Scopes
○○○○○○

Namespaces
○○○○○○

I/O
○○○○○

Strings
○○○○

Type Casting
○○○○○○○○○

COMP6771
Advanced C++ Programming

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

2016

www.cse.unsw.edu.au/~cs6771

Scopes
000000

Namespaces
000000

I/O
00000

Strings
0000

Type Casting
000000000

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!

```
1 int i = j; // error: j not in scope
2 int j = i; // ok
3
4 void f() {
5   g();    // error: g not in scope
6 }
7 void g() {
8   f();    // ok
9 }
```

Local Scope

```
1  void f() {
2    int i;
3    for (int i = 1; i <= N; ++i)
4    {...}
5    for (int i = 1; i <= N; ++i)
6    {...}
7  }
```

- 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):

```
1  for (unsigned int i = 0; i < s.size(); ++i) {
2      // i declared in the scope { } of for
3  }
```

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

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

Scopes
○○○○●○

Namespaces
○○○○○○

I/O
○○○○○

Strings
○○○○

Type Casting
○○○○○○○○○

Prototype Scope

```
1  void f(int, int);
2  void f(int a, int b);
3  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

```
1  for (int i = 0; i < 10; i++) {
2      if (a[i] == key) goto found;
3  }
4  found:
5  ...
```

- **found** is visible in the entire function body

Never use goto

Scopes
oooooo

Namespaces
●ooooo

I/O
ooooo

Strings
oooo

Type Casting
ooooooooo

Namespaces

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

```
1  namespace VendorOne {
2    class Matrix {...}
3    const double pi = 3.14;
4    void inverse (matrix &);
5    ...
6  }
```

```
1  namespace VendorTwo {
2    class Matrix {...}
3    const double pi = 3.14;
4    void inverse (matrix &);
5    ...
6  }
```

- 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

Scopes
○○○○○○

Namespaces
○●○○○○

I/O
○○○○○

Strings
○○○○

Type Casting
○○○○○○○○○

Namespaces (Cont'd)

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

```
1  // file1.cpp:
2  namespace {
3    int i = 3;
4  }
```

```
1  // file2.cpp:
2  namespace {
3    int i = 10;
4  }
```

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

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

```
1  // file2.c:
2  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

Scopes
○○○○○○

Namespaces
○○○●○○

I/O
○○○○○

Strings
○○○○

Type Casting
○○○○○○○○○

Namespace Clashes

```
 1  namespace X {
 2    int i, j, k;
 3  };
 4
 5  int k;
 6
 7  int main() {
 8    int i = 0;
 9    using namespace X; // make names from X accessible!
10    i++;                // local i
11    j++;                // X::j
12    k++;                // error: X::k or global k?
13    ::k++;              // the global k
14    X::k++;             // X::k
15  }
```
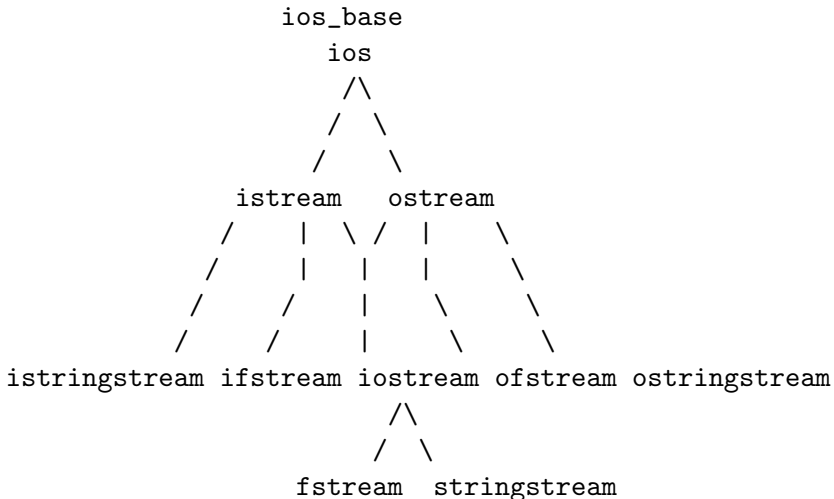
Namespace Clashes

```
1  namespace X {
2    int i, j, k;
3  };
4
5  int k;
6
7  int main() {
8    int i = 0;
9    X::i;           // error: i declared twice
10   using X::j;
11   using X::k;        // hides the global k
12   j++;              // X::j
13   k++;              // X::k
14 }
```

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).

```
1  int i,
2  float f;
3  std::string s;
4  std::complex c;
5
6  std::cin >> i >> f >> s >> c;
7  std::cout << i << f << s << c << std::endl;
8  std::cerr << i << f << s << c << std::endl;
9
```

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

Input and Output on Files

Input:

```
1  #include <iostream>
2  #include <fstream>
3
4  int i;
5  std::ifstream fin;
6  fin.open("data.in");
7  while (fin >> i) {
8    std::cout << i << std::endl;
9  }
10 fin.close();
```

Output:

```
1  std::ofstream fout;
2  fout.open("data.out");
3  ...
4  fout << i;
5  ...
6  fout.close()
```

Scopes
000000

Namespaces
000000

I/O
000●0

Strings
0000

Type Casting
000000000

Formatting

```cpp
1  #include <iostream>
2  #include <iomanip> // to use the setprecision manipulator
3
4  int main() {
5    std::cout << 1331 << std::endl;
6    std::cout << "In hex " << std::hex << 1331 << std::endl;
7    std::cout << 1331.123456 << std::endl;
8    std::cout.setf(std::ios::scientific, std::ios::floatfield);
9    std::cout << 1331.123456 << std::endl;
10   std::cout << std::setprecision(3) << 1331.123456 << std::endl;
11   std::cout << std::dec << 1331 << std::endl;
12   std::cout.fill('X');
13   std::cout.width(8);
14   std::cout << 1331 << std::endl;
15   std::cout.setf(std::ios::left, std::ios::adjustfield);
16   std::cout.width(8);
17   std::cout << 1331 << std::endl;
18   return 0;
19 }
```

18

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:

```cpp
#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;
}
```

Scopes
000000

Namespaces
000000

I/O
00000

Strings
00●0

Type Casting
000000000

Strings

What if we wanted to say hello to multiple people?

```cpp
#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

```
1  int main() {
2    int i1 {1};      // {}-list preferred (C++11)
3    int i2 = {1};
4    int i3 = 1;
5    int i4(1);
6
7    int j1{};         // the default value used
8    int j2 = int{};   // a temporary containing the default value
9  }
```

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:

```
1  float f;
2  float *pf = &f;  // ok
3  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

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

Caution

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

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

Static Casting

Three ways of static casting:

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

Example, int division:

```
1 int i = 5, j = 2;
2 int slopeInt = i / 5;
3 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.

```
1  #include <iostream>
2
3  // Returns a hash code based on an address
4  unsigned short hash(void *p) {
5    unsigned long val = reinterpret_cast<unsigned long>(p);
6    return ( unsigned short )(val ^ (val >> 16));
7  }
8
9  int main() {
10   int a[20];
11   for (int i = 0; i < 20; i++)
12     std::cout << hash(a + i) << std::endl;
13 }
```

const_cast

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

```
 1  3rd Party code:
 2
 3  // You know that this function doesn't modify fooPtr
 4  printFoo(Foo* fooPtr)   ...
 5
 6  Your code:
 7
 8  // Your code has made an effort to use Foo* in
 9  // contexts where writability is needed and
10  // const Foo* in all other contexts.
11
12  const Foo* f = &constFoo;
13  printFoo(<const_cast<Foo *>(f);
```