COMP6771
Advanced C++ Programming

Week 2
Part 1: Scopes, Namespaces, I/O, Strings, and Casting

2017

www.cse.unsw.edu.au/˜cs6771

This Week

- Scopes and Namespaces
- I/O
- std::string
- Casting
- Arrays
- Standard Template Library
- STL Containers
- Iterators

Scope

- Scope: the part of a program where a name is valid
- In the same scope, an entity (object, type, function or template) referred to by a name must be unique
- Scope rules used by compiler to determine if a reference to a identifier is legal or not at a particular point
- Five scopes (the first 3 are the same in C):
  1. local (blocks/compound statements delimited by { })
  2. function prototype (each parameter list is distinct)
  3. function (only applies to goto labels)
  4. namespace
  5. class scope

Scope

- A scope starts from its point of declaration
- Java programmers take note!

```
int i = j; // error: j not in scope
int j = i; // ok

void f() {
  g();     // error: g not in scope
}
void g() {
  f();     // ok
}
```

Local Scope

```
void f() {
  int i;
  for (int i = 1; i <= N; ++i)
  {...}
  for (int i = 1; i <= N; ++i)
  {...}
}
```

- The name i refers to three distinct entities
- A name is visible from declaration to its closing }
- A local name can be hidden

Statements in Local Scope

- Like in Java, this is ok (and recommended):

```
for (unsigned int i = 0; i < s.size(); ++i) {
    // i declared in the scope { } of for
}
```

- But, unlike in Java, this is also ok:

```
if (int i = compute_value())
  // do something
else
  j = i;  // ok: i in if in scope
k = i;     // error: out of scope
```

Prototype Scope

```
void f(int, int);
void f(int a, int b);
void f(int a, int a);
```

- The three declarations are equivalent
- a name is visible until its closest ',' or)
- the parameter names are only placeholders

Function Scope

```
for (int i = 0; i < 10; i++) {
  if (a[i] == key) goto found;
}
found:
...
```

- found is visible in the entire function body

Never use goto

Namespaces

- Namespace scope: part of program for packaging names
- To avoid name clashes with multiple, independently developed libraries:

```
namespace VendorOne {
  class Matrix {...}
  const double pi = 3.14;
  void inverse (matrix &);
  ...
}
```

```
namespace VendorTwo {
  class Matrix {...}
  const double pi = 3.14;
  void inverse (matrix &);
  ...
}
```

- Can be nested or aggregated
- Namespaces have the same feel as Java's packages
- The names in the C++ standard library are defined in std

Namespaces (Cont'd)

- Two special cases:
- Global scope: the outermost namespace scope of a file
- File scope: the namespace scope in a file (unnamed)

```
// file1.cpp:
namespace {
  int i = 3;
}
```

```
// file2.cpp:
namespace {
  int i = 10;
}
```

  - The scope automatically opened by compiler
  - Preferred over the use of static names in C:

```
// file1.c:
static int i = 3;
```

```
// file2.c:
static int i = 10;
```

Namespaces (Cont'd)

- using directive: makes the namespace members visible as if they were declared outside the namespace at the location where the namespace definition is located (e.g., `using namespace std;`)
- using declaration: Introduce a name into the scope where using declaration appears

```
std::cout << "hello" << std::endl;
using std::cout;
cout << "there" << endl;
```

- Scope resolution operator ::

Use "using directive" with care since it may pollute the global namespace

Namespace Clashes

```
namespace X {
  int i, j, k;
};

int k;

int main() {
  int i = 0;
  using namespace X; // make names from X accessible!
  i++;               // local i
  j++;               // X::j
  k++;               // error: X::k or global k?
  ::k++;             // the global k
  X::k++;            // X::k
}
```

Namespace Clashes

```
namespace X {
  int i, j, k;
};

int k;

int main() {
  int i = 0;
  X::i;          // error: i declared twice
  using X::j;
  using X::k;          // hides the global k
  j++;                 // X::j
  k++;                 // X::k
}
```

Namespace Style

- Don't put "using namespace std;" into header files.
- Generally avoid "using namespace std;" as it pollutes the scope.
- You may (with care) use specific namespace directives, e.g. "using std::cout;" — but even this is not recommended. See http://stackoverflow.com/a/1453605

C++ Input/Output Streams

- Class hierarchy:

```
                  ios_base
                     ios
                      /\
                     /  \
                    /    \
               istream   ostream
              /    |  \ /  |    \
             /     |   |   |     \
            /     /    |    \     \
           /     /     |     \     \
istringstream ifstream iostream ofstream ostringstream
                         /\
                        /  \
                   fstream  stringstream
```

- Will not cover this topic in detail

Predefined Stream Objects

- cin - the standard input (often the keyboard)
- cout - the standard output (often the terminal window)
- cerr - the standard error (often the terminal window).

```
int i,
float f;
std::string s;
std::complex c;

std::cin >> i >> f >> s >> c;
std::cout << i << f << s << c << std::endl;
std::cerr << i << f << s << c << std::endl;
```

<< and >> are pre-defined to work with built-in types and those in the library

Input and Output on Files

Input:

```
#include <iostream>
#include <fstream>

int i;
std::ifstream fin;
fin.open("data.in");
while (fin >> i) {
  std::cout << i << std::endl;
}
fin.close();
```

Output:

```
std::ofstream fout;
fout.open("data.out");
...
fout << i;
...
fout.close()
```

Formatting

```
#include <iostream>
#include <iomanip> // to use the setprecision manipulator

int main() {
  std::cout << 1331 << std::endl;
  std::cout << "In hex " << std::hex << 1331 << std::endl;
  std::cout << 1331.123456 << std::endl;
  std::cout.setf(std::ios::scientific, std::ios::floatfield);
  std::cout << 1331.123456 << std::endl;
  std::cout << std::setprecision(3) << 1331.123456 << std::endl;
  std::cout << std::dec << 1331 << std::endl;
  std::cout.fill('X');
  std::cout.width(8);
  std::cout << 1331 << std::endl;
  std::cout.setf(std::ios::left, std::ios::adjustfield);
  std::cout.width(8);
  std::cout << 1331 << std::endl;
  return 0;
}
```

Formatting (Cont'd)

The output is:

```
1331
In hex 533
1331.12
1.331123e+03
1.331e+03
1331
XXXX1331
1331XXXX
```

std::string

The C++ standard library provides a useful `string` type.

1. Prefer C++ `string` operations to C-style functions
2. Use iterators and `[]` for speed
3. Use `at()` when you need range checking
4. Use `find()` operations to locate values in a `string`
5. Append to `string` to add characters efficiently
6. be mindful of the character with the value 0; this is a valid character in C++, but it is treated as a terminator in C
7. Use `isalpha()`, `isdigit()`, etc. for character classification

Please refer to §3.2 of your textbook for the details.

Strings

Here is a simple example:

```
#include <iostream>
#include <string>

int main() {
  std::cout << "Please enter your first name: ";
  std::string name;
  std::cin >> name;

  const std::string greeting = "Hello, " + name + "!";
  std::cout << greeting << std::endl;

  return 0;
}
```

Strings

What if we wanted to say hello to multiple people?

```
#include <iostream>
#include <string>

int main() {
  std::string name, greeting;

  std::cout << "Please enter your first name: ";
  while(std::cin >> name) {
      greeting = "Hello, " + name + "!";
      std::cout << greeting << std::endl << std::endl;
      std::cout << "Please enter your first name: ";
  }

  return 0;
}
```

What happens if we input a blank name? What about multiple names? When does the while loop stop?

The `string` Interface

The standard library `string` is a surprisingly complex entity and it provides a rich interface.

- element access: iterators, `[]` and `at()`
- constructors: empty, string, C-style string, substring
- error handling (exceptions): `out_of_range()`
- assignment (copy): `=`
- concatenation and appending: `+`, `+=`, `insert()`, `append()`

Uninitialised `strings`

The default initialiser for `string` is the empty string.

Classes

In C++ we define a new type by defining a *class*.

- Pre-defined classes/types in the library: vector, string, . . .
- User-defined classes, e.g.

```
class Book {
  public:
    std::string isbn();
    ...
  private:
    std::string isbn;
    unsigned int units_sold;
    double revenue;
};
```

Objects

- An object is a region of memory with a type
- Two kinds of objects
  - an class object (i.e., an instance of a class)
  - an non-class object

All variables are identified with named objects:

```
int i = 1;          // an int object
std::string s = "abc";  // a string object
std::vector<int> v(5); // a vector of 5 int's
          // i.e., a vector object holding 5 int objects
```

- C++ is case-sensitive
- A name must be declared before it is used
- Read §2.2 about default initialisation.

What is a Variable Definition?

A variable is a construct that:

- provides named storage
- has a specific type
- is an *lvalue*
- must be defined (or declared) before it is used
- should NOT be used as an *rvalue* before it is initialised

§2.2.3 covers naming guidelines and conventions.
§2.2.1 covers initialisation issues.

Initialization

```
int main() {
  int i1 {1};    // {}-list preferred (C++11)
  int i2 = {1};
  int i3 = 1;
  int i4(1);

  int j1{};           // the default value used
  int j2 = int{};    // a temporary containing the default value
}
```

Will look at initialser lists later

Read §2.2 about default initialisation.

Strong Static Type Checking

Strong Static Typing

Type-checking happens during compilation. Compiler will produce type-mismatch errors if types are used illegally. Types must be known at compile time!

Example:

```
float f;
float *pf = &f;   // ok
int *pi = &f;     // error int pointer can't point to a float
```

Type Conversions

- Implicit, i.e., compiler-directed conversions
- Explicit, i.e., programmer-specified conversions

Promotion: value-preserving conversions

```
int b = true;   // a bool converted to an int
long l = 1;     // an int converted to a long
```

Caution

Type convert with care. Potential for data loss without any warning.

```
int pi = 3.14;  // pi has value 3. No compilier warning.
```

Static Casting

Three ways of static casting:

```
double myDouble = 3.14;
int cast1 = (int)myDouble;  // c-style
int cast2 = int(myDouble);
int cast3 = static_cast<int>(myDouble);  // recommended
```

Example, int division:

```
int i = 5, j = 2;
int slopeInt = i / 5;
double slopeDouble = static_cast<double>(i) / j;
```

reinterpret_cast

Low level reinterpretation of the bit pattern.

Example, convert and modify a pointer address to a long.

```
#include <iostream>

// Returns a hash code based on an address
unsigned short hash(void *p) {
  unsigned long val = reinterpret_cast<unsigned long>(p);
  return ( unsigned short )(val ^ (val >> 16));
}

int main() {
  int a[20];
  for (int i = 0; i < 20; i++)
    std::cout << hash(a + i) << std::endl;
}
```

const_cast

- Removes low level const, i.e., 'casts away the const'.
- Avoid doing this if you can.
- Used mostly for portability:

```
3rd Party code:

// You know that this function doesn't modify fooPtr
printFoo(Foo* fooPtr)  ...

Your code:

// Your code has made an effort to use Foo* in
// contexts where writability is needed and
// const Foo* in all other contexts.

const Foo* f = &constFoo;
printFoo(<const_cast<Foo *>(f);
```