Practical Aspects of C++ Programming

Chapter 4

Objectives

- **Call C code from a C++ program and vice versa**
- **Resolve name space conflicts in C++ programs.**
- **Explain the philosophy of C++ with regards to reliability.**
- **Summarize the features of C++ that promote reliable software.**
- **Specify an appropriate testing strategy for C++ code.**
- **Review C++ code for efficiency and implement performance improvements.**
- **Make use of class libraries to save on development effort.**
- **Gain experience through code walk-throughs and lab exercises.**
  - The example programs are in the chapter directory.

Interfacing C++ to Other Languages

- **In most languages a function's name (or the first n characters of it) is used by the linker to satisfy external references.**
- **This won't work in C++ because of function name overloading.**
- **The C++ compiler generates for each function a unique name based on the function name and its signature (argument list and types), a process known as name mangling.**
- **This causes a problem in linking a name-mangled C++ function and a function in another language whose name has not been mangled.**
- **Name mangling can be inhibited:**
  - extern "C" foo(int a, float x);

Demo Calling C from C++

- **Open and review the example program is in the Add folder.**

__cplusplus Macro

- **Defined when a program is being compiled under C++.**
- **Undefined under C compilation.**
- **This macro can be used to construct header files containing prototypes that can be used in C and that inhibit name mangling under C++.**

```
#ifdef __cplusplus
        extern "C" {
#endif
        int foo(int, int);
        int bar(float, char*);
#ifdef __cplusplus
        }
#endif
```

Demo Calling C++ from C

- **Usually cannot call directly, e.g. member function syntax *stack.push(x)* is not C!**
- **Create an interface module, which is a C++ shell that calls the C++ code and is itself callable from C.**
- **Use *extern "C"* to suppress name mangling.**
- **Our example program is in in the folder Stack.**

Namespace Collisions

- **C++ provides a single global namespace in which all names declared in global scope are entered.**
- **Single namespace is difficult for library providers and users.**
  - Global names in a library may collide with the global names in a user application or another library (e.g. there may be two **String** classes).
- **One workaround is for library vendors to adopt a unique prefix for names in their library.**
  - For example, classes in might begin with C, e.g. **CString.**
- **ANSI C++ standards committee has adopted a proposal for a standard mechanism for resolving namespace conflicts.**
- **Namespaces are now widely employed.**
  - e.g. C# and the Microsoft .NET Framework uses namespaces.

ANSI Namespace

- **A namespace (or user defined scope) is a mechanism for defining a scope.**
  - Namespaces are used to hold global C++ declarations, such as classes.
- **Names within a name space are accessed via the scope operator ::**
  - **lib_a::**Stack s;
- **A *using* declaration can make certain members of a namespace visible without requiring the names of these members to be qualified.**
  - using lib_a::Stack;
  - Stack s; // now Stack is in lib_a

Reliability Philosophies of Languages

- **Dynamic languages like Perl, PHP, Python, Ruby, etc.**
  - Emphasis on high productivity for small groups
  - Untyped language
  - Good for rapid prototyping and smaller projects
- **Languages like Java and C# emphasize reliability through a virtual machine or runtime that provides services such as garbage collection.**
- **C++**
  - In the middle
  - Strong typing
  - Use of **const**
  - Access control facilities

Prototypes and Type Checking

- **Function prototypes:**
  - Pioneered in C++
  - Part of ANSI C
  - Mandatory in C++
- **Strong type checking:**
  - Argument list and return type of every function call are type checked during compilation.
  - Number of arguments must agree.
  - Types of arguments and return value must agree either through an exact match or through an implicit type conversion.

Constant Types

- ***const* type modifier turns a symbolic variable into a *symbolic constant*.**
- **A symbolic constant is like a variable in having a memory location and a type, but is *read only*.**
- **A symbolic constant *must* be initialized when it is declared.**
- **You cannot assign the address of a symbolic constant to a pointer.**
  - Otherwise the value of the constant could get changed indirectly through the pointer.
- **Whenever you pass an argument by reference or through a pointer and you do not want the argument to be modified from within the called function, you should declare the argument as *const*.**
- **The compiler will check chains of function calls to ensure that a nested call will not break *const*.**

Access Control in C++

- **Avoid use of global variables.**
  - Prefer to pass data by arguments in functions calls.
  - Use file or class scope rather than **extern**.
  - Prefer enumerated types defined in a class to global constants.
- **Minimize use of global (free standing) functions.**
  - In an object oriented program C++ functions are normally member functions of a class.
  - Use static member functions if there is no dependency on instance data.
  - Free standing functions may be part of a C library.
- **Utilize private and protected access.**
  - Prefer data to be private within a class with access functions to read and write the data.
  - Use **friend** sparingly.
  - Use protected rather than public access when data or functions are needed by a derived class.

Reviews and Inspections

- **Not specific to C++.**
- **But when introducing a new programming technology such as OOP and C++ you have an opportunity at the same time to introduce or re-emphasize other important software engineering practices.**
- **Many studies have shown that systematic peer reviews are the most efficient means known to remove defects from software products.**
- **C++ specific checklists should be provided to assist in reviews.**
  - A good starting point for a checklist is the guidelines in the book *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* by Scott Myer.
- **An *inspection* is a particular kind of systematic review, first described by Fagan, and currently widely used in the industry.**

Inspections and C++

- **Preparation before Inspection Meeting:**
  - Promulgate organization- or project-wide programming standards.
  - Furnish each reviewer with a checklist of specific points to look for in the deliverable being reviewed.
  - The deliverable itself should be reviewed before the meeting.
- **Inspection Meeting:**
  - Formal meeting with defined roles for participants.
  - Moderator chairs the meeting. Recorder takes notes. Everyone is a reviewer.
- **Follow-up:**
  - After meeting make sure that rework needed to correct defects has been performed.

Testing Strategies for C++

- **Bottom-up testing is more important for C++. Every C++ class you define should be thoroughly tested as a standalone unit.**
  - For every class you develop also build an exerciser program that can call each member function with all parameter ranges.
  - Build scripts to automate running your exerciser programs.
- **As you incrementally add functions to a class do regression testing of previous functionality via your scripts.**
  - Consider inserting conditionally compiled code to increment a counter of objects in constructors and decrement the counter in destructors. Counter should be 0 on program termination.

Performance Considerations

- **There is great potential for inefficient C++ code due to such factors as:**
  - Invocation of hidden constructors in passing arguments, returning values, etc.
- **Ways to enhance performance include:**
  - Use reference arguments to cut down on copying objects and invoking constructors.
  - Use inline functions for small, frequently called functions.

Class Libraries

- **Implementing your own complete abstract data type is a big effort!**
- **Use existing libraries when available.**
  - Create and maintain your own libraries of classes specific to your application domain.
- **Sources of libraries:**
  - The ANSI standard C++ class library.
  - Class library that comes with your compiler
  - Public domain libraries such as National Institute of Health.
  - General purpose commercial class libraries (e.g. RogueWave).
  - Special purpose class libraries.

Summary

- **C code can be called from C++ programs by using the *extern "C"* directive to suppress name mangling.**
- **C++ code can be called from C programs by creating a C++ interface module that is callable from C, with *extern "C".***
- **C++ provides facilities for strong type checking. Using these facilities enables the compiler to catch many mistakes that otherwise might only show up at runtime.**
- **Emphasize bottom up testing in C++ code, so that you will have robust, generally usable classes.**
- **Review your C++ code for efficiency considerations, making sure you use references to avoid copying objects, use inline functions, etc.**
- **Use class libraries to cut down on development effort.**