Namespace, Pointers and references, Arrays

Shahram Rahatlou

SAPIENZA
UNIVERSITÀ DI ROMA

Corso di Programmazione++

http://www.roma1.infn.it/people/rahatlou/programmazione++/

Roma, 17 March 2008

Today's Topics

- Scope of variables
- Namespace: what they are and how to use it
- Arrays
- Pointers and references
- Functions
  - Use of constants in function interface

Problems with `cin`

```
// tinput_bad.cc
#include <iostream>
using namespace std;

int main() {

  cout << "iterations? ";

  int iters;
  cin >> iters;

  cout << "requested " << iters << " iterations" << endl;

  return 0;
}
```

```
$ g++ -Wall -o tinput_bad tinput_bad.cc
$ ./tinput_bad
iterations? 23
requested 23 iterations
$ ./tinput_bad
iterations? dfed
requested 134514793 iterations
```

First mistake: Always initialize your variables!

```
// tinput_bad2.cc
#include <iostream>
using namespace std;

int main() {

  int iters;
  cout << "iters before cin: " << iters << endl;

  cout << "iterations? ";
  cin >> iters;

  cout << "requested " << iters << " iterations" << endl;

  return 0;
}
```

Random value since not initialized!

```
$ g++ -Wall -o tinput_bad2 tinput_bad2.cc
$ ./tinput_bad2
iters before cin: 134514841
iterations? 3
requested 3 iterations
$ ./tinput_bad2
iters before cin: 134514841
iterations? er
requested 134514841 iterations
```

Checking `cin` success or failure

```
//tinput.cc
#include <iostream>
using namespace std;

int main() {

  cout << "iterations? ";

  int iters = 0;
  cin >> iters;

  if(cin.fail()) cout << "cin failed!" << endl;

  cout << "requested " << iters << " iterations" << endl;

  return 0;
}
```

Fails if input data doesn't match expected data type

```
$ g++ -Wall -o tinput tinput.cc
$ ./tinput
iterations? 34
requested 34 iterations
$ ./tinput
iterations? sfee
cin failed!
requested 0 iterations
```

Control Statements in C++

```
// SimpleIf.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

   if( 1 == 0 ) cout << "1==0" << endl;

   if( 7.2 >= 6.9 ) cout << "7.2 >= 6.9" << endl;

   bool truth = (1 != 0);
   if(truth) cout << "1 != 0" << endl;

   if( ! ( 1.1 >= 1.2 ) ) cout << "1.1 < 1.2" << endl;

   return 0;
} // end of main
```

```
$ g++ -o SimpleIf SimpleIf.cpp
$ ./SimpleIf
7.2 >= 6.9
1 != 0
1.1 < 1.2
```

Scope of Variables

- The scope of a name is the block of program where the name is valid and can be used
  - A block is delimited by { }
  - It can be the body of a method, or a simple scope defined by the user using { }

```
// scope.cc
#include <iostream>


double f1() {
  double y = 2;
  return y;
}

int main() {

   double x = 3;
   double z = f1();

   std::cout << "x: " << x << ", z: " << z << ", y: " << y
             << std::endl;
   return 0;
}
```

```
$ g++ -o scope scope.cc
scope.cc: In function `int main()':
scope.cc:16: error: `y' undeclared (first use this function)
scope.cc:16: error: (Each undeclared identifier is reported
 only once for each function it appears in.)
```

What is the difference between **`cout`** and **`std::cout`**?

What is **namespace** ?

- A mechanism to group declarations that logically belong to each other

```
namespace physics {
  class vector;
  class unit;
  class oscillator;
  void sort(const vector& value);
}

namespace electronics {
  void sort(const vector& value);
  class oscillator;
}

namespace graphics {
  void sort(const vector& value);
  class unit;
}
```

- Provides an easy way for logical separation of parts of a big project

- Basically a ‘scope’ for a group of related declarations

How do I use **namespaces** ?

```
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
  double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

int main() {
   double x = 3;
   double y = 4;

   double z1 = physics::mean(x,y);
   std::cout << "physics::mean(" << x << "," << y << ") = " << z1
             << std::endl;

   double z2 = foobar::mean(x,y);
   std::cout << "foobar::mean(" << x << "," << y << ") = " << z2
             << std::endl;
   return 0;
}
```

physics::mean

foobar::mean

Use "::" to specify the namespace

Defined in iostream

```
$ g++ -o namespace1 namespace1.cc
$ ./namespace1
physics::mean(3,4) = 3.5
foobar::mean(3,4) = 12.5
```

Common Errors with **namespaces**

```
// namespaceBad.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

int main() {

   double x = 3;
   double y = 4;

   double z1 = mean(x,y); // forgot the namespace!
   cout << "physics::mean(" << x << "," << y << ") = " << z1
             << std::endl;

   return 0;
}
```

If you forget to specify the namespace the compiler doesn't know where to find the method

```
$ g++ -o namespaceBad namespaceBad.cc
namespaceBad.cc: In function `int main()':
namespaceBad.cc:15: error: `mean' undeclared (first use this function)
namespaceBad.cc:15: error: (Each undeclared identifier is reported only
 once for each function it appears in.)
namespaceBad.cc:16: error: `cout' undeclared (first use this function)
```

`using namespace` directive

```
// namespace2.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

using namespace std; // make all names in std namespace available!

int main() {

   double x = 3;
   double y = 4;

   double z1 = physics::mean(x,y);
   cout << "physics::mean(" << x << "," << y << ") = " << z1
             << endl;

   return 0;
}
```

Provide default namespace for un-qualified names

Compiler looks for **`cout`** and **`endl`** first

if not found looks for **`std::cout`** and **`std::endl`**;

```
$ g++ -o namespace2 namespace2.cc
$ ./namespace2.exe
physics::mean(3,4) = 3.5
```

Be careful with **using** directive!

```
// namespaceBad2.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
  double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

using namespace foobar;
using namespace physics;
using namespace std;

int main() {
   double x = 3;
   double y = 4;

   double z1 = mean(x,y);
   double z2 = mean(x,y);

   return 0;
}
```

Ambiguous use of method **mean**!

Is it in **foobar** or in **physics**?

```
$ g++ -o namespaceBad2 namespaceBad2.cc
namespaceBad2.cc: In function `int main()':
namespaceBad2.cc:21: error: call of overloaded `mean(double&, double&)' is ambiguous
namespaceBad2.cc:5: note: candidates are: double physics::mean(const double&, const double&)
namespaceBad2.cc:9: note:  double foobar::mean(const double&, const double&)
namespaceBad2.cc:25: error: call of overloaded `mean(double&, double&)' is ambiguous
namespaceBad2.cc:5: note: candidates are: double physics::mean(const double&, const double&)
namespaceBad2.cc:9: note:  double foobar::mean(const double&, const double&)
```

Some tips on `using` directive

- Never use using directive in header files!
  - These can be included in other files that do not want to use default namespaces specified by you!
  - Limit use of using directive to the scope you need

```
// namespace3.cc
#include <iostream>

namespace physics {
  double mean(const double& a, const double& b) {
    return (a+b)/2.;
  }
}

void printMean(const double& a, const double& b) {
  double z1 = physics::mean(a,b);

  using namespace std; // using std namespace within this method!
  cout << "physics::mean(" << a << "," << b << ") = " << z1 << endl;
}

int main() {

   double x = 3;
   double y = 4;
   printMean(x,y);

   cout << "no namespace available in the main!" << endl;
   return 0;
}
```

Namespace defined only within printMean

```
$ g++ -o namespace3 namespace3.cc
namespace3.cc: In function `int main()':
namespace3.cc:23: error: `cout' undeclared (first use this function)
namespace3.cc:23: error: (Each undeclared identifier is reported only
once for each function it appears in.)
namespace3.cc:23: error: `endl' undeclared (first use this function)
```

No default namespace in the main()

Another Example on Scopes

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

using std::cout;
using std::endl;

int main() {

  double x = 1.2;

  cout << "in main before scope, x: " << x << endl;

  { // just a local scope
     x++;
     cout << "in local scope before int, x: " << x << endl;

     int x = 4;
     cout << "in local scope after int, x: " << x << endl;
  }

  cout << "in main after local scope, x: " << x << endl;

  return 0;
}
```

Another way to declare ONLY classes we are going to use instead of entire namespace

```
$ g++ -o scope scope.cc
$ ./scope
in main before scope, x: ???
in local scope before int, x: ???
in local scope after int, x: ???
in main after local scope, x: ???
```

What do you think the output is going to be?

Another Example on Scopes

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

using std::cout;
using std::endl;

int main() {

  double x = 1.2;

  cout << "in main before scope, x: " << x << endl;

  { // just a local scope
     x++;
     cout << "in local scope before int, x: " << x << endl;

     int x = 4;
     cout << "in local scope after int, x: " << x << endl;
  }

  cout << "in main after local scope, x: " << x << endl;

  return 0;
}
```

Another way to declare ONLY classes we are going to use instead of entire namespace

Changed value of x from main scope

Define new variable in this scope

Back to the main scope

```
$ g++ -o scope scope.cc
$ ./scope
in main before scope, x: 1.2
in local scope before int, x: 2.2
in local scope after int, x: 4
in main after local scope, x: 2.2
```

Arrays

- Arrays can be defined for any built-in or user types (classes)

```
// vect3.cc
#include <iostream>
using namespace std;

int main() {


   float vect[3] = {0.4,1.34,56.156}; // vector of int
   float v2[3];
   float v3[] = { 0.9, -0.1, -0.65}; // array of size 3

   for(int i = 0; i<3; ++i) {
     cout << "i: " << i << "\t"
          << "vect[" << i << "]: " << vect[i] << "    \t"
          << "v2[" << i << "]: " << v2[i] << " \t"
          << "v3[" << i << "]: " << v3[i]
           << endl;
   }

   return 0;
}
```

Index of arrays starts from 0 !!

v2[0] is the first elements of array **v2** of size 3.

v2[2] is the last element of **v2**

What happened to **v2**?

```
$ g++ -o vect3 vect3.cc
$ ./vect3
i: 0    vect[0]: 0.4        v2[0]: 5.34218e+36    v3[0]: 0.9
i: 1    vect[1]: 1.34       v2[1]: 2.62884e-42    v3[1]: -0.1
i: 2    vect[2]: 56.156     v2[2]: 3.30001e-39    v3[2]: -0.65
```

Example of Bad non-initialized Arrays

```
// vect1.cc
#include <iostream>
#include <cmath>

using namespace std;


int main() {

   float vect[3]; // no initialization

   cout << "printing garbage since vector not initialized" << endl;
   for(int i=0; i<3; ++i) {
     cout << "vect[" << i << "] = " << vect[i]
          << endl;
   }

   vect[0] = 1.1;
   vect[1] = 20.132;
   vect[2] = 12.66;

   cout << "print vector after setting values" << endl;
   for(int i=0; i<3; ++i) {
     cout << "vect[" << i << "] =    " << vect[i] << "\t"
          << "sqrt( vect[" << i << "] ) = " << sqrt(vect[i])
          << endl;
   }

   return 0;
}
```

```
$ ./vect1
printing garbage since vector not initialized
vect[0] = 2.62884e-42
vect[1] = NaN
vect[2] = 0
print vector after setting values
vect[0] =    1.1        sqrt( vect[0] ) = 1.04881
vect[1] =    20.132     sqrt( vect[1] ) = 4.48687
vect[2] =    12.66      sqrt( vect[2] ) = 3.55809
```

Another bad example of using arrays

```
// vect2.cc
#include <iostream>
using namespace std;

int main() {


   float vect[3] = {0.4,1.34,56.156}; // vector of int
   float v2[3]; // use default value 0 for each element
   float v3[] = { 0.9, -0.1, -0.65, 1.012, 2.23, -0.67, 2.22 }; // array of size 7

   for(int i = 0; i<5; ++i) {
     cout << "i: " << i << "\t"
          << "vect[" << i << "]: " << vect[i] << "  \t"
          << "v2[" << i << "]: " << v2[i] << "  \t"
          << "v3[" << i << "]: " << v3[i]
           << endl;
   }

   return 0;
}
```

Accessing out of range component!

```
$ g++ -o vect2 vect2.cc
$ ./vect2
i: 0    vect[0]: 0.4            v2[0]: 5.34218e+36      v3[0]: 0.9
i: 1    vect[1]: 1.34           v2[1]: 2.62884e-42      v3[1]: -0.1
i: 2    vect[2]: 56.156         v2[2]: 3.30001e-39      v3[2]: -0.65
i: 3    vect[3]: 5.60519e-45    v2[3]: 1.57344e+20      v3[3]: 1.012
i: 4    vect[4]: 1.72441e+20    v2[4]: 0.4              v3[4]: 2.23
```

Functions and Methods

- A function is a set of operations to be executed
  - Typically there is some input to the function
  - Usually functions have a return value
  - Functions not returning a specific type are **`void`**

```
// func1.cc
#include <iostream>

double pi() {
  return 3.14;
}

void print() {
  std::cout << "void function print()" << std::endl;
}

int main() {

   std::cout << "pi: " << pi() << std::endl;
   print();

   return 0;
}
```

```
$ g++ -o func1 func1.cc
$ ./func1
pi: 3.14
void function print()
```

Functions must be declared before being used

```
// func2.cc
#include <iostream>

double pi() {
  return 3.14;
}

int main() {

   std::cout << "pi: " << pi() << std::endl;
   print();

   return 0;
}

void print() {
  std::cout << "void function print()" << std::endl;
}
```

Compiler does not know what the name **`print`** stands for!

No declaration at this point!

```
$ g++ -o func2 func2.cc
func2.cc: In function `int main()':
func2.cc:11: error: `print' undeclared (first use this function)
func2.cc:11: error: (Each undeclared identifier is reported only
 once for each function it appears in.)
func2.cc: In function `void print()':
func2.cc:16: error: `void print()' used prior to declaration
```

Definition can be elsewhere!

```
// func3.cc
#include <iostream>

double pi() {
  return 3.14;
}

extern void print(); // declare to compiler print() is a void method

int main() {

   std::cout << "pi: " << pi() << std::endl;
   print();

   return 0;
}

// now implement/define the method void print()
void print() {
  std::cout << "void function print()" << std::endl;
}
```

```
$ g++ -o func3 func3.cc
$ ./func3
pi: 3.14
void function print()
```

Pointers and References

- A variable is a label assigned to a location of memory and used by the program to access that location

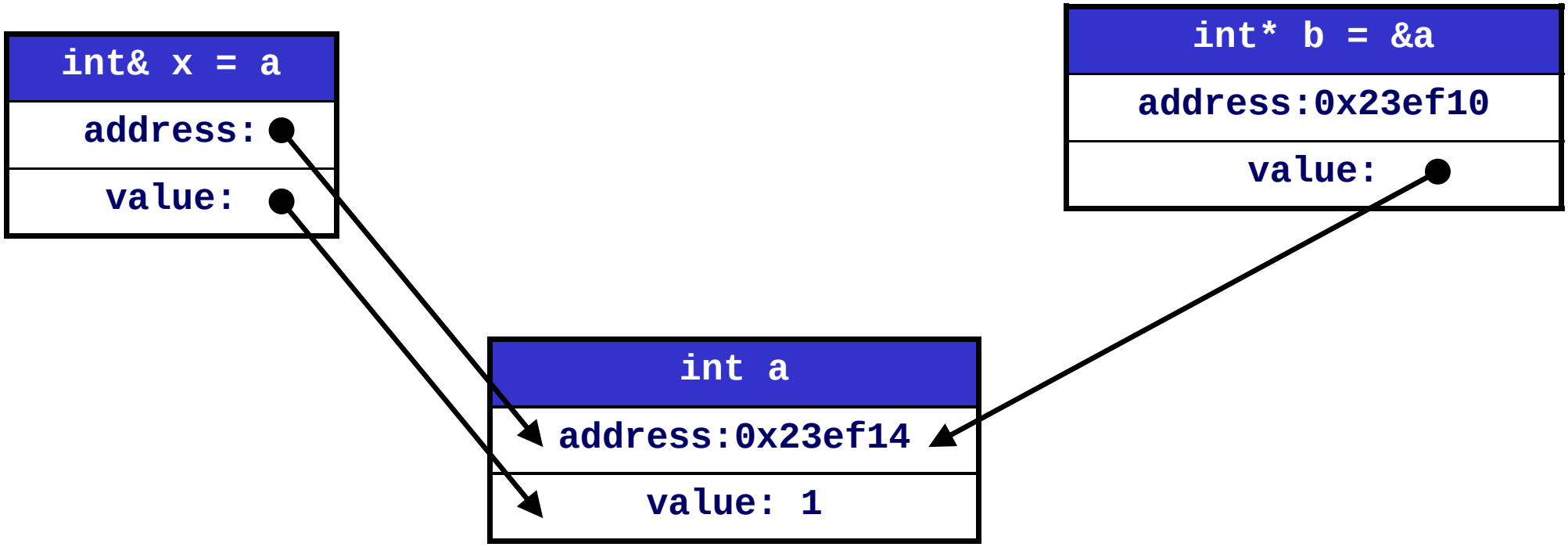
`int a` → 4 bytes==32bit of memory

```
// Pointers.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

   int a; // a is a label for a location of memory dtor'ing an int value

   cout << "Insert value of a: ";
   cin >> a; // store value provided by user
             // in location of memory held by a

   int* b; // b is  a pointer to variable of
           // type a

   b  = &a; // value of b is the address of memory
            // location assigned to a

   cout << "value of a: " << a << endl;
   cout << "address of a: " << b << endl;

   return 0;
} // end of main
```

Same location in memory but different values!

```
$ g++ -o Pointers Pointers.cpp
$ ./Pointers
Insert value of a: 3
value of a: 3
address of a: 0x23ef14
$ ./Pointers
Insert value of a: 1.2
value of a: 1
address of a: 0x23ef14
```

Pointers and References



- **x** is a reference to **a**
  - A different name for the same physical location in memory
  - Using **x** or **a** is exactly the same!
- **b** is a pointer to location of memory named **x** or **a**

Pointers and References

```
// refs.cpp
#include <iostream>
using namespace std;

int main() {

   int a = 1;

   int* b; // b is  a pointer to variable of type int

   b  = &a; // value of b is the address of memory location assigned to a

   int& x = a; //

   cout << "value of a: " << a
        << ", address of a, &a: " << &a
        << endl;

   cout << "value of x: " << x
        << ", address of x, &x: " << &x
        << endl;

   cout << "value of b: " << b
        << ", address of b, &b: " << &b
        << ", value of *b: " << *b
        << endl;

   return 0;
}
```

```
$ ./refs
value of a: 1, address of a, &a: 0x23ef14
value of x: 1, address of x, &x: 0x23ef14
value of b: 0x23ef14, address of b, &b: 0x23ef10, value of *b: 1
```

Using pointers and references

```
// refs2.cpp
#include <iostream>
using namespace std;

int main() {

   int a = 1;

   int* b = &a;
   *b = 3;

   cout << "value of a: " << a
        << ", address of a, &a: " << &a
        << endl;

   int& x = a;
   x = 45;

   cout << "value of a: " << a
        << ", address of a, &a: " << &a
        << endl;

   return 0;
}
```

Change value of a with pointer b

Change value of a with reference x

```
$ g++ -o refs2 refs2.cc
$ ./refs2
value of a: 3, address of a, &a: 0x23ef14
value of a: 45, address of a, &a: 0x23ef14
```

Bad and Null Pointers

- Pointers can point to invalid locations in memory

```
// badptr1.cpp
#include <iostream>
using namespace std;

int main() {

   int* b; // b is  a pointer to varible of type int

   int vect[3] = {1,2,3}; // vector of int

   int* c; // non-initialized pointer
   cout << "c: " << c << ", *c: " << *c <<endl;

   for(int i = 0; i<3; ++i) {
     c = &vect[i];
     cout << "c = &vect[" << i << "]: " << c << ", *c: " << *c << endl;
   }

   // bad pointer
   c++;
   cout << "c: " << c << ", *c: " << *c <<endl;

   // null pointer causing trouble
   c = 0;
   cout << "c: " << c << endl;
   cout << "*c: " << *c <<endl;

   return 0;
}
```

What is the size of an int in memory?

No problem compiling

```
$ g++ -o badptr1 badptr1.cc
$ ./badptr1
c: 0x7c90d592, *c: -1879046974
c = &vect[0]: 0x23eef0, *c: 1
c = &vect[1]: 0x23eef4, *c: 2
c = &vect[2]: 0x23eef8, *c: 3
c: 0x23eefc, *c: 1627945305
c: 0
Segmentation fault (core dumped)
```

Crash at runtime