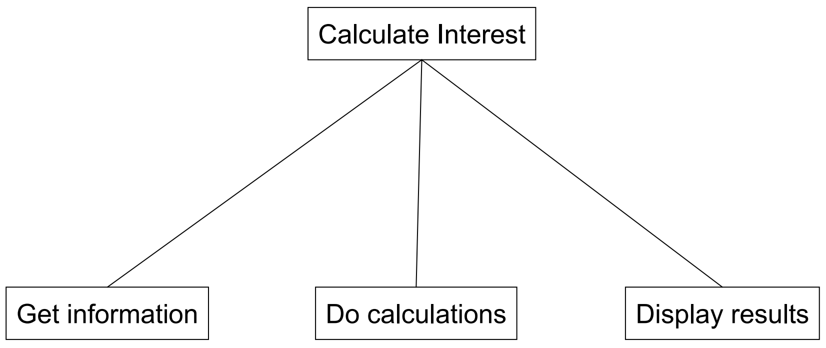

Top
Bottom
General approach
Approach to part of problem
Approach to part of problem
Approach to part of problem
Specific steps of the solution
Specific steps of the solution
Specific steps of the solution
Specific steps of the solution
Abstract/ General
Particular

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

Top Down Approach: Breaking A Programming Problem Down Into Parts (Modules)



Types Of Modules That Can Be Used In Pascal

- Procedures
- Functions

Situations In Which Functions And Procedures Are Used In Pascal

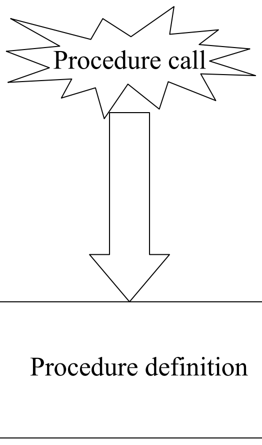
Definition

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

Call

- Getting the function or procedure to run (executing the module)

Procedures (Basic Case – No Information Is Passed In/ No Parameters)



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 *)
```

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 (within the 'body' of a method.

Header

```
```

Declarations

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

Statements

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

James Tam

Important: A Module Must Be Defined Before It Can Be Called!

Correct ☺

```
program exampleModule (output);

procedure exampleProcedure;
begin
   :
end;

begin
   exampleProcedure;
end.
```

First: Defining the module

Second: Calling the module

James Tam

Important: A Module Must Be Defined Before It Can Be Called! (2)

Incorrect ☹

```
program exampleModule (output);
begin
    exampleProcedure;
end.

procedure exampleProcedure;
begin
    :
end;
```

Code?

First: Calling the module

Second: Defining the module

Procedures: Putting Together The Basic Case

The full version of this example can be found in Unix under /home/231/tamj/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 *)
```

Procedures: Putting Together The Basic Case

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

```
program firstExampleProcedure (output);

procedure displayInstructions;
begin
   writeln ('The ');
end; (*Procedure displayInstructions *)

begin
      displayInstructions;
      writeln('Thank you, come again!');
```

Procedure definition

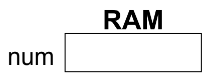
Procedure call

James Tam

What You Know: Declaring Variables

- Variables are memory locations that are used for the temporary storage of information.

```
var num : integer;
```

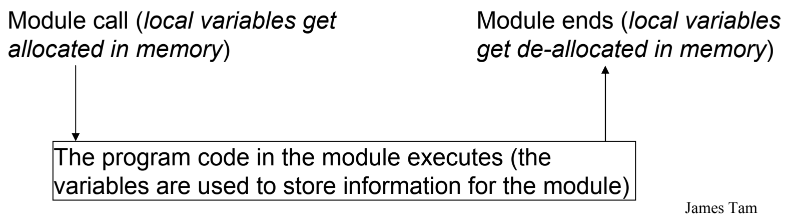


- Each variable uses up a portion of memory, if the program is large then many variables may have to be declared (a lot of memory may have to be allocated – used up to store the contents of variables).

James Tam

What You Will Learn: Declaring Variables That Are Local To Modules

- To minimize the amount of memory that is used to store the contents of variables only declare variables when they are needed.
- When the memory for a variable is no longer needed it can be 'freed up' and reused.
- To set up your program so that memory for variables is only allocated (reserved in memory) as needed and de-allocated when they are not (the memory is free up) variables should be declared locally to modules.


Module call (*local variables get allocated in memory*)
Module ends (*local variables get de-allocated in memory*)
The program code in the module executes (the variables are used to store information for the module)

How To Declare Local Variables

Format:

```
procedure name;
var
     <variable 1 name> : <variable 1 type>;
     <variable 2 name> : <variable 2 type>;
           :                    :
 begin
         :
 end;
```

Example:

```
procedure proc;
var
    num1 : integer;
    num2 : integer;
begin
    :   :
 end;
```

Defining Local Variables: Putting It All Together

The full version of this example can be found in Unix under
/home/231/tamj/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.
```

Defining Local Variables: Putting It All Together

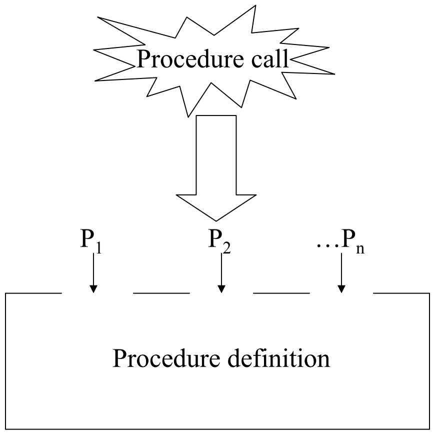
The full version of this example can be found in Unix under
/home/231/tamj/examples/modules/secondExampleProcedure.p

```
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

Procedures With Parameters/Information Passed In



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 *)
```

Calling Modules (Procedures) With Parameters

Format:

name (*Name of parameter 1*, *Name of parameter 2…Name of parameter n*);

Example:

```
celciusToFahrenheit (celciusValue);
```

Procedures: Putting Together The Case Of Procedures With Parameters

The full version of this example can be found in Unix under /home/231/tamj/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 *)
```

Procedures: Putting Together The Case Of Procedures With Parameters

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

Procedure definition

```
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 *)
```

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 *)
```

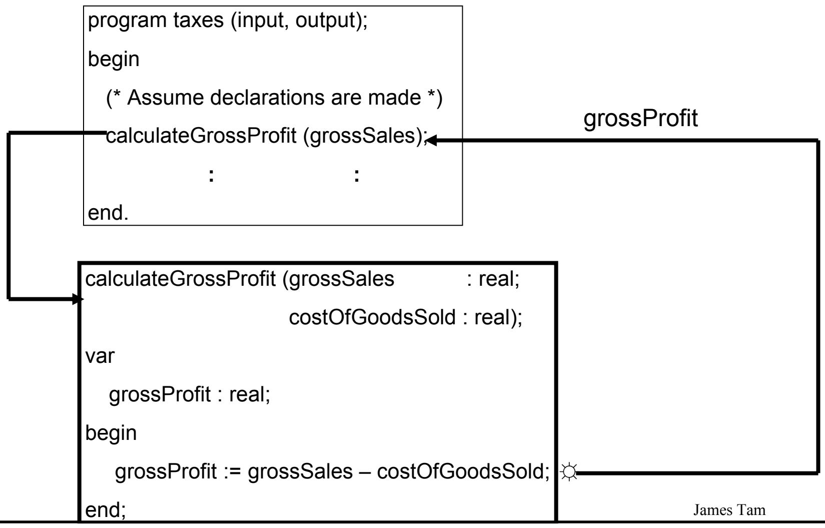
Procedures: Putting Together The Case Of Procedures With Parameters (2)

```
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 *)
```

Procedure call

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

For example: producing an income statement



```
program taxes (input, output);
begin
   (* Assume declarations are made *)
   calculateGrossProfit (grossSales);
             :                :
end.
```

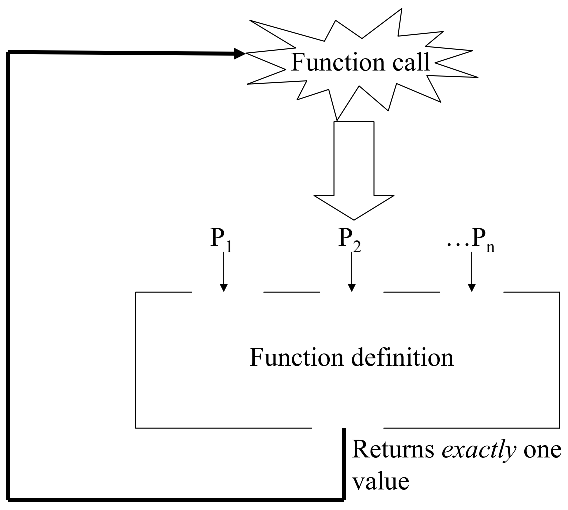
```
calculateGrossProfit (grossSales          : real;
                      costOfGoodsSold : real);
var
   grossProfit : real;
begin
    grossProfit := grossSales – costOfGoodsSold; ☼
end;
```

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



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;
```

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: Typically the last statement that gets executed in the function

Calling Functions

Format:

variable := *name of function*;

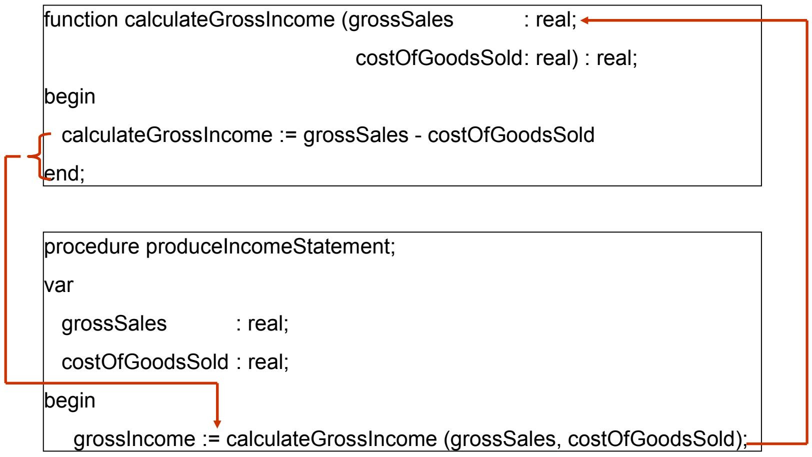
variable := *name* of function (*name of parameter 1*, *name of parameter 2…name of parameter n*);

Example:

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

James Tam

Tracing The Execution Of A Function Call



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

```
procedure produceIncomeStatement;
var
  grossSales         : real;
  costOfGoodsSold : real;
begin
   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/tamj/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;
```

Functions: Putting It All Together

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

Function definitions

```
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;
```

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);
```

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

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 *)
```

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;
```

Functions: Putting It All Together (5)

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

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)**

Passing Parameters As Value Parameters

Previous examples

```
procedureName (p1);
```

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

Passing Parameters As Value Parameters

Previous examples

```
procedureName (p1);
```

Pass a copy

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

Passing Parameters As Variable Parameters

Example coming up

```
procedureName (p1);
```

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

Passing Parameters As Variable Parameters

Example coming up

```
procedureName (p1);
```

Pass the variable

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

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/tamj/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 *)
```

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 *)
```

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 *)
```

Passing Variable Parameters: Putting It All Together (4)

```
procedure produceIncomeStatement;
var
  grossSales        : real;
  costOfGoodsSold   : real;
  grossIncome       : 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 *)
```

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.;
```

Passing Variable Parameters: Putting It All Together (6)

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

Functions Vs. Variable Parameters

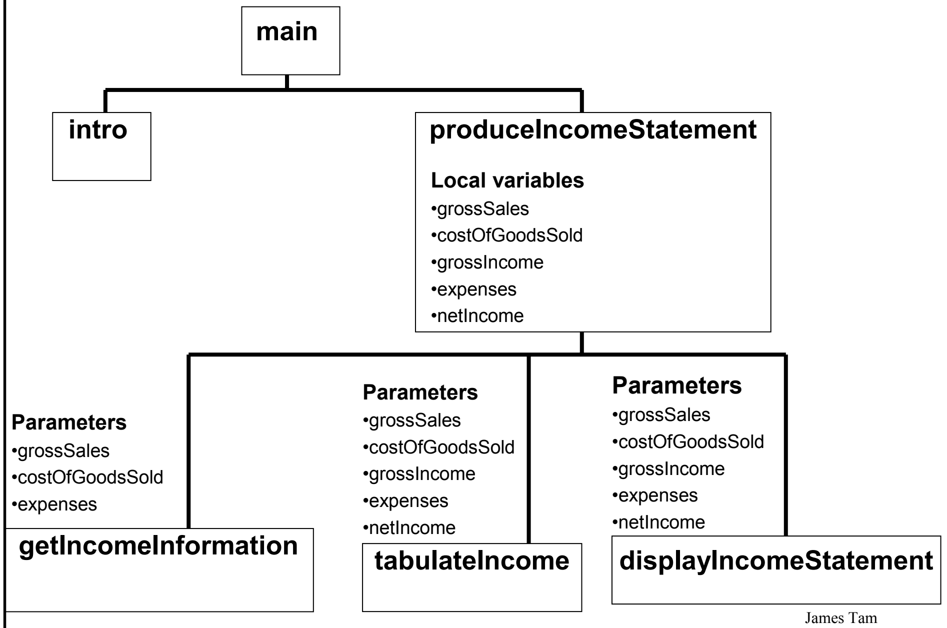
Functions: *Exactly one value is returned by the function.*

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

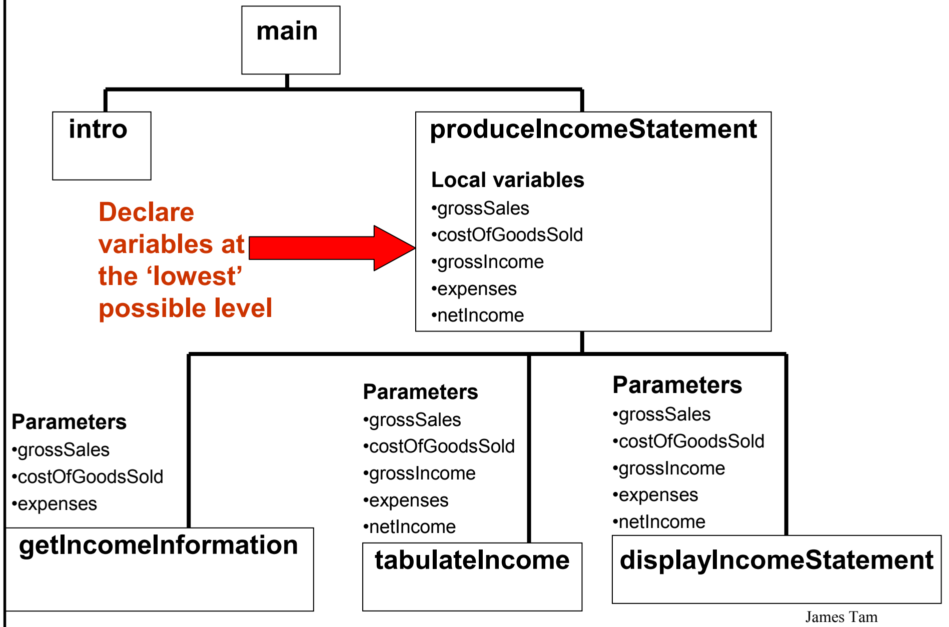
Variable parameters: *One or more parameters may be modified in the module*

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

Where To Declare Variables?



Where To Declare Variables?



Variable Parameters And Multiple Module Calls

- If during a series of module calls, a parameter must be passed as a variable parameter during one of the later module calls then the earlier module calls must also pass the parameter as a variable parameter.

```
procedure proc2 (var num : integer);
begin
    proc1(num);
end;
```

If the changes made to 'num' in proc1 are to persist after proc2 ends then num must ALSO be passed as a variable parameter to proc2

```
procedure proc1 (var num : integer);
begin
    num := num * 10;
end;
```

In proc1 'num' is being changed so if those modifications are to persist after the module ends then num must be passed as variable parameter.

Variable Parameters And Multiple Module Calls (2)

```
program parameter1 (output);

procedure proc1 (var num : integer);
begin
  num := num * 10;
end;

procedure proc2 (var num : integer);
begin
  writeln('num=', num);
  proc1(num);
end;

begin
  var num : integer;
  num := 7;
  writeln('num=', num);
  proc2(num);
  writeln('num=', num);
end.
```

Correct approach:

If the changes made to ‘num’ in proc1 are to persist after the procedure calls end then num must ALSO be passed as a variable parameter to proc2

James Tam

Variable Parameters And Multiple Module Calls (3)

```
program parameter1 (output);

procedure proc1 (var num : integer);
begin
  num := num * 10;
end;

procedure proc2 (num : integer);
begin
  writeln('num=', num);
  proc1(num);
end;

begin
  var num : integer;
  num := 7;
  writeln('num=', num);
  proc2(num);
  writeln('num=', num);
end.
```

Incorrect approach:

The changes made to ‘num’ in proc1 will be made to a variable that is local to proc2. The variable num that is local to ‘main’ will not be modified by the changes made in proc1.

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.
```

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.

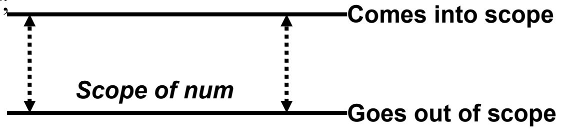
```
begin                                Declaration
    var num: integer;
    num := 10;
                                     Usage
      :          :
end.
```

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.
```



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.
```

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;                  2)  Check global scope
var
   num : integer;

procedure proc;                          1)  Check local scope
var
    num : integer;                       Reference to an identifier
begin
    num := 1;
end;

begin
    :        :
end.
```

First Scoping Example

The full version of this program can be found in Unix under:
/home/231/tamj/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.
```

Avoid / Minimize The Use Of Global Variables

- Remember global variables can be accessed or changed anywhere in the program after their declaration.
- This results in:
  - Tightly coupled modules – changes in one module may effect other modules
  - Programs that are more difficult to trace and understand.
- Unless there is a compelling reason variables should be declared locally and passed as a parameter where ever it is needed.

Second Scoping Example

The full version of this program can be found in Unix under:
/home/231/tamj/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;
```

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;
```

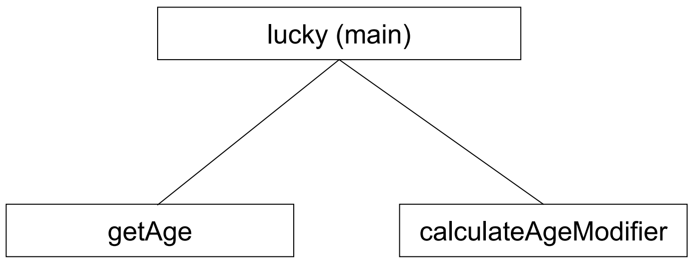
Second Scoping Example (3)

```
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.
```

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.

Outline Of The Lucky Number Program



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.
```

Implementation Of Procedure “getAge”

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

Testing Procedure “getAge”

Testing simply involves checking the input:

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

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;
```

Testing Function “calculateAgeModifier”

```
(* Testing calculateAgeModifier in the main procedure *)
 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');
```

Testing Function “calculateAgeModifier” (2)

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

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)

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

How to test functions and procedures