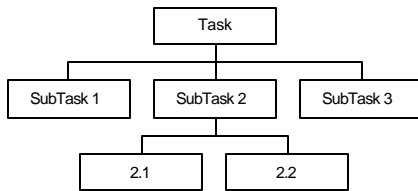


SubPrograms



- Top-Down Design using stepwise refinement leads to division of tasks

Outline

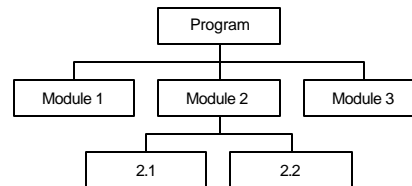
III. Modular Programming

- A. Advantages
 - readability, reusability, modularity, data security
- B. Declaration (prototype)
- C. Definition
- D. Call
- E. Parameters
 - by value
 - by reference
- F. Executing Functions

Outline (cont)

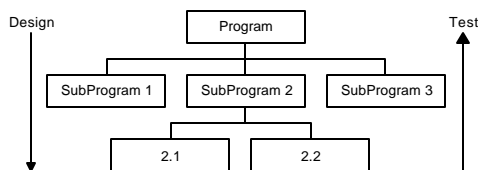
- G. Function Types
 - no parameters
 - value parameters
 - reference parameters
 - return type
- H. Scope
- I. Documentation
- J. Iterative Programming
- K. Standard Library Functions
 - use of include
 - abs, labs, fabs, floor, ceil, rint, pow, sqrt, rand, srand

Modularity



- Subtasks captured by Modules of program
- Should be able to design, code, test each module independently

Structured Programming



- Structured programs lend themselves to Bottom-Up Testing
- Large programs can be written by teams

What is a Subprogram?

- Program within a program
- Own name and set of local declarations
- In C, done with functions
- Advantages
 - reusability
 - readability
 - modularity (ease of implementation)
 - protecting data

Unstructured Program

```
int main( void ) {
  /* Get data from user */
  statement1;
  statement2;
  statement3;
  /* Perform computations */
  statement4;
  statement5;
  statement6;
  statement7;
  /* Print results */
  statement8;
  statement9;
}
```

Corresponding Structured Program

```
int main( void ) {
  GetData(parameters); /* procedure calls */
  Compute(parameters);
  Print(parameters);
}
```

- Subprograms (functions) defined after main
- Program more *readable*, function names give an idea of what is being done

Flow through Structured Program

```
GetData -- call --> statement1;
                    statement2;
                    statement3;
                    <- return -
Compute -- call --> statement4;
                    statement5;
                    statement6;
                    statement7;
                    <- return -
Print   -- call --> statement8;
                    statement9;
                    <- return -
```

Reusability

- Subprograms can be reused (no rewriting)

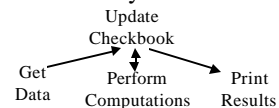

```
{
  funcA(); /* Statements in A executed */
  funcB(); /* Statements in B executed */
  funcA(); /* A executed again */
}
```
- But what if two versions of A differ slightly?
 - (e.g., print 1st person data, 2nd person data)

Parameters

- Parameters used to supply information to, and receive info from functions
 - for example, pass print person function the data to be printed (once with data for first person, once with data for second person)
- Types of parameter passing in C
 - by value: value is calculated, then passed
 - by reference: address is calculated, passed
 - reference allows functions to change variables

Transmission of Data

- Arrows used in structure chart to indicate if data received/sent by module



- Module specifications describe data:
 - Perform Computations
 - Data received: starting balance, trans type and amount
 - Info returned: ending balance

Function Call

- Syntax: *Name(Argument_List)*
- Name indicates which function being called
- Arguments give values for each parameter in the parameter list (separated by commas)
 - syntax: *Value, Value, Value, ...*
 - one argument for each parameter
 - parameters matched in order
 - must be of same type as parameter
- If function returns value, it may be included in expression, if void, may stand alone

Function Execution

- Parameters evaluated:
 - Each argument is evaluated to produce a value
 - New local variable created for each parameter
 - Arguments copied to new local variables
- Body of function executed:
 - Local declarations created (local variables, etc.)
 - Each statement in body executed in order (until last statement or return statement encountered)
- Function ends:
 - local variables removed (freed)
 - return value determined
 - execution returns to after call

Execution Example

```
int main ( void ) {
    float tempConv(float f); /* Prototype */
    float ftemp = 77;
    float ctemp;

    ctemp = tempConv(ftemp); /* Call */
}

float tempConv(float f) { /* Header */
    float cval; /* Local Decl */

    cval = (f - 32) * 5 / 9; /* Statements */
    return cval; /* Return */
}
```

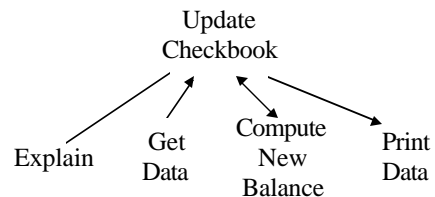
Execution Example (cont)

- Main function executed:
 - variables declared (initialized)
 - ftemp: 77 ctemp: ?
 - tempConv called
 - ftemp evaluated (77)
 - local var f created
 - value 77 is copied to f
 - local var cval created
 - assignment statement executed (25 stored in in cval)
 - return executed - 25 sent as value calculated
 - assignment evaluated - 25 stored in ctemp

Function Prototype

- Return type must be the same in definition
- Parameter list must have same number, same type and same order as list in def.
- Parameter names do not have to match
- In fact, parameter names can be left out entirely
- Prototypes can be omitted in function used is defined *first*

Checkbook Program



Checkbook Example

```
int main ( void ) {
void Explain( void ); /* Prototypes */
void GetData( float *StartBal,
char *TransType, float *TransAmount );
float ComputeNewBal( float StartBal,
char TransType, float TransAmount );
void PrintData( float StartBal,
float EndBal, char TransType,
float TransAmount );

float sbal; /* Local vars in main */
char ttype;
float tamount;
float ebal;
```

Checkbook Example

```
/* Function calls making up program */

Explain();
GetData(&sbal,&ttype,&tamount);
ebal =
ComputeNewBal ( sbal , ttype , tamount );
PrintData ( sbal , ebal , ttype , tamount );

return 0;
}
```

Function without Parameters

```
void Explain( void ) {
printf("This is a checkbook program.");
printf(" I will prompt for a starting\n");
printf("balance and then a transaction");
printf(" to perform (a Withdrawl\n");
printf("or Deposit) and the amount of");
printf(" the transaction.\n");
printf("\n");
}
```

Function without Parameters (cont)

- Called with an empty argument list
 - example: Explain();
- Same set of statements executed
 - could still differ if the statements interact with the user
- Could return a value (reading a number from user which is returned as value calculated)

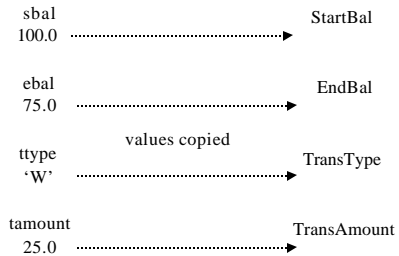
Passing Data to Function

```
void PrintData( float StartBal,
float EndBal, char TransType,
float TransAmount ) {
printf("\n\n");
printf("Starting Balance $%.2f\n",
StartBal);
printf("Transaction $%.2f %c\n",
TransAmount,TransType);
printf(" ----- \n");
printf("Ending Balance $%.2f\n",
EndBal);
}
```

Parameters Passed by Value

- Arguments evaluated to produce values
- Local variables are created for each formal parameter
- Values are copied to new variables
- Arguments and local variables are then separated (changing a local variable does not change the argument)

Calling PrintData



Changing Local Variables

- If local variable changed, does not affect argument
 - example:
`EndBal = EndBal - 25.0;`
– does not change `ebal`
- When function ends, local variables are released

Parameter Passing Rules

- Number of formal parameters/arguments must match
- Types of formal parameters/arguments must match
- Order of formal parameters/arguments must match
- Types of parameters determined in function header

Arguments for Parameters

- When passing parameters by value, any value of the correct type is ok
 - example:
`PrintData(sbal,75.0,(char) 87,25.0 * 3);`
- The value of the argument is determined, and that value is then copied to the new local variable

Changing Non-local Variables

- The `scanf` function actually changes the value of its arguments:
 - example:
`scanf("%f",&Balance);`
changes the variable `Balance`
but how?
- The key is the `&`, the argument is being passed by *reference* (a value is passed, the location of the variable `Balance` in memory)

Passing by Reference

- Argument passed is address of a variable
 - in the call, an `&` is included before the variable address being passed
- In function header, type of the variable is the type of the variable being passed by reference followed by `*`
- Variable is the address of a location (to refer to the value at the address also use `*`)

Passing by Reference Example

```
void GetData( float *StartBal,
char *TransType, float *TransAmount ) {
printf("Please enter starting balance: ");
scanf("%f", StartBal);
fflush(stdin);
printf("What type of transaction (W for");
printf(" withdrawl or D for Deposit): ");
scanf(" %c", TransType);
printf("What is the amount of the ");
printf("transaction: ");
scanf("%f", TransAmount);
```

Passing by Reference Example

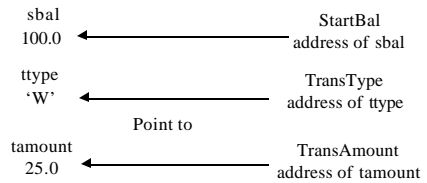
```
printf("\nYou have entered the following");
printf(" information:\n");
printf("Starting Balance: %1.2f\n",
*StartBal);
printf("Transaction Type: %c\n", *TransType);
printf("Transact. Amount: %1.f\n",
*TransAmount);
printf("\n");
}
```

Using Reference Variables

- Name is the address of the argument
Example: StartBal in GetData is the location of the argument
- Can refer to the variable using the * form
Example: *StartBal is the value of the variable StartBal is connected to

Calls with Reference Variables

```
GetData(&sbal,&ttype,&tamount);
```



Reference Arguments

- Argument must be a variable with an & before the variable
Example:
GetData(&(sbal + 25.0),ttype,&75);
Each argument is a problem (no address for each case)

Functions with Return Types

- Functions that return/calculate single values can use the return type for that value
- Type of the function indicates the value being calculated
- Must be at least one return statement in function
- Return statement indicates value calculated

Function with Return

```
float ComputeNewBal( float StartBal,
char TransType, float TransAmount ) {
float EndBal = StartBal;

if (TransType == 'W')
    EndBal -= TransAmount;
else if (TransType == 'D')
    EndBal += TransAmount;

return EndBal; /* value calculated */
}
```

Creating Functions

- Choose a meaningful name
- Data sent to function?
 - Yes, one parameter for each data value
 - No, void parameter list
- Function calculates/changes one value?
 - Yes, use appropriate return type
 - No, void return type
- Function calculates/changes >1 value?
 - Yes, use reference parameters

Scope

- Scope of identifier is part of program where the identifier can be referenced
- Global scope - identifiers declared outside of any function, can be viewed anywhere in file
- Local scope - identifiers declared inside a function are local to that function, also identifiers declared within blocks are local to that block

Scope Example

```
int x1;
float f1
(float x2) {
int x3;
{
char x4;
}
```

x1, f1 declared in outer scope

x2, x3 declared in inner scope

x4 declared inner, inner scope

Scope Rules

- An identifier can only be used in a block in which it is declared
- Declaration of identifier must occur before its first use
- For two identifiers with the same name, the one with the smaller scope is the one in effect within its subblock

Using Same Identifier

```
int x; /* global x */
float f1( void ) {
float x; /* x within f1 */
{
char x; /* x in block within f1 */
x = 'A';
x = 5.0;
}
int main( void ) {
x = 3;
}
```

Programming Practices

- Changing global variables within functions allowed but not encouraged
- To change global variables pass them as reference parameters to functions
- Since constants do not change, can be referenced anywhere
- Using same name for different variables is a matter of style

Documenting Subprograms

- Each function should have comment following header with:
 - /* Given: summary of data received */
 - /* Task: statement of task being performed */
 - /* Return: statement of value(s) to be returned */
- Specification should be written during algorithm design

Developing Modular Programs

- Develop one module at a time
 - not necessarily first to last
- Test modules independently using driver program
 - driver has call(s) to test modules
 - calls to other modules left or commented out
 - completely test before working on next module
- *Iterative programming*

Iterative Programming

- Create main function with function calls
- Comment out calls to functions not written
- As functions defined, add calls
- main of checkbook program:

```
Explain(); /* test Explain() alone */
/* GetData(&sbal, &ttype, &tamount);
ebal = ComputeNewBal(sbal, ttype, tamount);
PrintData(sbal, ebal, ttype, tamount); */
```

Iterative Programming (cont)

- Can test even if earlier routines not implemented
- Test Print Data by passing test values rather than variables:

```
Explain(); /* test Explain() alone */
/* GetData(&sbal, &ttype, &tamount);
ebal = ComputeNewBal(sbal, ttype, tamount);
*/
PrintData(100.0, 125.0, 'D', 25.0);
```

Standard Library Functions

- C provides libraries that contain lots of functions of interest
 - stdio.h - standard input/output routines
 - stdlib.h - other useful functions
 - math.h - math functions
- To use these functions we include the header file of the library (ending in .h)
 - header file contains prototypes of functions
 - definitions are compiled elsewhere, connected to your program when the program is linked

Using Library Functions

- Add #include with name of desired library
example: #include <math.h>
- May need to inform linker that library should be added to program
using gcc or cc on unix :
gcc filename.c -lm
- How this is done depends on compiler

Absolute Value Functions

- From <stdlib.h>
- Prototypes:
int abs(int number)
long labs(long number)
double fabs(double number)
- Return absolute value of number
abs(-5) is 5

Rounding Numbers

From <math.h>

double ceil(double number)
round number up to next whole number
ex. ceil(4.2) is 5.0
double floor(double number)
round number down to next whole number
ex. floor(4.8) is 4.0
double rint(double number)
round number to nearest whole number
ex. rint(4.6) is 5.0

Other Math Functions

double pow(double x, double y)
returns x raised to the exponent y
pow(2.0,3.0) returns 8.0

double sqrt(double number)
returns sqrt of number
sqrt(16.0) returns 4.0

Random Numbers

From <stdlib.h>

int rand(void)
generates mathematically a "random" integer
note the same sequence is generated each time unless
you use srand
void srand(unsigned int seed)
sets the initial "seed" when generating random
sequence (different seed - different sequence)
standard use:
srand(time (NULL)); /* include time.h */
sets seed based on clock time