Programming II - CMP1025

Lecture One

By David W. White
dwwhite@utech.edu.jm

University of Technology, Jamaica
January 14, 2014

Expected Outcome

At the end of this lecture, the student should:

- Have a general understanding of what will be covered in this course
- Explain the purpose of modules in program design
- Explain common concepts used in modularity such as top-down design, stepwise refinement, local and global scope, main driver module

Introduction to the Programming II Course

- The precursor to this course is Programming I (CMP1024)
- Programming I introduced students to structured programming concepts
- Focused on how to design algorithms to solve problems
- … and implement those algorithms in simple C programs

Introduction to the Programming II Course

- This Programming II (CMP1025) course teaches students more advanced concepts
- Includes modularity using functions, arrays, structures, unions and enumerated types, searching, sorting and files
- Also includes recursion and pointers
- Uses C as implementation language

Assessment

- Two lab tests worth 10% each
- Two lecture tests worth 15% each
- One individual assignment worth 10%
- A Final Exam worth 40%

- Total course work = 60%, Final Exam = 40%

Textbook and other resources

- **Recommended Textbooks**
  - C How to Program, Dietel and Dietel, Pearson

  - Head First C, Griffiths and Griffiths, O'Reilly Media

  - Teach yourself C in 21 days, Jones and Aitken, SAMS

  - The C Programming Language, Kernighan and Ritchie, Prentice Hall

- **Recommended Websites**
  - Cprogramming.com
  Short tutorials with examples, exercises and answers
  http://www.cprogramming.com/tutorial/c-tutorial.html

Programming Language and Compiler

- Course will be taught using the C Programming Language
- Compiler used in labs – Microsoft Visual C++ Express 2010
- Compiler available as free download from Microsoft Website or from UTech Lab shared folders

Modularity

- Opposite of the monolithic programs students wrote in Programming I
- Programs are split up into “modules”
- Each module does one specific task, a little chunk of the complete task the entire program will tackle
- Each module practices “separation of concerns”

Functions

- In the C programming language, modules are implemented as C functions
- Each module is implemented as a separate function
- The program starts from the main() function
- Each program must have one and only one main function

Example

- Let us write a program that:
- Prompts and accepts an integer n from the user
- Calculate and display the additive inverse of the number n
- Calculate and display the square of the number n
- Calculate and display the cube of the number n

Monolithic Program Solution

Pseudocode

```
Algorithm main()
Start
      Declare n, i, s, c as integer
      Write “Enter an integer”
      Read n

      i ←  n * -1
      Write “Additive inverse is ”, i

      s ←  n * n
      Write “Additive inverse is ”, s

      c ←  n * n * n
      Write “Additive inverse is ”, c
Stop
```

Monolithic Program Solution

Pseudocode

```
Algorithm main()
Start
    Declare n, i, s, c as integer
    Write “Enter an integer”
    Read n

    i ←  n * -1
    Write “Additive inverse is ”, i

    s ←  n * n
    Write “Additive inverse is ”, s

    c ←  n * n * n
    Write “Additive inverse is ”, c
Stop
```

C Code

```c
#include <stdio.h>

int main()
{
    int n, i, s, c;

    printf("Enter an integer");
    scanf("%d", &n);

    i =  n * -1;
    printf("Additive inverse is %d\n", i);

    s =  n * n;
    printf("Additive inverse is %d\n", s);

    c =  n * n * n;
    printf("Additive inverse is %d\n", c);
    return 0;
}
```

Modular Program Solution

- In devising the modular solution, we look at the different tasks involved, and separate them
- Each task of group of related tasks, becomes a separate module
- Each individual module will have its own name, arguments, and start and stop.
- Finally, a single main() module is needed to tie the other modules together. All programs must have a single main().

Modular Program Solution

Pseudocode

```
Algorithm GetInteger()
Start
    Declare n as integer
    Write "Enter an integer"
    Read n
    Return n
Stop

Algorithm AddInverse(n as integer)
Start
    Declare i integer
    i ←  n * -1
    Write "Additive inverse is ", i
Stop

Algorithm Square(n as integer)
Start
    Declare s as integer
    s ←  n * n
    Write "Additive inverse is ", s
Stop
```

```
Algorithm Cube(n as integer)
Start
    Declare c as integer
    c ←  n * n * n
    Write "Additive inverse is ", c
Stop

Algorithm main()
Start
    Declare n as integer
    n ← GetInteger()
    AddInverse(n)
    Square(n)
    Cube(n)
Stop
```

Modular Program Solution

- The variables and code in a module are like an island unto themselves
- They don't interfere with the variables and code in other modules, and other modules' variables and code don't interfere with those in this module
- The next version of the modular solution in pseudocode demonstrates this

Modular Program Solution

Pseudocode – another version

```
Algorithm GetInteger()
Start
      Declare n as integer
      Write "Enter an integer"
      Read n
      Return n
Stop

Algorithm AddInverse(a as integer)
Start
      Declare i integer
      i ←  a * -1
      Write "Additive inverse is ", i
Stop

Algorithm Square(b as integer)
Start
      Declare s as integer
      s ←  b * b
      Write "Additive inverse is ", s
Stop
```

```
Algorithm Cube(y as integer)
Start
      Declare c as integer
      c ←  y * y * y
      Write "Additive inverse is ", c
Stop

Algorithm main()
Start
      Declare x as integer
      x ← GetInteger()
      AddInverse(x)
      Square(x)
      Cube(x)
Stop
```

Modular Program Solution

- Writing the C program from the modular pseudocode involves translating each module into a separate C function, as demonstrated in the next slide

Modular Program Solution

C Code

```
#include <stdio.h>
int GetInteger()
{
    int n;
    printf("Enter an integer");
    scanf("%d", &n);
    return n;
}

void AddInverse(int a)
{
    int i;
    i =  a * -1;
    printf("Additive inverse is %d\n", i);
}

void Square(int b)
{
    int s;
    s =  b * b;
    printf("Additive inverse is %d\n", s);
}
```

```
void Cube(int y)
{
    int c;
    c =  y * y * y;
    printf("Additive inverse is %d\n", c);
}

int main()
{
    int x;
    x = GetInteger();
    AddInverse(x);
    Square(x);
    Cube(x);
    return 0;
}
```