Review: Chapter 9 – Functions and Program Design

CSCI 2212 INTERMEDIATE PROGRAMMING IN C/C++

Objectives:

1. A Review of Prototypes
2. Scope and Visibility
3. Basic Program Architecture

I. Function Prototypes

HOW TO SET UP A FUNCTION IN C AND C++

Function Prototypes

- **In order to call a function, the system must know about** it, the function must be declared.
- In order to write the function declaration, you must know what it is called and what the parameters and return are.
- A **function call** has some similarities to a **function prototype** but they are not identical.
- A **function prototype** gives the **types of the arguments that must be supplied** in the call.
- The **call gives arguments but does NOT give types**.
- The **number of arguments and their o**rder in the **call must match the prototype.**
- **If the argument is an array name**, the parameter must have an **array type** or a **pointer type**.
- The **length of an array is not part of the array in C**. Thus, the length of an array argument **must be passed to the function as an additional argument.**

Function Calls

When **a function is called**, the **argument**s (in the call) **become the values of the parameters** (in the definition).

- If the **argument is a simple variable** (int, double, etc.), the value is copied into the parameter variable. The **function is not able to change the original copy** of the argument variable.
- If the **argument is an expression**, the value of the expression is copied into the parameter variable.
- If the **argument is the name of an array**, a reference to the array is copied into the parameter variable. This **enables the function to change the contents of the caller's arr**ay.
- When we **use an array argument**, the array is a way for **the function to return values to the caller.**

Type Mismatches

The **types of the arguments in a call** are **supposed to match** the types **declared in the prototype**.

- If **they do not match**, the argument value will be **converted** (if possible) to the parameter type during the call. This is **not an error** and will not cause a warning message.
- If a **required conversion is not possible**, the result is a **compiler error**.
- **Any numeric type can be converted to any** other numeric type. Numeric types include double, float, int, unsigned, long, short, and char.
- **An array** cannot be converted to **another length or base type** of array.

Prototypes

Before a function call can be compiled, the C compiler **needs to know** the type of the function.

- That **can be supplied by an #include file** . . . or **by a prototype at the top of the file**.
- **Alternatively, the entire function definition can be given (at the top of the file) prior to the first call on it.** I prefer functions to be defined at the end of the program for readability.
- **If you do NOT supply this type information, C will construct a prototype for you**, using the types of the **actual arguments in the first call** on that function. The **return type will always be int**.
- This is **likely to cause a compiler error** when the **function prototype or definition is found**: “attempt to redefine" the function.

Example D1: power.c

```
#include <stdio.h>
#include <math.h>
int pow2 ( double ); // prototype gives fxn name, parameter & return types
int main( void )
{
  printf("Welcome to the Power Program, ");
  puts("it will square a series of values. By AEF and LP\n");
  for (int m=0; m<10; ++m)
      printf( “2 raised to the power %2i = %10i\n", m, pow2(m) );
  puts("\nThank you for using the Power Program, bye!");
  return 0;
}

int pow2 ( double x ) { return pow( 2.0, x ); } // 2 to the power x
```

II. Scope and Visibility

CHAPTER 9

Global Symbols

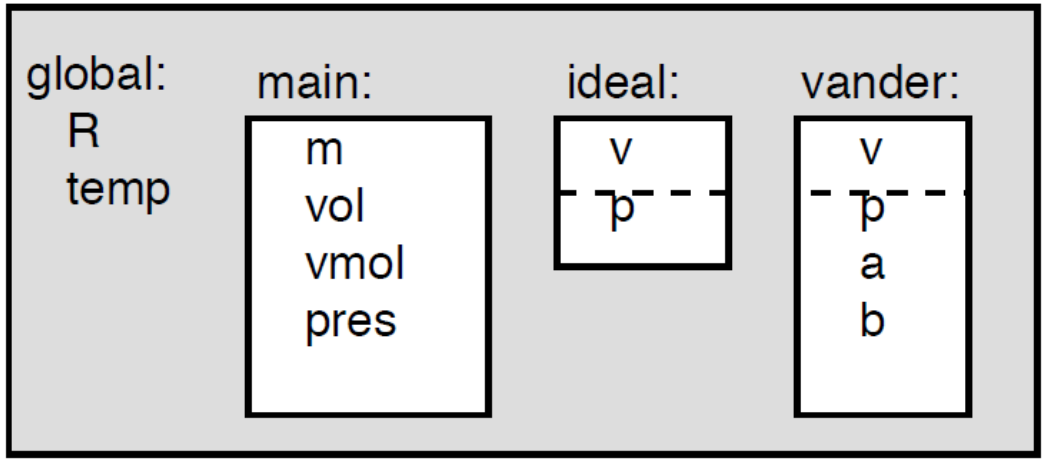
A symbol, such as a variable, can have **local or global scope**.

- A **global symbol is declared outside of all the functions**, usually **at the top**.
- A **#define symbol is globally visible**, starting on the line after its definition.
- **Function names are global**.
- It is possible to **define a variable outside of the functions**.
- Page 332 shows an example of a global variable, temp.
- **Global constants are OK**. However, the use of **global variables leads to too many nasty errors**.
- **The use of global variables is not permitted in this class.**
- **Using a global variable will require revision of your program.**

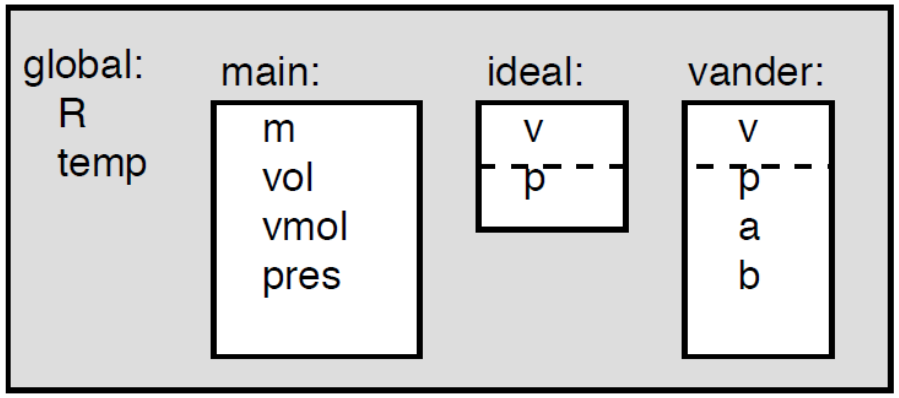
```
#include <stdio.h>
#define R   8314              // universal gas constant
float ideal( float v );       // prototypes
float vander( float v );
float temp;                   // GLOBAL variable: not inside any function
                              // temperature of CO gas

int main( void )
{
    // Local Variables ---------------------------------------
    float m;                   // mass of CO gas, in kilograms
    float vol;                 // tank volume, in cubic meters
    float vmol;                // molar specific volume
    float pres;                // pressure (to be calculated)

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );
    vmol = 28.011 * vol / m;          // molar volume of CO gas
    pres = ideal( vmol );             // pressure; ideal gas model
    printf( "\n The ideal gas at %.3g K has pressure "
            "%.3f kPa \n", temp, pres );
    pres = vander( vmol );         // pressure; Van der Waal's model
    printf( "\n Van der Waal's gas has pressure "
            "%.3f kPa \n\n", pres );

    return 0;
}
```

Figure 9.5. Gas models before global elimination.

Scope of the Variables

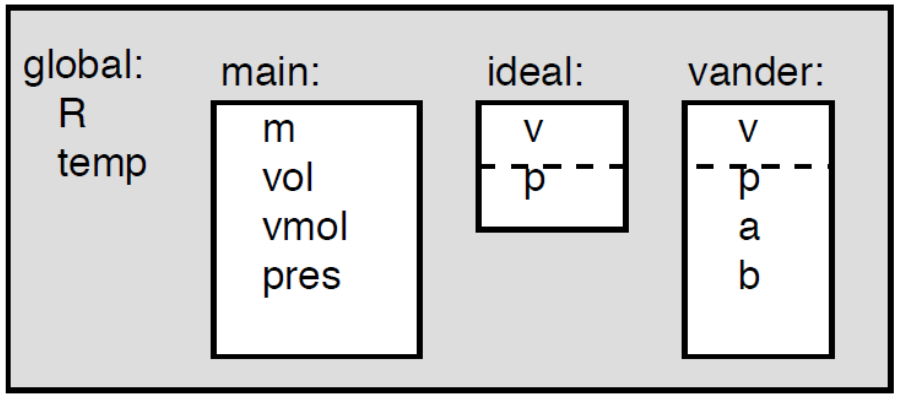


Scope of the Variables




```
// ---------------------------------------------------------------
//  Pressure of CO gas in a tank, using the ideal gas equation Pv = RT.
float ideal( float v )
{
    float p;                    // LOCAL variable DECLARATION

    p = R * temp / v;           // pressure in Pascals
    return p / 1000.0;          // pressure in kilo Pascals (kPa)
}
```

Scope of the Variables

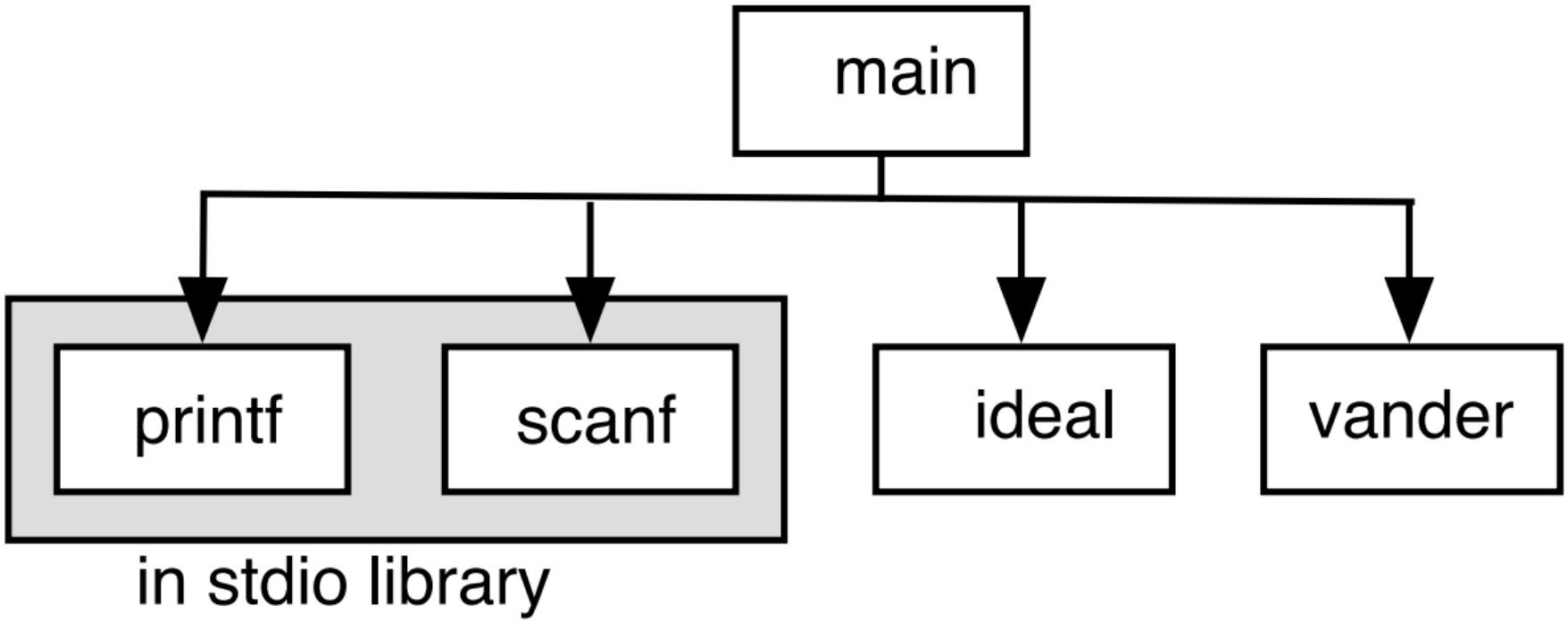



```
// ---------------------------------------------------------------
//  Pressure of CO gas in a tank, using Van der Waal's equation,
//  P = RT/(v-b) - a/(v*v).
float vander( float v )
{
    float p;                    // LOCAL declaration, not same p as above
    const float a = 1.474E+05;                  // constants for CO gas
    const float b = .0395;

    p = R * temp / (v - b) - a / (v * v);       // pressure in Pascals
    return p / 1000.0;                          // kPa pressure
}
```

Local Symbols and Visibility

- **Parameter names** are **local to the function** that defines them.
- A **local variable** is **declared inside a function**.
- **Parameters and local variables** are **created when a function is called and discarded when the function returns**. They are **not visible anywhere except within the function**.

This diagram shows the local and global symbols from the program on pages 257 – 8 of the text, pages 73 - 74 in Part 3. The global variable temp needs to be **changed into a local variable** in main() and passed as a parameter to ideal() and vander().



Function Call Graph for CO Gas Program

```
#include <stdio.h>
#define R  8314            // universal gas constant

float ideal( float v, float temp );
float vander( float v, float temp );

int main( void )
{
    // Local Variables ------------------------------------
    float temp;            // Temperature of CO gas
    float m;               // mass of CO gas, in kilograms
    float vol;             // tank volume, in cubic meters
    float vmol;            // molar specific volume
    float pres;            // pressure (to be calculated)

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );

    vmol = 28.011 * vol / m;                    // molar volume of CO gas
    pres = ideal( vmol, temp );    // pressure; ideal gas model
    printf( "\n The ideal gas at %.3g K has pressure "
            "%.3f kPa \n", temp, pres );
    pres = vander( vmol, temp );   // pressure; Van der Waal's model
    printf( " Van der Waal's gas has pressure "
            "%.3f kPa \n\n", pres );

    return 0;
```

Program with Global Variable Removed

Note: Temp is now passed as a parameter.

III. Basic Program Architecture

CHAPTER 9

Programmer's Responsibility

- I expect you to **learn and follow the design and coding standards** covered in the style guide, lecture and the book.
- You need to meet the **professional standard** of producing **understandable, maintainable code that is well structured** to allow for improvements over time.
- It is your job **to prove that the code works**. Therefore: you must **design your test cases to test every option in your program** adequately and include output showing this with your submission.

Your Programming Environment – The IDE

Please download and use either XCode (Mac) or Eclipse (Windows and Linux). Eclipse users need to get the CLang compiler or the Gnu C++ compiler (g++) not the stripped-down mingw compiler. Widows people will need to use cygwin.

Everything you turn in must be written entirely in standard C++ 11 standard or higher, and I must be able to compile it on my machine without modification. DO NOT include any Windows pathnames or any Microsoft-only header files! (~~conio.h, windows.h~~).

The Visual Studio IDE puts inappropriate things into your code and does not give good error comments. Students who use it inevitably have more trouble than others.

CodeBlocks does not supply the minimum level of support and diagnostics that you need. Don't waste your time on it.

Reference

For online reference, use one of these sites.

www.cplusplus.com

http://en.cppreference.com/w/

If you want a printed textbook, I recommend that you buy the book Herbert Schildt, C++: The Complete Reference. It serves as a reference manual for C++ syntax. as well as a basic text.

Do not use online sites such as Stack Overflow or Geeks for Geeks.

They contain notoriously bad advice and bad code that does not meet the standards of this course.

Three types of programming

People program for different reasons. Programming can be . . .

1. **One-time software**: code that solves a computational problem.

2. **Personal software** that you can reuse.

3. **Software Product Development**, building a **product to be used by others**, over a period of several years.

Problem solving

Desired properties of single-use programs for solving problems:

- Correct outputs from correct inputs
- Succinct expression of algorithm
- Simple development cycle

Beginning programming courses focus on solving small problems. Script programming is quickly written and, often, quickly forgotten.

Software Product Construction

Desired properties of software constructed for widespread use:

- Correct outputs from correct inputs
- Robust in face of bad inputs; reliable
- Economical in resource usage (time and space)
- Understandability and verifiability of code
- Security
- Ease of repurposing
- Ease of deployment
- Maintainability

Programming in the large

This course will help prepare you for advanced courses where you will **construct large-scale software**. Important characteristics of this type of software include:

- Interface design and quality is important
- Thousands of lines of code
- Potentially written by many programmers
- And maintained over the years by different programmers
- Over a large span of time
- Deployed on a large number of computers
- With different architectures and operating systems
- Interacting with foreign code and devices

Course Goal: Learn the Foundation of Quality C++ Programming

Quality Code should . . .

1. Meet the **specifications (requirements)**.

2. Be **correct and** should provide **evidence** to users to trust the correctness.

3. Be **easy to debug and find logic errors**.

4. Be portable.

Quality C++ programs are written according to **strict design standards**.

Course goals, continued

- **Follow instructions and** question them if you think they are wrong or unclear.
- Learn how to **get a big job done one module at a time**.
- Use a **reference manual**.
- Learn **clean design paradigms and practices**.
- Write **code that is concise, readable, and non-repetitive**.
- **Test, analyze, debug**, and **present your work in a professional manner.**
- **Become proficient at modern C++ programming**.

Tools to Achieve the Goals of this Course

1. **Privacy:** isolation of each class from the others.
2. **Type clarity:** use the correct type for everything. No more using 1 for a truth value.
3. A **good compiler** that identifies type errors and gives clear error comments.
4. A knowledge of how to **debug effectively** without a debugger by tracing code manually.
5. Carefully **designed test suites** and well documented test results.
6. The **ISO Standard C++ 11 or later software toolset**.

Five Must Haves for this course

1. **Use C++ input and output**, not C I/O for all assigned work.
2. **Don't use global variables**. If you think you need one, your program or class-design is probably defective. Ask for help.
3. **Test every line of code you write**. It is your job to prove to me that your entire program works. If I get a program without a test plan and output, I will assume that it does not compile. A program with a partial or inadequate test plan will be assumed to be buggy.
4. **Clean up after yourself**. Close files. Delete dynamic memory.
5. **Put your name** and the file name in comments **at the top of every file.**

Basic Design Principles and Standards

1. **Conform** to the specifications.
2. **Keep it simple**.
3. Make your **code readable**.
4. **Identify the expert** for each task in your program – the expert is a module in your program.
5. Maintain **data privacy**. Use delegation.
6. **Maximize data isolation**: use const wherever meaningful.
7. Know **what object has custody of dynamic memory areas** at all times.
8. **Avoid hardware- or OS-dependent code**.

Conform to the Specification

A product is worthless if it does not do what the client ordered.

1. In this course, PLEASE read the instructions. **Read all of the instructions before you begin to write code**. **Take the time to understand** what you are supposed to do.

2. **Code design takes time, but** it makes **code production much faster and easier**. **Take the time to understand** the class structure you are trying to implement.

3. In this course, the **assignments are designed to give you practice using a wide range of tools**. **Use the tools and techniques that are required.** You might find something different on the internet, but that is irrelevant. (You are wiser not to search for solutions “out there".)

4. **If you think my specification is wrong, ask.** Don't just ignore it.

Keep it Simple

CAN is not the same as SHOULD.

1. A **function is too long if it does not fit on your screen** all at once.
2. A **function is not simple if its logic is nested to level 4 or more**.
3. A function is **not simple** if I cannot comprehend its purpose and action in **30 seconds**.
4. **Very long and very short identifiers** are **not simple**.
5. A **name is not simple if you cannot pronounce it**.
6. A **class name that is the same as a variable name** is not simple. Avoid it.

Make your code readable

1. **Minimize the use of underscores** (this is not Python!).

2. **Avoid using variable names i, l, and o.** People's eyes may not see fine detail well or miss it. Use letters that are easy to see (j, k, n).

2. **Put a space** after every comma or semicolon. Donotwritecodethatlookslikethis.

3. **Use blank lines to divide code into paragraphs**. DO NOT USE THEM EVERYWHERE.

4**. Indent your code properly**. Xcode and Eclipse will do this for you. Eclipse users use **CTRL-A (select all), CTRL-I (fix indents).**

5. **Restrict code and comments to your screen width (120 columns).** Avoid names that are so long that you cannot do this.

6. **Put a divider-comment before the first line of every function**. Give purpose and describe parameters and return meaning here.

The Main Program

- **The main() function is not part of a class.** (Unlike Java.)
- Its prototype is one of these:

```
int main( void )
int main( int argc, char* argv[] )
```

- **Include the header file for your major class in this file**.
- The **main function should** have a **greeting** with program and your name and a **goodbye** thanking the user.

Tracing Code Execution

- **Tracing code** is a critical skill for **debugging and testing code**.
- **Use code tracing to examining your code** when you hit a bug or logic problem.
- Tracing code can also be **used to confirm that the code is working as expected.**
- This is what you should be doing in a **code walkthrough** when you look at code.
- In **pairs programming**, the passenger (non-driving programmer) should be mentally tracing the code and making sure it meets the requirements.

How to Trace Code Execution

- Three ways to trace code – you should know all three:
  - Trace your code **mentally**. You can use paper and pencil to keep track of the variables. This is **necessary to figure out logic when writing code**.
  - You can also trace your code's execution by **printing the variables of interest within your code** while running it with various test cases.
  - You can also **use the trace feature in your IDE**. This will print all the variable which can help you find hidden errors.

Unit Tests.

- **A Unit test should test every option for every line of code.**
  - You have to think about this and **plan it out with test cases that will do this**.
  - The exception to this rule is for **obscure system failures** that cannot be induced.
- A unit test should **call every function you wrote, one or more times**.
  - Choose a **sequence of calls that causes every line of code** in the function to be executed.

To Be Continued when we discuss classes