Lesson #2: Program Structure, Functions, and Variables

2.1 Separating Interface from Implementation

- Two clients of a class
  - User
    - Uses public interface
    - Creates and uses objects
    - Doesn't care about implementation of class
  - Implementor
    - Cares about entire class
- Best practice:
  - Put Interface in .h, implementation in .cpp

2.2 Defining Classes in a Namespace

- Best practice:
  - Place all code in a namespace
- Classes in namespace
  - Must resolve 2 scopes: namespace & class
- Chained scope resolution operator
  - Provides access to nested scopes

2.3 Unary Scope Resolution Operator

- Allows access to objects in global scope
  - Necessary when local object hides object in global scope
- Best practice:
  - Always use unary scope resolution operator to access objects in global scope
  - Prevents accidentally hiding global objects in future

2.4 Inline Functions

- Allow compiler to optimize out function calls
  - Compiler may decide not to inline
- Ways to define function as inline
  - Explicit: Use inline keyword
  - Implicit: Place function body in class definition
- Inline function definition must go in .h file

2.5 Default Arguments

- Allow default values to be provided in function definitions
- Must be defined right to left
- May only be omitted in call from right to left
- Function call must be unambiguous

2.6 Constants

- Constants in C++ evaluated at compile time
- Can be used to size arrays, etc.
- Prefer constants to macros
- Best practice:
  - Always use constants instead of macros
  - Type safe

2.7 Constant Member Functions

- Objects can be constant
  - Only constant member functions can be called on a constant object
- Member functions non-const by default
- Best practice:
  - Mark all member functions that don't change state const
  - Allows const objects to be used by clients

2.8 const_cast

- Allows constant object to be cast to non-const object
- Caution:
  - Only use const_cast when known that object is not const
  - Changing state of const object results in undefined behavior

2.9 Mutable Data Members

- Ignored when compiler enforcing const rules
- Use only for data members that don't logically represent an object's state

2.10 Returning an Object From a Function

- Objects returned by value are temporary
- Returning non-const object
  - Changes to temporary are discarded
- Returning const object
  - Can’t make changes to temporary
- Choose appropriately based on how you expect function return value to be used

2.11 References

- Reference creates alias for variable
- Must be initialized when created
- Implemented as constant pointer by compiler

2.12 Reference Parameters

- Allow function to change value in calling environment

2.13 Reference vs. Pointer Parameters

- Reference parameters use same syntax as value parameters
  - Confusing at call point – will function change the argument?
- Best practice:
  - Use pointer parameters when change will occur
  - Use const reference parameters when no change will occur

2.14 Returning a Reference

- Object referred to must exist after function call
  - Use same guidelines for returning a reference as you do for returning pointer
    - (reference is just a constant pointer after all!)

2.15 Function Overloading

- Same function name can be used for multiple function definitions
  - Function signatures must be different
  - Compiler selects best match
- Function signature
  - Name of function
  - Parameter types
  - Important: Does not include return type

2.16 Function Overloading with Varying Return Types

- Return types of overloaded functions can be different, e.g.:
  - int foo(int);
  - double foo(double);
- Signature must still be different or compile error, e.g.:
  - int foo(int);
  - double foo(int); // Error!

2.17 Name Mangling

- Compiler mangles names to achieve function overloading
- Compiled names include parameter types
- Valuable to understand
  - Debuggers may show mangled names

2.18 Function Overloading Rules

- Selection of overloaded function must be unambiguous
- Rules, in order of precedence
  - Exact match
  - Match using promotions
  - Match using standard conversions
  - Match using user-defined conversions
  - Match using the ellipsis (…)

2.19 const is Part of Function Signature

- Compiler considers const vs. non-const parameters different when overloading functions:
  - void foo(int);
  - void foo(const int); // Different function
- Compiler consders const vs. non-const member functions differen when overloading functions

2.20 Operators new and delete

- new allocates object and returns pointer to that object
  - If out of memory - exception thrown
    - Default behavior is program termination
- delete deallocates the pointed-to object
- Prefer new and delete to C-memory management functions

2.21 Operators new[] and delete[]

- new[] allocates array of objects and returns pointer to beginning of array
- delete[] deallocates pointed-to array
- Be careful:
  - Always use new[] with delete[]
  - Always use new with delete

2.22 Data Members of User-Defined Types

- Data Members can be user-defined types
  - Work just like data members of built-in types
- Memory for data members laid out as part of parent object's memory

End of Lesson 2

- Supported by C (and C++)
- Focus is on actions
  - Procedures (functions), verbs
- Program viewed as series of function calls
- Data is secondary (parameters)
- Breaks down for large projects