Course overview

Inst. Nguyễn Minh Huy

Contents

- Coding convention.
- Function overloading.
- Error handling.
- Abstract programming.

Contents

- **Coding convention.**
- Function overloading.
- Error handling.
- Abstract programming.

Coding convention

- Why using coding convention?
  - Case 1 - Working alone:
    - Self understanding.
    - **Understand your stuff last year?**

  - Case 2 - Team-work:
    - Work shared among members.
    - **How to gather work?**
    - **How to understand each other?**

For effective collaboration!!

Apply discipline!!

Coding convention

- **Rule #0: no universal standard!!**
  - Depend on programming languages.
  - Depend on companies/ communities.
- Common conventions:
  - Naming convention.
  - Statement convention.
  - Comment convention.
- *References:*
  - *Course Coding Conventions on Moodle.*
  - *Google C++ Styles Guide: https://google.github.io/styleguide/cppguide.html*

Contents

- Coding convention.
- **Function overloading.**
- Error handling.
- Abstract programming.

Function overloading

- Function signature/prototype:
  - To identify a function.
  - C/C++ function signature:
    - Function name.
    - Argument list.

```
double sort( int a[ ], int size );
double sort( float a[ ], int size );
```

    - **Return type is not counted!!**

Function overloading: functions differ only by argument list.

Function overloading

- Invalid function overloading?

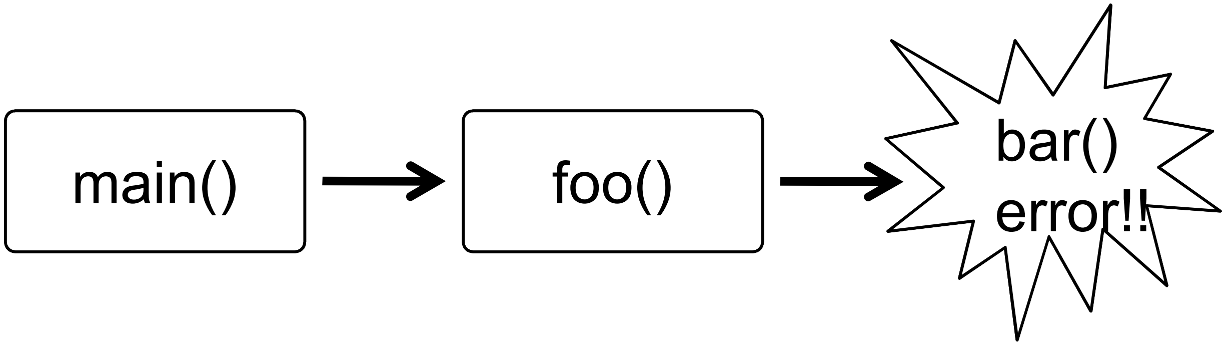
1. int add(int a, int b);
2. int add(int x, int y);
3. int add(int a, float b);
4. float add(int u, int v);
5. int add(int a, long b);

Contents

- Coding convention.
- Function overloading.
- **Error handling.**
- Abstract programming.

Error handling

- Basic concept:
  - Errors are unexpected cases in code.
  - Errors can come from:
    - User input.
    - External data.
    - Previous code result.
  - Code must be guided to due with predictable errors:
    - Resolve in-place.
    - Raise error, handle later.



Error handling

- C-style error handling:
  - Function return:
    - Simple.
    - Interfere function signature.

```
bool divide( int a, int b, int &result ) {
    if (b == 0)
        return false;  // Raise error.
    result = a / b;
    return true;
}
```

```
int main( ) {
    int  x, y, z;

    // Input x, y…

    if ( !divide( x, y, z ) )  // Handle.
        fprintf(stderr, “Division error”);
    else
        printf(“Result = %d”, z);
}
```

Error handling

- C-style error handling:
  - Use **errno** (library <errno.h>):
    - Before function call: set **errno** = 0.
    - After function call: check **errno**.

```
int divide( int a, int b ) {
    if (b == 0) {
        errno = 1;  // Raise error.
        return 0;
    }
    return a / b;
}
```

```
int main( ) {
    int  x, y, z;

    // Input x, y…

    errno = 0;
    z = divide( x, y );
    if ( !errno )  // Handle.
        fprintf(stderr, “Division error”);
    else
        printf(“Result = %d”, z);
}
```

Error handling

- C++ exception handling:
  - Raise error: **throw** <error value>.
    - <error value>: simple or complex type.
  - Receive and handle:
    - **try** { <examined code> }
    - **catch** (<error value>) {
          <handling code>
      }

```
int divide( int a, int b ) {
    if (b == 0)
        throw 1;  // Raise error.

    return a / b;
}
```

```
int main( ) {
    int  x, y, z;

    // Input x, y…

    try {  // Receive.
        z = divide( x, y );
        printf(“Result = %d”, z);
    }
    catch (int &e) {  // Handle.
        fprintf(stderr, “Division error”);
    }
}
```

Error handling

- C++ exception handling:
  - Built-in exception type:
    - std::invalid_argument.
    - std::out_of_range.
    - std::overflow_error.
    - std::bad_alloc.

```
int * create_array( int length ) {
    if ( length <= 0 )
        throw std::invalid_argument(“Error: length is negative or zero”);

    try {
        return new int [ length ] { 0 };
    catch ( std::bad_alloc &e ) {
        return nullptr;
    }
}
```

Error handling

- Multiple handlers:

```
// Ugly nested if-else.
int main() {
    if ( f1() ) {
        if ( f2() ) {
            if ( f3() ) {
                // Expected case.
            } else {
                // Handle f3 error.
            }
        } else {
            // Handle f2 error.
        }
    } else {
        // Handle f1 error.
    }
}
```

```
// Handle errors first.
int main( ) {
    if ( !f1() ) {
        // Handle f1 error.
        return;
    }
    if ( !f2() ) {
        // Handle f2 error.
        return;
    }
    if ( !f3() ) {
        // Handle f3 error.
        return;
    }
    // Expected case.
}
```

```
// Use C++ exception.
int main( ) {
    try {
        f1();
        f2();
        f3();
        // Expected case.
    } catch (<f1 error>) {
        // Handle f1 error.
    } catch (<f2 error>) {
        // Handle f2 error.
    } catch (<f3 error>) {
        // Handle f3 error.
    }
}
```

Contents

- Coding convention.
- Function overloading.
- Error handling.
- **Abstract programming.**

Abstract programming

- What is an abstract program?
  - Not fix to a specific case.
  - Can apply to different context.
  - Write once, use everywhere.
  - Parameterization:
    - Data parameterization (passing arguments).
    - Type parameterization (function template).
    - Process parameterization (function pointer).

Abstract programming

- Data parameterization:
  - Problem: add two integers 12345 and 67890.
  - Abstraction 1: add two integers of ***any values***.

```
// Fix to specific case:
// Hard code.
int calc( ) {
        return 12345 + 67890;
}
```

```
// Abstraction 1:
// Passing arguments.
int calc( int x, int y ) {
        return x + y;
}
int main( ) {
        calc( 12345, 67890 );
        calc( 49381, 97723 );
}
```

Abstract programming

- Type parameterization:
  - Abstract 2: add two numbers of ***any types***.

    ➔ Use Function Template.

```
// Abstraction 2:
// Passing type as arguments.
template <class T>
T  calc( T  x, T  y ) {
        return x + y;
}
```

```
int main( ) {
        int x = calc( 3, 5 );

        float y = calc( 4.2, 6.3 );

        Fraction  p1, p2;
        Fraction  p3 = calc( p1, p2 );
}
```

Abstract programming

- Function Template:
  - A way to parameterize type.
  - Pass type as arguments into function.
- Notes:
  - Keyword **“class”** can be replaced by **“typename”**.
  - Template declaration must be in both function declaration and implementation.
  - Function implementation must be in the same file:
    - With function declaration.
    - With calling function.
  - Template FAQ: https://isocpp.org/wiki/faq/templates

Abstract programming

- Process parameterization:
  - Abstraction 3: do **any operation** on two integers.

    ➔ Use function pointer.

```
// Abstraction 3:
// Passing function as arguments.
typedef int ( * Operation )( int, int );
int  calc( int  x, int  y, Operation p ) {
    x = x * x;
    y = y * y * y;
    return p( x, y );
}
```

```
int add( int x, int y ) {
        return x + y;
}
int mul( int x, int y ) {
        return x * y;
}
int main( ) {
        int x = calc( 3, 5, add );
        int x = calc( 4, 6, mul );
}
```

Abstract programming

- Function pointer:
  - S1: use **typedef** to create alias for function pointer.

```
typedef int (* Operation) ( int, int );
```

  - S2: pass function as arguments.

```
int calc( int x, int y, Operation p ) {
        x = x * x;
        y = y * y * y;
        return  p( x, y );
}
```

  - S3: create concrete function.

```
int add( int x, int y ) {
        return x + y;
}
```

  - S4: pass concrete function as argument:

```
calc( 3, 5, add );
```

Abstract programming

- Function pointer:
  - A way to parameterize command.
  - Pass function as argument into another function.
- Notes:
  - Pass function pointer directly, no **typedef**:

    ```
    int  calc( int  x, int  y, int ( *p ) ( int, int ) );
    ```

  - Function pointer with function template must be passed directly.

  ➔ Abstraction 4: do any operations on two numbers of any types.

Summary

- Coding Convention:
  - Apply discipline to collaborate efficiently.
  - Naming convention.
  - Statement convention.
  - Comment convention.
- Function Overloading:
  - Functions differ only by argument list.
- Error handling:
  - Errors: unexpected cases.
  - Code must be guided to due with errors.

Summary

- Error handling:
  - Raise error, handle later:
    - C-style: return type, errno (<errno.h>).
    - C++style: throw, try, catch.
- Abstract programming:
  - Not fix to specific case.
  - Use parameterization:
    - Data parameterization (passing arguments).
    - Type parameterization (function template).
    - Command parameterization (function pointer).

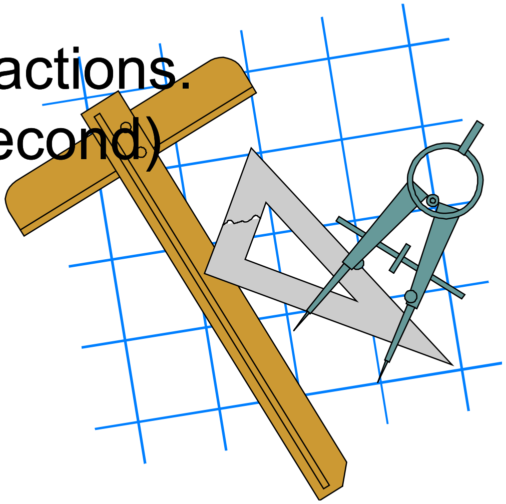
Practice

- Practice 1.1:

  Write C++ program to do the followings on type **Fraction**:

  (use appropriate error handling)

  - **input**: enter fraction from keyboard.
  - **output**: print fraction to screen.
  - **reduce**: return the reduction of fraction.
  - **inverse**: return the inversion of fraction.
  - **add**: return the sum of two fractions.
  - **compare**: return the comparison result of two fractions.
    (0: first = second, -1: first < second , +1: first > second)

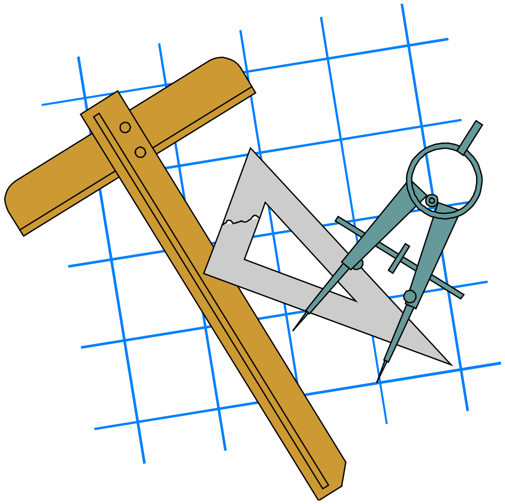


Practice

- ## Practice 1.2:

  Write C++ function to sort an array of **Fraction**, the sort criteria can be customized by user.

  Hint: void **sort** (**Fraction** *arr, int size, **<sort criteria>**)



Practice

- ## Practice 1.3:

  Upgrade function in practice 1.2, so that it can sort an array of elements **of any types**, the sort criteria can also be customized by user.

  Hint: void **sort** (**<any type>** *arr, int size, **<sort criteria>**)