Advanced Programming
Topic 3: Program Organizing

Nguyen, Thien Binh
Mechatronics and Sensor Systems Technology
Vietnamese-German University

21 January 2024

Vietnamese - German University

Outline

1. Variables
   - Blocks and Local Variables
   - Global Variables
   - Static Variables
   - Extern Variables
2. Functions
   - Declaration and Definition
   - Default Arguments
   - Call by Value, Reference, Arrays
   - Return Types
   - Function Overloading
   - Recursive Functions
   - `inline` Functions
3. The Preprocessor
   - `#include` Directive
   - `#define` Directive
   - Conditional Compilation with `#ifdef`, `#ifndef`, `#elseif`, `#endif`
   - Header Files

Vietnamese - German University

Outline

1. Variables

Blocks and Local Variables

- A block: is whatever defined in between a pair of brackets {}
- A local variable:
  - is defined inside a block
  - has **block scope**, i.e., exists within that block only
  - has **automatic duration**, i.e., they are automatically allocated on the stack memory when defined, and deallocated when out of scope

```
int main()
{
  {
    int localVar = 10;
    cout << "Inside the block: localVar = "
         << localVar << endl;
  }
  cout << "Outside the block: localVar = "
       << localVar << endl;
  return 0;
}
```

Global Variables

- A global variable:
  - is defined outside any blocks, by convention, at the top of a file, below the includes, and above the `main`
  - has **file scope**, i.e., is available anywhere in the file where it is defined
  - has **static duration**, i.e., exists until the end of the program

```cpp
#include <iostream>
using namespace std;
int globalVar = 100;

int main()
{
  {
    globalVar = 10;
    cout << "Inside the block: globalVar = "
         << globalVar << endl;
  }
  cout << "Outside the block: globalVar = "
       << globalVar << endl;
  return 0;
}
```

Global Variables

- What if a local and global variable have the same name? What is the output inside and outside the block?

```
#include <iostream>
using namespace std;
int var = 100;

int main()
{
  {
    int var = 20;
    cout << "Inside the block: var = "
         << var << endl;
  }
  cout << "Outside the block: var = "
       << var << endl;
  return 0;
}
```

Vietnamese - German University

Global Variables

- What if a local and global variable have the same name? What is the output inside and outside the block?

```cpp
#include <iostream>
using namespace std;
int var = 100;

int main()
{
  {
    int var = 20;
    cout << "Inside the block: var = " << var << endl;
  }
  cout << "Outside the block: var = " << var << endl;
  return 0;
}
```

⇒ **The local variable is prioritized inside the block it is defined.**

Static Variables

- A static variable:
  - can be defined anywhere in a program
  - has file scope and static duration, just like a global variable
  - has its memory allocated fixed for the lifetime of the program
  - is initialized just ONCE, and has its value carried on to the next times when it is used
  - is commonly used to generate a unique ID for each generated object

```cpp
#include <iostream>
using namespace std;
int ObjCounting()
{
  static int objID = 0;
  return ++objID;    // starting with 1
}
int main()
{
  for (int i = 0; i < 5; ++i)
    cout << "Object ID = " << ObjCounting() << endl;
  return 0;
}
```

Extern Variables

- Question: A global variable has file scope, i.e., it is visible within the file it is defined. Then how can a global variable be referred to in **a different file**? For example, we want to use the same variable `x` in both `ExampleLinkage_File_1.cpp` and `ExampleLinkage_File_2.cpp` below.

```cpp
int x(5);    // global variable
```

Listing 1: ExampleLinkage_File_1.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
  // want to refer to x defined in ExampleLinkage_File_1.cpp
  // error: x is not defined!
  cout << "x = " << x << endl;
  return 0;
}
```

Listing 2: ExampleLinkage_File_2.cpp

Extern Variables

- Linkage: determines whether a variable can be referred to in multiple files.
- No linkage: variables without linkage are the local ones since they exist with the block they are defined only
- Internal linkage: variables with internal linkage can be visible within the file that they are defined, i.e., strictly has file scope. These include
  - `static` global variables
  - `const` variables
  - `in-line` functions
  - `typedef` names
  - enumerations

Extern Variables I

- External linkage: variables with external linkage can be visible within all the files of the program. These include
  - `non-static` global variables
  - `static` class members
  - `non-const` variables
  - functions
- In order for a variable with external linkage to be used, a *forward declaration* with keyword `extern` must be added to tell the compiler that the variable has been defined in a different file of the program.
- Since `extern` variables are visible in multiple files, they have **global scope**.
- `extern` variables has scope depending on where they are forward declared in the file.

Extern Variables II

```cpp
// global variables have external linkage
// no need to add "extern" here
int x(5);
int y;
```

Listing 3: ExampleLinkage_File_1.cpp

```cpp
#include <iostream>
using namespace std;
// forward declaration: this tells the compile that
// x has been defined in a different file
// in this file, x has file scope
extern int x;
int main()
{
cout << "x = " << x << endl;
{
  // forward declaration for y
  // y has local scope in this file
```

Extern Variables III

```
        extern int y;
        y = 10;
        cout << "y = " << y << endl;
    }
    //cout << "y = " << y << endl; NOT visible
    return 0;
}
```

Listing 4: ExampleLinkage_File_2.cpp

- Constants have internal linkage ⇒ adding `extern` where defining the constants to change their linkage

Extern Variables IV

```cpp
// global variables have external linkage
// no need to add "extern" here
int x(5);
int y;

// constants have internal linkage
// adding '"extern" is needed
extern const int c = 100;
```

Listing 5: ExampleLinkage_File_1.cpp

```cpp
#include <iostream>
using namespace std;
// forward declaration for x
extern int x;
// forward declaration for const c
extern const int c;
int main()
{
```

Extern Variables V

```cpp
  cout << "x = " << x << endl;
  cout << "c = " << c << endl;
  {
    // forward declaration for y
    extern int y;
    y = 10;
    cout << "y = " << y << endl;
  }
  return 0;
}
```

Listing 6: ExampleLinkage_File_2.cpp

Outline

2. Functions

Vietnamese - German University

Functions

- A function can be both declared and defined
  - Declaration:
    ```
    return_type function_name(type arg_1, type arg_2, ...,
    type arg_n);
    ```
  - Definition:
    ```
    return_type function_name(type arg_1, type arg_2, ...,
    type arg_n)
    {
      // function body;
      return result;
    }
    ```
- A function can be declared as many times as possible, but defined only ONCE.

Functions

- Why declarations?
  - For complicated programs with multiple files involved, declarations help code maintenance and modularity. A common coding practice is that all relevant functions are declared and collected in a header file which is included into another source file.
  - Function declarations play a role in defining classes in C++.
  - Declarations are in particular important for code packaging which makes it possible for shared libraries. The declarations collected in a header file play as the interfaces to the source codes where the functions are defined. These source codes can be pre-compiled for saving compile time or protecting copyrights, etc.
- Functions have external linkage, i.e., forward declarations with `extern` is needed if the functions have global scope

- Example: The following functions are declared, defined, and used in different files. 4 files involve in this program.

```
double  TriArea(double height_, double base_);
void Print(double result_);
```

Listing 7: FunctionDeclare.h

```
double  RecArea(double side1_, double side2_)
{
  double  area_;
  area_ = side1_ * side2_;
  return area_;
}


// TriArea is re-declared here. OK!
double  TriArea(double height_, double base_);
```

Vietnamese - German University

Listing 8: FunctionExtern.cpp

```
#include <iostream>
// Declarations of TriArea and Print
// are included here
#include "FunctionDeclare.h"
using namespace std;

double  TriArea(double height_, double base_)
{
  double  area_;
  area_ = 0.5*height_*base_;
  return area_;
}

void Print(double result_)
{
```

Functions III

```
  cout << "Result = " << result_ << endl;
}

// RecArea is re-defined here. Error!
/*
double  RecArea(double side1_, double side2_)
{
  double  area_;
  area_ = side1_ * side2_;
  return area_;
}
*/
```

Listing 9: FunctionDefine.cpp

Functions IV

```cpp
#include <iostream>
#include "FunctionDeclare.h"
// forward declaration for RecArea
extern double RecArea(double side1_, double side2_);
using namespace std;
int main()
{
  double  h(3.0), b(5.0);
  Print( TriArea(h, b) );
  Print( RecArea(h, b) );
  return 0;
}
```

Listing 10: FunctionMain.cpp

Vietnamese - German University

Functions V

```
CC      = g++
CFLAGS  = -g -Wall
LDFLAGS =
OBJS    = FunctionExtern.o  FunctionDefine.o FunctionMain.o
TARGET  = aaa

all: $(TARGET)

$(TARGET):  $(OBJS)
	$(CC) $(CFLAGS) -o  $(TARGET) $(OBJS) $(LDFLAGS)

clean:
	rm  -f  $(OBJS)
```

Listing 11: makefile_function

Default Arguments

- Default values of arguments can be set when a function is declared

```
double  TriArea(double height_ = 3.0, double base_ = 5.0)
{
  double  area_;
  area_ = 0.5*height_*base_;
  return area_;
}

int main()
{
  Print( TriArea() );
  Print( TriArea(2.0) );
  Print( TriArea(0.5, 2.0) );
  return 0;
}
```

Vietnamese - German University

Call by Value

- If a variable is in the argument list, a copy of it this variable is created locally within the scope of the function

```cpp
#include <iostream>
using namespace std;

double  square(double x_)
{
  cout << "&x_ = " << &x_ << endl;
  x_ = x_ * x_;
  return x_;
}

int main()
{
  double x(10);
  cout << "&x = " << &x << endl;
  cout << "square(x) = " << square(x) << endl;

  return 0;
}
```

Call by Value

- Modification of a local variable in a function does not change the original variable where the function was called

```
#include <iostream>
using namespace std;
void donothing(double x)
{
  x = x * x;
}
int main()
{
  double x(10);
  donothing(x);
  cout << "x = " << x << endl;
  return 0;
}
```

- Calling by value for large objects is usually expensive (running time and memory allocation) due to the copying process of local variables

Vietnamese - German University

Call by Reference

- If a reference or a pointer is in the argument list, a copy of this reference or pointer pointing to the same variable, i.e., the same memory location, is created
- Modification of a local variable changes the original variable where the function was called since the the reference or pointer points to the same memory location of the original variable

Vietnamese - German University

Call by Reference I

```
#include <iostream>
using namespace std;

// call by reference
void setArg_ref(double& x_)
{
  cout << "&x_ = " << &x_ << endl;
  x_ = 100.0;
}

// call by pointer
void setArg_ptr(double* x_)
{
  cout << "x_ = " << x_ << endl;
  *x_ = 100.0;
}

int main()
{
```

Call by Reference II

```cpp
    double x;
    x = 10.0;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;

    // pass by value
    setArg_ref(x);
    cout << "x = " << x << endl;
    // pass by address
    setArg_ptr(&x);
    cout << "x = " << x << endl;

    return 0;
}
```

Call by Reference I

- Using references or pointers, a function can return multiple variables

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI(3.141592653589793238462643383279502884);

void  Polar2Cartesian(double& r_, double& theta_,
                 double& x_, double& y_)
{
   cout << "&x_ = " << &x_
       << ", &y_ = " << &y_ << endl;
   cout << "&r_ = " << &r_
       << ", &r_ = " << &r_ << endl;
   x_ = r_ * cos(theta_);
   y_ = r_ * sin(theta_);
}

int main()
```

Call by Reference II

```
{
    double r(3.5), theta(PI/3.0);
    double x, y;
    Polar2Cartesian(r, theta, x, y);
    cout << "&x = " << &x
          << ", &y = " << &y << endl;
    cout << "&r = " << &r
          << ", &r = " << &r << endl;
    cout << "(r, theta) = " << r
          << ", " << theta << endl;
    cout << "(x, y) = " << x
          << ", " << y << endl;

    return 0;
}
```

Call by Reference

- In case one does not want the original variables to be modified, `const` can be used. This is commonly used for large objects passed as arguments into functions, for example, *r_* and *theta_* are supposed not to be modified since they are input parameters. In this case, we can change to function as

```
void  Polar2Cartesian(const double& r_, const double& theta_,
                            double& x_, double& y_)
{
  r_ = 10.0; theta_ = 0.5*PI; // r_, theta_ cannot be modified
  cout << "&x_ = " << &x_
      << ", &y_ = " << &y_ << endl;
  cout << "&r_ = " << &r_
      << ", &r_ = " << &r_ << endl;
  x_ = r_ * cos(theta_);
  y_ = r_ * sin(theta_);
}
```

- Call by reference is *cost efficient* since no copies of large objects are needed to pass through function interfaces

Call by Array I

- Static allocated arrays can be passed into a function by using square brackets without specifying the exact number of array elements, e.g., `v[], A[][]`
- Pointers are used for dynamically allocated arrays, e.g., `*v` for 1D arrays, and `**A` for 2D arrays
- Example: function to compute the dot product of two vectors of the same size

```
#include <iostream>
#include <cmath>
using namespace std;

// to compute the dot product of 2 vectors of size
double DotProd(const int& size,
          const double v[],
          const double w[])
{
```

Call by Array II

```
  double dot(0.0);
  for (int i = 0; i < size; ++i)
    dot += v[i] * w[i];

  return dot;
}

int main()
{
  int size(4);
  double v[size] = {1, 2, 3, 4}
  double w[size] = {4, 3, 2, 1};
  cout << "dot(v, w) = " << DotProd(size, v, w) << endl;

  return 0;
}
```

Function Return

- Similarly to the input arguments, a function can return either by value, reference, or pointer, or nothing (void)
- Example: functions to allocate and de-allocate a vector

```
// return by pointer
double* allocateVec(const int& numCols)
{
  double* v;
  v = new double[numCols];
  return v;
}

// void return
void deallocateVec(double* v_)
{
  delete[] v;    // for arrays
}
```

Function Return I

- Example: Write a functions to set and get the value for each entry of a vector

Vietnamese - German University

Function Return I

- Example: Write functions to set and get the value for each entry of a vector

```cpp
#include <iostream>
using namespace std;

// return by reference
// allow to modify the returned variable
double& setVal(double v[], const int& index)
{
  return v[index];
}

// return by value
// does not allow to modify the returned variable
double getVal(double v[], const int& index)
{
  return v[index];
}
```

Function Return II

```cpp
// void return
void    printVec(double v[], const int& size)
{
  for (int j = 0; j < size; ++j)
  cout << getVal(v,j) << ", ";
  cout << endl;
}
int main()
{
  int size(3);
  double v[size];
  setVal(v, 0) = 1.0;
  setVal(v, 1) = 2.0;
  setVal(v, 2) = 3.0;
  printVec(v, size);
  return 0;
}
```

Vietnamese - German University

Function Return

- Example: What is wrong in the following function?

```cpp
#include <iostream>
using namespace std;
// return by reference
double& somethingWrong(const double& x)
{
  double y;
  y = x + 2;
  return y;
}
int main()
{
  cout << "Result = " << somethingWrong(10.0) << endl;
  return 0;
}
```

Vietnamese - German University

Function Return

- Example: What is wrong in the following function?

```
#include <iostream>
using namespace std;
// return by reference
double& somethingWrong(const double& x)
{
  double y;
  y = x + 2;
  return y;
}
int main()
{
  cout << "Result = " << somethingWrong(10.0) << endl;
  return 0;
}
```

⇒ *Return the reference of a local variable (y) which has been destroyed when the function is returned!*

Function Overloading

- In C++, it is possible that a same function is declared and defined many times with different bodies. This is known as function overloading
- Over loaded functions must be distinguished one another by having different number of arguments or argument types
- Example: Write function `add` which do the summation of either two scalars or vectors. Use function overloading with two definitions of the same function `add`

Function Overloading I

- Example: Write function add which do the summation of either two scalars or vectors. Use function overloading with two definitions of the same function add

```
#include <iostream>
using namespace std;

double* allocateVec(const int& numCols)
{
  double* v;
  v = new double[numCols];
  return v;
}

void deallocateVec(double* v)
{
  delete[] v;    // for arrays
}
```

Vietnamese - German University

Function Overloading II

```cpp

void printVec(const int& numCols_, double* v_)
{
  for (int j = 0; j < numCols_; ++j)
     cout << v_[j] << ", ";
  cout << endl;
}

double add (double x1, double x2)
{
  return x1 + x2;
}

// add two scalars
void  add(const double& alp1, const double& alp2, double& beta)
{
  beta = alp1 + alp2;
}

```

Function Overloading III

```
// add two vectors
void  add(const int& length, const double* v1, const double* v2
{
  for (int i = 0; i < length; ++i)
    w[i] = v1[i] + v2[i];
}

int main()
{
  int length(5);
  double alp1(10), alp2(20), beta;
  double *v1, *v2, *w;

  v1 = allocateVec(length);
  v2 = allocateVec(length);
  w = allocateVec(length);

  for (int i = 0; i < length; ++i)
  {
```

Vietnamese - German University

Function Overloading IV

```
    v1[i] = i;
    v2[i] = 2.0*i;
  }

  add(alp1, alp2, beta);
  add(length, v1, v2, w);

  cout << "beta = " << beta << endl;
  printVec(length, w);

  deallocateVec(v1);
  deallocateVec(v2);
  deallocateVec(w);

  return 0;
}
```

VGU Vietnamese - German University

Recursive Functions

- Recursion is that a function calls itself, and the corresponding function is called a recursive function
- A base case must be specified in a recursive function
- Example: Write a function to compute the factorial of an non-negative integer

Recursive Functions I

- Example: Write a function to compute the factorial of an non-negative integer

```cpp
#include <iostream>
using namespace std;

int Fact(int n)
{
  int x;
  // base case: must be specified
  if (n < 0)
    cout << "Err:␣n␣must␣be␣greater␣than␣or␣equal␣zero!"
         << endl;
  else if (n == 0 || n == 1)
    x = 1;
  else
    // recursion: function calls itself
    x = n * Fact(n - 1);
  return x;
```

Recursive Functions II

```
}

int main()
{
  cout << "Fact(5) = " << Fact(5) << endl;
  return 0;
}
```

`inline` Functions

- **Overhead cost:** whenever a function is called, the program needs to
  - store the address of the current statement it is executing
  - copy, allocate the memory, and assign values to the input arguments of the function
  - jump to the new memory location allocated for the function execution
  - etc.

  $\Rightarrow$ *This overhead cost is significant for small functions!*
- `inline` functions: having their contents substituted directly to the code at run time $\Rightarrow$ *No overhead cost!*

Vietnamese - German University

- Example: Replacing functions setVec, getVec, and printVec above with inline versions in a header file.

```
#include <iostream>
using namespace std;
// return by reference
// allow to modify the returned variable
inline double& setVal(double v[], const int& index)
{
  return v[index];
}

// return by value
// does not allow to modify the returned variable
inline double getVal(double v[], const int& index)
{
  return v[index];
}

```

inline Functions II

```
// void return
inline void   printVec(double v[], const int& size)
{
  for (int j = 0; j < size; ++j)
  cout << getVal(v,j) << ", ";
  cout << endl;
}
```

Listing 12: ExampleFunction_inline.h

```
#include "ExampleFunction_inline.h"
int main()
{
  int size(3);
  double v[size];
  setVal(v, 0) = 1.0;
  setVal(v, 1) = 2.0;
  setVal(v, 2) = 3.0;
  printVec(v, size);
```

inline Functions III

```
  return 0;
}
```

Listing 13: ExampleFunction_inline.cpp

- In run time, these inline functions are directly substituted into the code.

```
#include "ExampleFunction_inline.h"
int main()
{
  int size(3); double v[size];
  v[0] = 1.0; v[1] = 2.0; v[2] = 3.0;
  for (int j = 0; j < size; ++j)
    cout << v[j] << ", ";
  cout << endl;
  return 0;
}
```

Vietnamese - German University

`inline` Functions

- Since inline functions have internal linkage, it is a common coding practice that they are defined in header `.h` files so that they are always copied into the source files when being used.
- Inline functions usually increase the size of the generated code but decrease the execution time (no overhead cost). Thus, inline functions are best suited for short functions only.

Vietnamese - German University

Outline

3. The Preprocessor

The Preprocessor

- Prior to compilation, the code goes through a phase known as *translation* in which a *preprocessor* takes place.
- The preprocessor ignores all code contents but looks for special directives starting with `#` and makes appropriate changes/substitutions
- The following directives are noteworthy: `#include`, `#define`, and the conditional compilation directivesñ `#ifdef`, `#ifndef`, `#elseif`, `#endif`

Vietnamese - German University

#include Directive

```
#include <iostream>
#include <cmath>
#include <cassert>
#include "user-header-file.h"
```

- When the preprocessor scans and finds the `#include`, it will replace the directive by all the preprocessed contents of associate header file.
- A `< >` bracket is used for standard ANSI C++ libraries, e.g., `iostream`, `cmath`, or `cassert`, whereas a quotation `" "` is used for user-defined header files.
- The `#include` directive is mainly used to substitute header files `.h` into source files `.cpp`

Vietnamese - German University

#include Directive

```cpp
double  TriArea(double height, double base);
void Print(double result);
```

Listing 14: ExampleDeclare.h

```cpp
#include <iostream>
#include "ExampleDeclare.h"
using namespace std;

double  TriArea(double height, double base)
{
  double  area;
  area = 0.5*height*base;
  return area;
}

void Print(double result)
{
  cout << "Result = " << result << endl;
}
```

- ExampleDeclare.h and ExampleDefine.cpp are equivalent to

```
// all preprocessed contents of
// /usr/include/g++/iostream
// the contents of ExampleDeclare.h
double  TriArea(double height, double base);
void Print(double result);
using namespace std;

double  TriArea(double height, double base)
{
  double  area;
  area = 0.5*height*base;
  return area;
}

void Print(double result)
{
  cout << "Result = " << result << endl;
}
```

Vietnamese - German University

#define Directive

```
#define IDENTIFIER tokens
```

- When a processor scans and finds #define, it will textually substitute all occurrences of IDENTIFIER with tokens
- The #define directive is mostly used for defining and giving meaningful names for global constants

```
#define PI          3.14159265358979323846264338
#define LIGHTSPEED  2.997925e8
#define ZERO        0.00000000000000000000000000
```

#define Directive

- The #define directive can also be used to define simple functions, e.g.,

```
#define SQUARE(X) ((x) * (X))
```

then

```
y = SQUARE(4.0);
```

is equivalent to

```
y = ( (4.0) * (4.0) );
```

- It is always considered a better practice to use const to define constants instead of #define (see Lecture 2)

Vietnamese - German University

Conditional Compilation

```c
#define CONDITION_1

#ifdef CONDITION_1
  // code segment 1
#endif

#ifndef CONDITION_2
  // code segment 2
#endif
```

- `#ifdef`, `#ifndef`, `#elseif`, `#endif` can be used to determine which part of the code is going to be compiled and which is not.
- `code segment 1` will be compiled if `CONDITION_1` is defined. On the contrary, `code segment 2` will be compiled if `CONDITION_2` is *not* defined.

Vietnamese - German University

Conditional Compilation

- Example: What is printed out to the screen?

```
#include <iostream>
using namespace std;
#define COMPILE
void printsomething()
{
  #ifdef COMPILE
    cout << "code segment 1" << endl;
  #endif

  #ifndef COMPILE
    cout << "code segment 2" << endl;
  #endif
}

int main()
{
  printsomething();
  return 0;
}
```

Conditional Compilation

- Example: What is printed out to the screen? Why is that?

```
#include <iostream>
using namespace std;
void printsomething()
{
  #ifdef COMPILE
    cout << "code segment 1" << endl;
  #endif

  #ifndef COMPILE
    cout << "code segment 2" << endl;
  #endif
}

#define COMPILE
int main()
{
  printsomething();
  return 0;
}
```

Conditional Compilation

- Example: What is printed out to the screen? Why is that?

```cpp
#include <iostream>
using namespace std;
void printsomething()
{
  #ifdef COMPILE
    cout << "code segment 1" << endl;
  #endif
  #ifndef COMPILE
    cout << "code segment 2" << endl;
  #endif
}
#define COMPILE
int main()
{
  printsomething();
  return 0;
}
```

⇒ *A preprocessor* ignores all code contents or sequences but looks only for directives from top to bottom of the code.

Conditional Compilation

- The `#ifdef`, `#ifndef`, `#endif` together with `#define` directive are of particularly useful in creating header guards for header files which prevents multiple definition.

Header Files

- A C++ files are basically classified into 2 types:
  - source files with the extension `.cpp` which contain all variables, functions, and classes definitions,
  - header files with the extension `.h` in which functions and classes are declared. Short `inline` functions and global constants may also be defined in header files.
- Header files are included into source files with the `#include` directive.
- Using header files enhance code readability and abstraction since users could justify the use of, e.g., a class, by inspecting its member data and methods declared in the header file
- Header files also serve as the interface for packaged libraries. It is common that a shared C++ library has its source files precompiled for the reason of security or copyrights, and users just need to include the library's header file in order to use it.

VGU Vietnamese - German University

Header Files I

- Although a header file can be included in as many files as wanted, this could
  - increase the overhead cost as a preprocessor has to substitute all the contents of the header file at the inclusion location
  - return errors if there are variables or non-inline functions defined more than once.
- Example: What is wrong with the following code?

```
#include <iostream>
using namespace std;

// global variable;
double area;

// functions
double recArea(const double& side1, const double& side2);
void printArea()
{
```

Header Files II

```
  cout << "The area is " << area << endl;
}
```

Listing 16: RecAreaDeclared.h

```
#include "RecAreaDeclared.h"

// definition for RecArea
double recArea(const double& side1, const double& side2)
{
  return side1 * side2;
}
```

Listing 17: RecAreaDefined.h

Header Files III

```cpp
#include <iostream>
#include "RecAreaDeclared.h"
#include "RecAreaDefined.h"
using namespace std;
int main()
{
  double side1(5), side2(10);
  area = recArea(side1, side2);
  printArea();
  return 0;
}
```

Listing 18: RecAreaMain.cpp

Vietnamese - German University

- Example: What is wrong with the following code? $\Rightarrow$ *area and recArea are defined twice.*
- Substituted code:

```
#include <iostream>
using namespace std;

//=== from RecAreaDeclared.h
// global variable;
double area;

// functions
double recArea(const double& side1, const double& side2);
void printArea()
{
  cout << "The area is " << area << endl;
}

```

Vietnamese - German University

Header Files II

```
//=== from RecAreaDefined.h
// global variable;
double area;

// functions
double recArea(const double& side1, const double& side2);
void printArea()
{
  cout << "The area is " << area << endl;
}

// definition for RecArea
double recArea(const double& side1, const double& side2)
{
  return side1 * side2;
}
```

Header Files III

```
int main()
{
  double side1(5), side2(10);
  area = recArea(side1, side2);
  printArea();
  return 0;
}
```

Header Files I

- Header Guards: to prevent multiple definitions of the same variable or function, or unnecessary inclusion of header files.
- Guarded header files:

```cpp
#ifndef _RECAREA_DECLARED_     // header guard
#define _RECAREA_DECLARED_     // header guard
#include <iostream>
using namespace std;
// global variable;
double area;
// functions
double recArea(const double& side1, const double& side2);
void printArea()
{
  cout << "The area is " << area << endl;
}
#endif
```

Listing 19: RecAreaDeclared.h

```
#ifndef _RECAREA_DEFINED_ // header guard
#define _RECAREA_DEFINED_ // header guard

#include "RecAreaDeclared.h"

// definition for RecArea
double recArea(const double& side1, const double& side2)
{
  return side1 * side2;
}

#endif
```

Listing 20: RecAreaDefined.h

Header Files III

```cpp
#include <iostream>
#include "RecAreaDeclared.h"
#include "RecAreaDefined.h"
using namespace std;
int main()
{
  double side1(5), side2(10);
  area = recArea(side1, side2);
  printArea();
  return 0;
}
```

Listing 21: RecAreaMain.cpp

Vietnamese - German University

Header Files

- Header Guards: to prevent multiple definitions of the same variable or function, or unnecessary inclusion of header files.
- `RecAreaDeclared.h`: initially, since `_RECAREA_DECLARED_` was not defined, the whole file will be compiled due to the `#ifndef` directive which defines condition `_RECAREA_DECLARED_`. When included for the second time, since `_RECAREA_DECLARED_` has been defined, the whole file is ignored.
  $\Rightarrow$ *No matter how many times `RecAreaDeclared.h` is included, the file is in fact compiled just ONCE.*

Vietnamese - German University

Reading

1. Capper, *Introducing C++ for Scientists, Engineers, and Mathematicians*, **Chapter 5**
2. Pitt-Francis, and Whiteley, *Guide to Scientific Computing in C++*, **Chapter 5**