cs121 - software development
c++ code design – ground rules

alexandre r.j. françois

visiting associate professor of computer science

outline

definitions

naming conventions

encapsulation

the global name space

documentation

doxygen

definitions

a *declaration* introduces a name into a program

a *definition* provides a unique description of an entity (e.g. type, instance, function) within a program

definitions

a name has *internal linkage* if it is local to its translation unit and cannot collide with an identical name defined in another translation unit at link time

a name has *external linkage* if, in a multi-file program, that name can interact with other translation units at link time

naming conventions

be consistent about identifier names

- use uppercase letters to delimit words in multi-word names
- (alternative is underscores)

```
someLongVariableName

some_long_variable_name
```

naming conventions

use a consistent method to distinguish type names from non-type names

- non-type (data and function) names begin with lowercase letter; type (entities that are neither data nor function) names start with an uppercase letter

```
AcustomTypeName

aVariableOfTypeACustomTypeName

someFunctionName()
```

naming conventions

use a consistent method to distinguish class names

first letter capitalized, prefixed with capital C

```
CSomeClass
```

naming conventions

use a consistent method to highlight class data members

- prefix with `m_`

```
m_someDataMember

someNonDataMember
```

naming conventions

use a consistent method to highlight data types in variable names

- n: integer, f or d: floating point, b: boolean, str: string, p for pointers, etc.

```
m_nSomeIntegerDataMember

fSomeFloatingPointVariable

pSomeString->size()

strSomeString.size()
```

naming conventions

use a consistent method to identify immutable values such as enumerators, constant data, and preprocessor constants

- all uppercase with underscores

```
const int MAX_INDEX=100;
```

naming conventions

be consistent about names used in the same way

- adopt consistent method names and operators for recurring design patterns such as iteration
- in C++, often i and j are integer indexes, p is a pointer; etc.

naming conventions

use a consistent method to highlight global variables

if you *must* have global variables, prefix with g_

```
bool g_bYouJustHadToDoIt
```

the global name space

partition the global name space

```
namespace fsf{
  class CRepository{
  ...
  };
} // namepsace fsf


fsf::CRepository aRepository;

using namespace fsf;
CRepository *pRepository = &aRepository;
```

global data

avoid data with external linkage at file scope

- no global variables

free functions

avoid free functions (except operator functions) at file scope in header files

avoid free functions with external linkage (including operator functions) in implementation files

enumerations, typedefs, and constant data

avoid enumerations, typedefs, and constants at file scope in header files

preprocessor macros

avoid using preprocessor macros in header files except as include guards

prefer `const` and `inline` to `#define`

```
const int MIN_VALUE=0;
const int MAX_VALUE=100;

inline int min(int a, int b){
  return (a<b)?a:b;
}
inline int max(int a, int b){
  return (a>b)?a:b;
}
```

include guards

place a unique and predictable (internal) include guard around the contents of each header file

```
#ifndef FSF_REPOSITORY_H
#define FSF_REPOSITORY_H
...
#endif
```

place a redundant (external) include guard around each preprocessor include directive in every header file

optional: important for very large projects

names in header files

only classes, structures, unions, and free operator functions should be declared at file scope in header files

only classes, structures, unions, and inline (member or free operator) functions should be defined at file scope in header files

layout

prefer space saving layouts

e.g. place opening bracket on same line as statement

prefer C++ style comments

```
// even for multi-line
// comments...
```

organize class members by categories of funcionality

e.g. creators, accessors, manipulators

place implementation details (private and protected) ahead of public interface

documentation

document the interfaces so they are usable by others

have at least one other developer review each interface

explicitly state conditions under which behavior is undefined

document assumptions

use of `assert` statements can help

documentation style guide

- when mentioned in descriptions, keywords and names should appear with the <code> style
  - keywords, namespaces, class names, method names, field names, argument names, code examples, etc.
- do not over-use in-line links
- do not use parens in the general form of methods and constructors
- use 3rd person (descriptive) rather than 2nd person (prescriptive)
- method descriptions begin with a verb phrase

documentation style guide

- class/interface/field descriptions can omit the subject and simply state the object
- use "this" instead of "the" when referring to an object created from the current class
- write descriptions that add information to the entity names

doxygen

Doxygen (www.doxygen.org) is a free, open source documentation system that supports the main documentation syntaxes in a variety of programming languages

Doxygen can automatically generate reference manuals in several formats (including HTML and LaTeX) from a set of documented source files

doxygen

Doxygen parses the code to extract specially formatted comments that document code entities. In the process, Doxygen also extracts the structure of the code and the relations between the various code elements. These relations can be visualized through automatically generated dependency graphs, inheritance diagrams and collaboration diagrams.

Doxygen can also be used to generate human-written documentation (beyond the automatically generated reference guide)