Data Flow & Control Structures

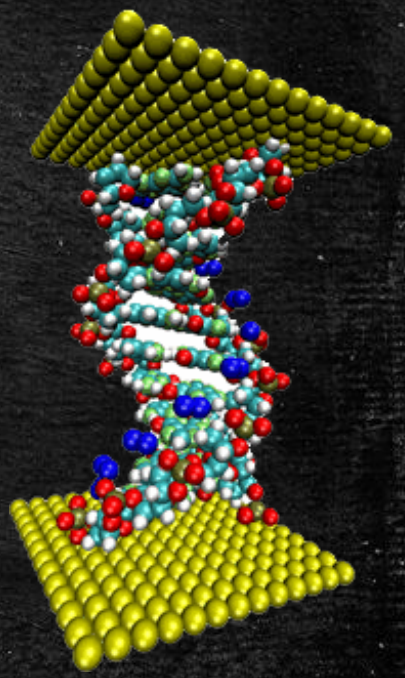
CS 150 – C++ Programming I
Lecture 9

Review: Separate Compilation & Testing

- 1. Create the client (or tester) program
  - Write 3 *assertEquals* tests for both *min* and *max*
  - Build and screenshot the error message
- 2. Create the header file
  - Remember header guards
  - Add documentation
  - Build and screenshot the error message
- 3. Create the implementation
  - Test & shoot the final screenshots

Function Signatures

- In languages like C, function names must be unique
  - Required inventing new names to do the same work
  - In the standard C library: `abs()`, `fabs()`, `labs()`
- In C++, two functions may have the same name, if:
  - The number of arguments they require differs, or…
  - …the type or order of their arguments differ
- The combination of method name, parameter type, number and order is called the function signature
  - Acts like "DNA" to uniquely identify a particular function



Function Overloading

- Overloaded functions have the same name, but a different signature and body
  - ```
    double f(int);
    double f(int, double);
    double f(double, int);
    double f(double, double, double);
    ```

```
177 T _Z1fddd
15b T _Z1fdi
129 T _Z1fi
13f T _Z1fid
```

- When compiled, the name is combined with the signature to produce a mangled name
  - Used internally by compiler and linker to uniquely identify a particular function
```

Overload Resolution

- When you call a function, the compiler must have some way of telling exactly which function you mean
- Resolving overloaded functions:
  - 1) Candidate set: same name
  - 2) Viable set: correct number of convertible args
  - 3) Exact type matches
  - 4) Partial matches
  - 5) Conversions
- Ambiguity: more than one viable conversion

Question

```
g(1, 2);
```

- Consider this function call. Which of these overloaded functions will be invoked
  - A. `int g(int count, double value);;`
  - B. `void g(double value, int count);`
  - C. `float g(int value, int count);`
  - D. `void g(double value, double val);`
  - E. Ambiguous. Compiler cannot decide

Question

```
fn(1.0, 2.0, 3.5);
```

- Consider this function call. Which of these overloaded functions is selected?
  - A. `void fn(int, double, double&);`
  - B. `void fn(int, int, double&);`
  - C. `void fn(int, int, double);`
  - D. `void fn(int, int, int);`
  - E. Ambiguous. Compiler cannot decide

Default Arguments

- Function with mandatory and optional arguments
  - `double f(int a, int b=3, int c=4);`
- Call the function in three different ways:
  -
```
f(7);        // b is 3 and c is 4
f(7, 2);     // c is 4
f(8, 5, 3); // all arguments supplied
```
- No mandatory parameter after an optional. No references.
  - `double f(int a=1, int b, int c=4);`
  - `double f(int& a, int& b=3, int c=4);`

Try It Yourself. Get this to Work.

```
int main()
{
    // With no newlines or decimals
    print("Hello");
    print(acos(-1.0));
    cout << endl;
    cout << "Expected: Hello3.14159" << endl;

    // With a newline
    print("Goodbye", true);
    cout << "Expected: Goodbye" << endl;

    // With a newline and 17 decimal places
    print(acos(-1.0), true, 17);
    cout << "Expected: 3.14159265358979312" << endl;
}
```

Parameter Categories & Data Flow

- Input parameters: information flows only into the function
  -
```
n = sqrt(a);    // a flows into sqrt()
n = sqrt(9.5);  // can use literal
```
- Output parameters: supply a variable, the function fills it
  -
```
char ch;
bool ok = cin.get(ch); // ch flows out
cin.get('A');          // cannot use a literal
```
- Input-Output parameters: data flows both ways
  -
```
swap(a, b); // variables a and b are modified
```

A Checklist for Declaring Parameters

- Output and input-output parameters
  - Always pass by reference (`&` after type) all type categories
  - Documentation tags: `@param[in, out]` or `@param[out]`
- Input parameters depend on type of input
  - Library types (including *`string`*) pass by `const` reference
    - Do this for efficiency. Uses less memory and is faster
  - Built-in types (`int`, `double`, `char`), pass by value
    - Do not use `const` or `&` for built-in-type input parameters

An Alternative Parameter Checklist

- Is the parameter a primitive or built-in type?
  - Does the function change the argument? Pass by reference.
    - `void swap(int& lhs, int& rhs);`
  - Does not change argument? Use pass by value
- Object types? Always use pass-by-reference
  - Is the argument unchanged? Use `const` reference

MEMORIZE THESE CHECKLISTS