Cyclic association – a simple example

```
#include <iostream>
#include <string>
using namespace std;

class Princess {
    friend ostream& operator<<(ostream& os, const Princess& princess)
{
        os << "Princess: " << princess.name;
        return os;
    }
public:
    Princess(const string& name) : name(name) {}
private:
    string name;
};

class Prince {
    friend ostream& operator<<(ostream& os, const Prince& rhs) {
        os << "Prince: " << rhs.name;
        return os;
    }
public:
    Prince(const string& name) : name(name) {}
private:
    string name;
};
```

Cyclic association – a simple example, cont.

```
int main() {
    Princess snowy("Snow White");
    cout << snowy << endl;
    Prince charmy("Charmy");
    cout << charmy << endl;

    // snowy.marries(charmy);
    cout << snowy << endl
         << charmy << endl;
}
```

```
Princess: Snow White
Prince: Charmy
Princess: Snow White
Prince: Charmy
```

But now I need to implement the prince“marries” princess method

Cyclic association

```cpp
#include <iostream>
#include <string>
using namespace std;


class Princess {
    friend ostream& operator<<(ostream& os, const Princess& princess);
public:
    Princess(const string& name) : name(name), spouse(nullptr) {}
    void marries(Prince& aPrince) {
        spouse = &aPrince;
        aPrince.setSpouse(this);
    }
private:
    string name;
    Prince* spouse;
};

class Prince {
    friend ostream& operator<<(ostream& os, const Prince& prince);
public:
    Prince(const string& name) : name(name), spouse(nullptr) {}
    const string& getName() const { return name; }
    void setSpouse(Princess* spouse) {
        this->spouse = spouse;
    }
private:
    string name;
    Princess* spouse;
};
```

```
Build started...
1>------ Build started: Project: Lect_01_B, Configuration: Release x64 ------
1>testPrincess_om.cpp
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(10,18): error C2061: syntax error: identifier 'Prince'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(16,15): error C2143: syntax error: missing ';' before '*'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(16,15): error C4430: missing type specifier - int assumed. Note:
C++ does not support default-int
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(16,23): error C2238: unexpected token(s) preceding ';'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(9,48): error C2614: 'Princess': illegal member initialization:
'spouse' is not a base or member
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(11,9): error C2065: 'spouse': undeclared identifier
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(11,19): error C2065: 'aPrince': undeclared identifier
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(12,9): error C2065: 'aPrince': undeclared identifier
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(36,21): error C2039: 'spouse': is not a member of 'Princess'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(6): message : see declaration of 'Princess'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(36,57): error C2039: 'spouse': is not a member of 'Princess'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(6): message : see declaration of 'Princess'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(48,25): error C2660: 'Princess::marries': function does not take
1 arguments
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic
Association\testPrincess_om.cpp(10,10): message : see declaration of 'Princess::marries'
1>Done building project "cs2124.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Cyclic association

```
ostream& operator<<(ostream& os, const Princess& princess) {
    os << "Princess: " << princess.name;
    if (princess.spouse) os << "; Married to " << princess.spouse-
>getName();
    else os << "; Single";
    return os;
}
ostream& operator<<(ostream& os, const Prince& prince) {
    os << "Prince: " << prince.name;
    return os;
}
int main() {
    Princess snowy("Snow White");
    cout << snowy << endl;
    Prince charmy("Charmy");
    cout << charmy << endl;
    snowy.marries(charmy);
    cout << snowy << endl
         << charmy << endl;
}
```

Cyclic association

```cpp
#include <iostream>
#include <string>
using namespace std;

class Prince;

class Princess {
    friend ostream& operator<<(ostream& os, const Princess& princess);
public:
    Princess(const string& name) : name(name), spouse(nullptr) {}
    void marries(Prince& aPrince) {
        spouse = &aPrince;
        aPrince.setSpouse(this);
    }
private:
    string name;
    Prince* spouse;
};

class Prince {
    friend ostream& operator<<(ostream& os, const Prince& aPrince);
public:
    Prince(const string& name) : name(name), spouse(nullptr) {}
    const string& getName() const { return name; }
    void setSpouse(Princess* spouse) {
        this->spouse = spouse;
    }
private:
    string name;
    Princess* spouse;
};
```

```
Build started...
1>------ Build started: Project: Lect_01_B, Configuration: Release x64 ------
1>testPrincess_om.cpp
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic Association\testPrincess_om.cpp(13,1): error C2027: use of undefined type 'Prince'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic Association\testPrincess_om.cpp(5): message : see declaration of 'Prince'
1>Done building project "cs2124.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

- Note that when the function definition is inside the class → we are instructing the compiler to do inlining.
- **An inlined function** is expanded instead of making an actual function call – see next slide

Cyclic association

```
ostream& operator<<(ostream& os, const Princess& princess) {
    os << "Princess: " << princess.name;
    if (princess.spouse) os << "; Married to " << princess.spouse->getName();
    else os << "; Single";
    return os;
}
ostream& operator<<(ostream& os, const Prince& prince) {
    os << "Prince: " << prince.name;
    return os;
}
int main() {
    Princess snowy("Snow White");
    cout << snowy << endl;
    Prince charmy("Charmy");
    cout << charmy << endl;
    snowy.marries(charmy);
    cout << snowy << endl
         << charmy << endl;
}
```

```
Build started...
1>------ Build started: Project: Lect_01_B, Configuration: Release x64 ------
1>testPrincess_om.cpp
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic Association\testPrincess_om.cpp(13,1): error C2027: use of undefined type 'Prince'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\05a.Cyclic Association\testPrincess_om.cpp(5): message : see declaration of 'Prince'
1>Done building project "cs2124.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

- In this line the marries method is expanded and not called due to inlining
  - More time efficient - saves time overhaead for making calls (pushing parameters into stack, etc.)
  - Less space efficient - if a method is used 10 times, the code is replicated 10 times!

Cyclic association - solution

```
#include <iostream>
#include <string>
using namespace std;

class Prince;

class Princess {
    friend ostream& operator<<(ostream& os, const Princess& princess);
public:
    Princess(const string& name);
    void marries(Prince& aPrince);
private:
    string name;
    Prince* spouse;
};

class Prince {
    friend ostream& operator<<(ostream& os, const Prince& prince);
public:
    Prince(const string& name);
    const string& getName() const;
    void setSpouse(Princess* spouse);
private:
    string name;
    Princess* spouse;
};

int main() {
    Princess snowy("Snow White");
    cout << snowy << endl;
    Prince charmy("Charmy");
    cout << charmy << endl;
    snowy.marries(charmy);
    cout << snowy << endl
        << charmy << endl;
}
```

- **Solution:**
  - Prototype the methods within the class.
  - Define the methods later

Cyclic association

```
//
// Princess function definitions
//
Princess::Princess(const string& name) : name(name), spouse(nullptr) {}

void Princess::marries(Prince& aPrince) {
    spouse = &aPrince;
    aPrince.setSpouse(this);
}
ostream& operator<<(ostream& os, const Princess& princess) {
    os << "Princess: " << princess.name;
    if (princess.spouse) os << "; Married to " << princess.spouse->getName();
    else os << "; Single";
    return os;
}

//
// Prince function definitions
//
Prince::Prince(const string& name) : name(name), spouse(nullptr) {}

const string& Prince::getName() const { return name; }

ostream& operator<<(ostream& os, const Prince& prince) {
    os << "Prince: " << prince.name;
    return os;
}

void Prince::setSpouse(Princess* spouse) {
    this->spouse = spouse;
}
```

Princess: Snow White; Single
Prince: Charmy
Princess: Snow White; Married to Charmy
Prince: Charmy

- Note the importance of class tags (and the scope operator)
- operator<< is not a part of the class and thus requires no class tag (it's just a friend function)

Cyclic association

```
//
// Princess function definitions
//
Princess::Princess(const string& name) : name(name), spouse(nullptr) {}

void Princess::marries(Prince& aPrince) {
    spouse = &aPrince;
    aPrince.setSpouse(this);
}
ostream& operator<<(ostream& os, const Princess& princess) {
    os << "Princess: " << princess.name;
    os << (princess.spouse ? "; Married to " + princess.spouse->getName() :
            "Single");
    return os;
}

//
// Prince function definitions
//
Prince::Prince(const string& name) : name(name), spouse(nullptr) {}

const string& Prince::getName() const { return name; }

ostream& operator<<(ostream& os, const Prince& prince) {
    os << "Prince: " << prince.name;
    return os;
}

void Prince::setSpouse(Princess* spouse) {
    this->spouse = spouse;
}
```

- The ternary operator may come in handy here. It takes the form:

(condition)? expression_1 : expression_2;

Namespaces

- Two entities may not have the same variable name within the same scope, otherwise a name collision occurs.
  - The same applies for types (e.g. classes and structs)

- Name collisions rarely exist ***for local variables***, since blocks tend to be relatively short and written by one programmer.

Namespaces

```
#include <iostream>
#include <string>
using namespace std;

int main() {

    {
        string temp = "outer block";
        {
            string temp = "inner block";
            cout << temp << endl;
        }
        cout << temp << endl;
    }
}
```

```
inner block
outer block
```

- Note that if we have a block of code that is enclosed in another code block, then:
  - The same variable name may be used twice, once for the inner block and another for the outer block.
  - This is not considered a name collision since they don't have the same scope.

Namespaces (cont.)

- ***Global variables*** are much more likely to have a name collisions
  - Different libraries may be written by different programmers
    - It is common for the names chosen to be descriptive of the function/variable's functionality.Different programmers may choose the same variable name, even though they are designing different libraries, e.g.:
      - `get(), set(),`
      - `i, j, k, count`, etc.).

- Namespaces allow us to **group named entities** that otherwise would have *global scope* **into narrower scopes**, giving them *namespace scope*.
  - This allows organizing the elements of programs into different logical scopes referred to by names.

Namespaces (cont.)

- Namespaces may only be defined **at file scope** or **within another namespace**, i.e. they **may not be defined within a function body for local variables**.

- A namespace is defined using the following syntax:

```
namespace identifier1{
        type identifier2;
}
```

Namespaces (example)

```
// namespaces
#include <iostream>
using namespace std;

namespace ns1{
  int get_val() { return 5; }
}

namespace ns2{
  const double pi = 3.1416;
  double get_val() { return 2*pi; }
}

int main () {
  cout << ns1::get_val() << '\n';
  cout << ns2::get_val() << '\n';
  cout << ns2::pi << '\n';
  return 0;
}
```

Namespaces (example)

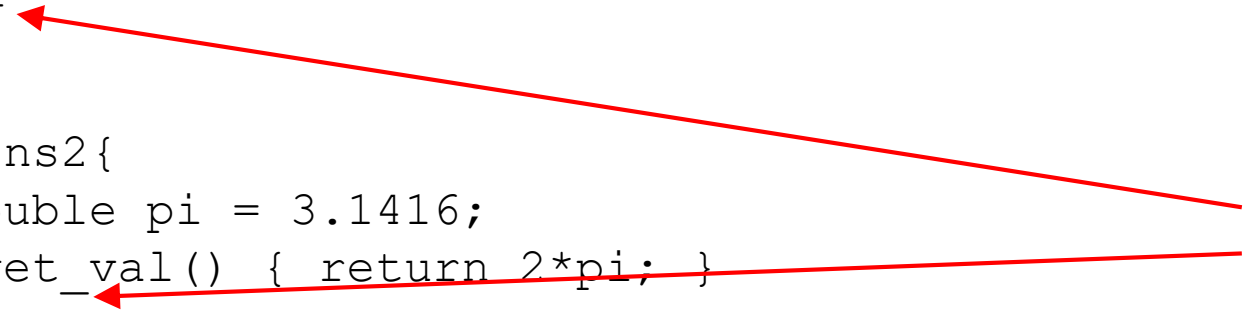
```
// namespaces
#include <iostream>
using namespace std;

namespace ns1{
  int get_val() { return 5; }
}

namespace ns2{
  const double pi = 3.1416;
  double get_val() { return 2*pi; }
}

int main () {
  cout << ns1::get_val() << '\n';
  cout << ns2::get_val() << '\n';
  cout << ns2::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```



Same name but no collision (thanks to the namespace declaration)

Namespaces (example)

```
// namespaces
#include <iostream>
using namespace std;

namespace ns1{
  int get_val() { return 5; }
}

namespace ns2{
  const double pi = 3.1416;
  double get_val() { return 2*pi; }
}

int main () {
  cout << ns1::value() << '\n';
  cout << ns2::value() << '\n';
  cout << ns2::pi << '\n';
  return 0;
}
```

No need to qualify the name with the scope operator "::" when accessed within the namespace

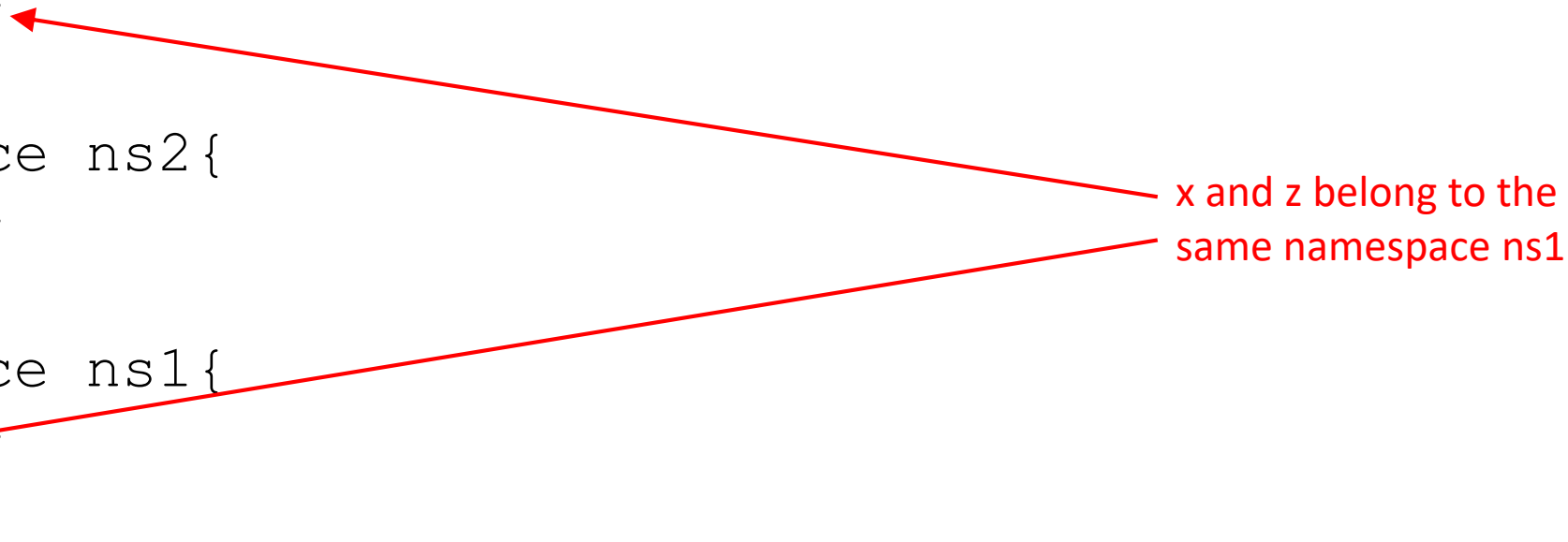
```
5
6.2832
3.1416
```

Need to qualify the name with the scope operator "::" when accessed outside the name space declaration

Namespaces - splitting

- Namespaces can be split:

```
namespace ns1{
   int x;
}
namespace ns2{
   int y;
}
namespace ns1{
   int z;
}
```



Namespaces – the `using` keyword

```
#include <iostream>
using namespace std;
namespace first{
  int x = 5;
  int y = 10;
}
namespace second {
  double x = 3.1416;
  double y = 2.7183;
}
int main (){
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

Don’t need to use the scope operator “::” when we use the “using” keyword

Need to use the scope operator “::” to avoid name collisions

Namespaces – the `using` keyword

```
#include <iostream>
using namespace std;
namespace first{
  int x = 5;
  int y = 10;
}
namespace second {
  double x = 3.1416;
  double y = 2.7183;
}
int main (){
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

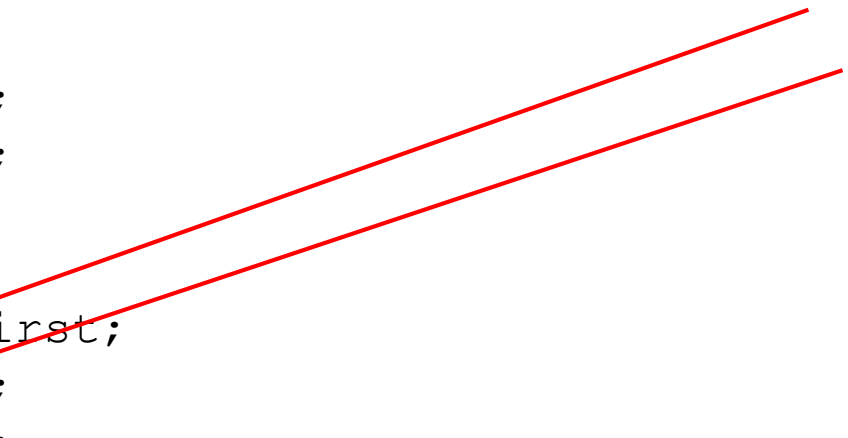
```
5
2.7183
10
3.1416
```

Don’t need to use the scope operator “::” when we use the “using” keyword

Need to use the scope operator “::” to avoid name collisions

Namespaces – the `using` keyword

```
#include <iostream>
using namespace std;
namespace first{
  int x = 5;
  int y = 10;
}
namespace second{
  double x = 3.1416;
  double y = 2.7183;
}
int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

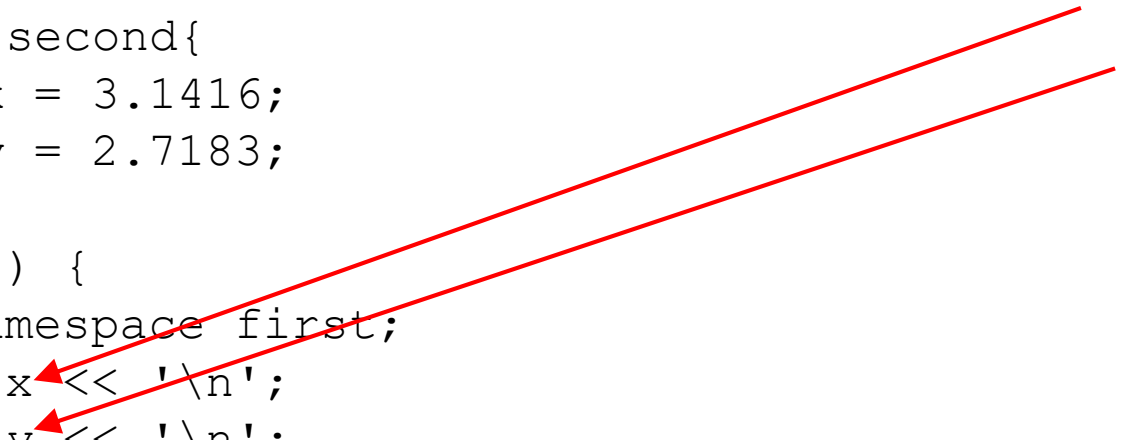


Don’t need to use the scope operator “::” when we use the “using namespace” keyword to include the entire namespace

Namespaces – the `using` keyword

```
#include <iostream>
using namespace std;
namespace first{
  int x = 5;
  int y = 10;
}
namespace second{
  double x = 3.1416;
  double y = 2.7183;
}
int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

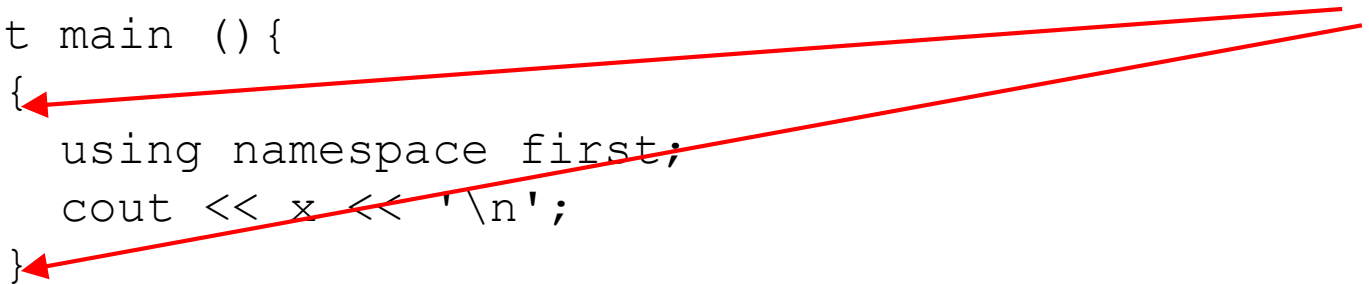
```
5
10
3.1416
2.7183
```



Don’t need to use the scope operator “::” when we use the “using namespace” keyword to include the entire namespace

Namespaces – the `using` keyword

```
// using namespace example
#include <iostream>
using namespace std;
namespace first{
  int x = 5;
}
namespace second{
  double x = 3.1416;
}
int main (){
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```



- Declaring that we are using a namespace does not invalidate the previous usage declarations
- But enclosing “using” and “using namespace” keywords within a block invalidates them (i.e. makes them invisible) as soon as we exit that block.

Namespaces – the `using` keyword

```
// using namespace example
#include <iostream>
using namespace std;
namespace first{
  int x = 5;
}
namespace second{
  double x = 3.1416;
}
int main (){
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

```
5
3.1416
```

- Declaring a that we are using a namespace does not invalidate the previous usage declarations
- But enclosing “using” and “using namespace” keywords within a block invalidates them (i.e. makes them invisible) as soon as we exit that block.

Namespaces – cont.

- Existing namespaces can be ***aliased*** with new names, with the following syntax:

```
namespace new_name = current_name;
```

- All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the **`std`** namespace. Most of the examples we used so far, in fact, include the following line:

```
using namespace std;
```

- It is also common to see programmers use:

```
std::cout << "Hello world!";
```

instead of:

```
cout<< "Hello world!";
```

Namespace

```
namespace Fantasy {

    class Prince;

    class Princess {
            friend ostream& operator<<(ostream& os, const Princess&
            princess);
    public:
            Princess(const string& name);
            void marries(Prince& aPrince);
    private:
            string name;
            Prince* spouse;
    };

    class Prince {
            friend ostream& operator<<(ostream& os, const Prince&
            aPrince);
    public:
            Prince(const string& name);
            const string& getName() const;
            void setSpouse(Princess* spouse);
    private:
            string name;
            Princess* spouse;
    };

}

int main() {
    Fantasy::Princess snowy("Snow White");
    cout << snowy << endl;
    Fantasy::Prince charmy("Charmy");
    cout << charmy << endl;
    snowy.marries(charmy);
    cout << snowy << endl
         << charmy << endl;
}
```

Namespace – cont.

```
namespace Fantasy {
    //
    // Princess function definitions
    //
    Princess::Princess(const string& name) : name(name),
        spouse(nullptr) {}

    void Princess::marries(Prince& aPrince) {
           spouse = &aPrince;
           aPrince.setSpouse(this);
    }
    ostream& operator<<(ostream& os, const Princess& princess) {
        os << "Princess: " << princess.name;
        if (princess.spouse) os << "; Married to " <<
            princess.spouse->getName();
        else os << "; Single";
        return os;
    }

    //
    // Prince function definitions
    //
    Prince::Prince(const string& name) : name(name), spouse(nullptr)
{}

    const string& Prince::getName() const { return name; }

    ostream& operator<<(ostream& os, const Prince& aPrince) {
           os << "Prince: " << aPrince.name;
           return os;
    }

    void Prince::setSpouse(Princess* spouse) {
           this->spouse = spouse;
    }
}
```

Princess: Snow White; Single
Prince: Charmy
Princess: Snow White; Married to Charmy
Prince: Charmy

- Can we use the ternary operator instead?

Namespace – cont.

```
namespace Fantasy {
    //
    // Princess function definitions
    //
    Princess::Princess(const string& name) : name(name),
        spouse(nullptr) {}

    void Princess::marries(Prince& aPrince) {
           spouse = &aPrince;
           aPrince.setSpouse(this);
    }
    ostream& operator<<(ostream& os, const Princess& princess) {
        os << "Princess: " << princess.name;
        os << (princess.spouse ? "; Married to " +
            princess.spouse->getName() : "Single");

        return os;
    }
}
    //
    // Prince function definitions
    //
    Fantasy::Prince::Prince(const string& name) : name(name),
spouse(nullptr) {}

    const string& Fantasy::Prince::getName() const { return name; }

    ostream& Fantasy::operator<<(ostream& os, const Prince& aPrince)
{
           os << "Prince: " << aPrince.name;
           return os;
    }

    void Fantasy::Prince::setSpouse(Princess* spouse) {
           this->spouse = spouse;
    }
//}
```

Princess: Snow White; Single
Prince: Charmy
Princess: Snow White; Married to Charmy
Prince: Charmy

- The member-function definition may or may not be inside the namespace.
- If outside the namespace, it needs to be qualified with "Fantasy::"