Constants, Pointers and Arrays
Introduction to Class

Shahram Rahatlou

SAPIENZA
UNIVERSITÀ DI ROMA

Corso di Programmazione++

Roma, 23 March 2008

Today's Lecture

- More on pointers and references
  - Arrays and pointers
- Constants in C++
- Function interface
  - Using constant references instead of pointers!
- Introduction to classes in C++

Constants

- C++ allows to ensure value of a variable does not change within its scope
  - Can be applied to variables, pointers, references, vectors etc.
  - Constants must be ALWAYS initialized since they can't change at a later time!

```
// const1.cpp

int main() {

   const int a = 1;
   a = 2;

   const double x;

   return 0;
}
```

```
$ g++ -o const1 const1.cc
const1.cc: In function `int main()':
const1.cc:6: error: assignment of read-only variable `a'
const1.cc:8: error: uninitialized const `x'
```

Constant Pointer

```
// const2.cpp

int main() {

   int a = 1;
   int * const b = &a; // const pointer to int

   *b = 5; // OK. can change value of what b points to

   int c = 3;
   b = &c; // Not OK. assign new value to c

   return 0;
}
```

Read from right to left:
`int * const b`:

`b` is a constant pointer to int

```
$ g++ -o const2 const2.cc
const2.cc: In function `int main()':
const2.cc:11: error: assignment of read-only variable `b'
```

Pointer to Constant

a is not a constant!
But we can treat it as such when pointing to it

Read from right to left:
`const int * b`:

b is a pointer to constant int

```
// const3.cpp

int main() {

   int a = 1;
   const int * b = &a; // pointer to const int

   int c = 3;
   b = &c; // assign new value to c ... OK!

   *b = 5; // assign new value to what c point to ... NOT OK!

   return 0;
}
```

```
$ g++ -o const3 const3.cc
const3.cc: In function `int main()':
const3.cc:11: error: assignment of read-only location
```

NB: the error is different!

Constant Pointer to Constant Object

- Most restrictive access to another variable
  - Specially when used in function interface
- Can not change neither the pointer nor what it points to!

```
// const4.cpp

int main() {

   float a = 1;
   const float * const b = &a; // const pointer to const float

   *b = 5; // Not OK. can't change value of what b points to

   float c = 3;
   b = &c; // Not OK. can't change what b points to!

   return 0;
}
```

```
$ g++ -o const4 const4.cc
const4.cc: In function `int main()':
const4.cc:8: error: assignment of read-only location
const4.cc:11: error: assignment of read-only variable
```

Arrays and Pointers

- The name of the array is a pointer to the first element of the array

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

int main() {

   int vect[3] = {1,2,3}; // vector of int
   int v2[3]; //what is the default value?
   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;
   int* c = vect;
   int* e = v2;

   for(int i = 0; i<5; ++i) {
     cout << "i: " << i << ", d = " << d << ", *d: " << *d;
     ++d;
     cout << ", c = " << c << ", *c: " << *c;
     ++c;
     cout << ", e = " << e << ", *e: " << *e << endl;
     ++e;
   }
   return 0;
}
```

What happened to **e**?

```
$ g++ -o array array.cc
$ ./array
i: 0, d = 0x23eec0, *d: 1, c = 0x23eef0, *c: 1, e = 0x23eee0, *e: -1
i: 1, d = 0x23eec4, *d: 2, c = 0x23eef4, *c: 2, e = 0x23eee4, *e: 2088773120
i: 2, d = 0x23eec8, *d: 3, c = 0x23eef8, *c: 3, e = 0x23eee8, *e: 2088772930
i: 3, d = 0x23eecc, *d: 4, c = 0x23eefc, *c: 1627945305, e = 0x23eeec, *e: 2089866642
i: 4, d = 0x23eed0, *d: 5, c = 0x23ef00, *c: 1876, e = 0x23eef0, *e: 1
```

Bad Use of Pointers

```
int vect[3] = {1,2,3};
int v2[3];
int v3[] = { 1, 2, 3, 4, 5, 6, 7 };
```

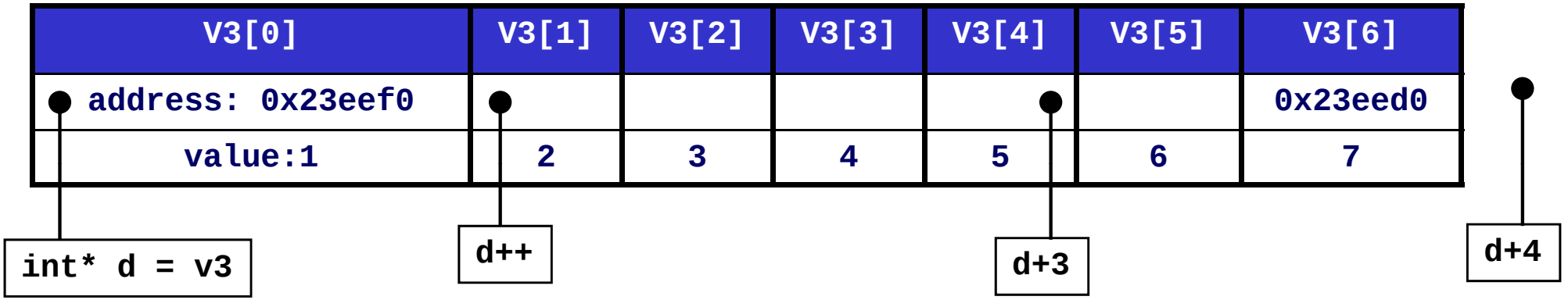
```
$ g++ -o array array.cc
$ ./array
i: 0, d = 0x23eec0, *d: 1, c = 0x23eef0, *c: 1, e = 0x23eee0, *e: -1
i: 1, d = 0x23eec4, *d: 2, c = 0x23eef4, *c: 2, e = 0x23eee4, *e: 2088773120
i: 2, d = 0x23eec8, *d: 3, c = 0x23eef8, *c: 3, e = 0x23eee8, *e: 2088772930
i: 3, d = 0x23eecc, *d: 4, c = 0x23eefc, *c: 1627945305, e = 0x23eeec, *e: 2089866642
i: 4, d = 0x23eed0, *d: 5, c = 0x23ef00, *c: 1876, e = 0x23eef0, *e: 1
```

V3[0]	V3[1]	V3[2]	V3[3]	V3[4]	V3[5]	V3[6]
address: 0x23eec0						0x23eed0
value:1	2	3	4	5	6	7

How many bytes in memory between v3[6] and v2[0] ?

V2[0]	V2[1]	V2[2]		vect[0]	vect[1]	Vect[2]
0x23eee0	0x23eee4	0x23eee8	0x23eeec	0x23eef0	0x23eef4	0x23eef8
-1	2	3		1	2	3

Pointer Arithmetic




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

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7
   int* d = v3;
   cout << "d = " << d << ", *d: " << *d <<endl;

   d++;
   cout << "d = " << d << ", *d: " << *d <<endl;

   d = d+3;
   cout << "d = " << d << ", *d: " << *d <<endl;

   d = d+4;
   cout << "d = " << d << ", *d: " << *d <<endl;

   return 0;
}
```

```
$ g++ -o ptr ptr.cc
$ ./ptr
d = 0x23eef0, *d: 1
d = 0x23eef4, *d: 2
d = 0x23ef00, *d: 5
d = 0x23ef10, *d: 1628803505
```

+ and – operators with Pointers

```
// ptr2.cc
#include <iostream>
using namespace std;

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;
   int*c = &v3[4];
   cout << "d = " << d << ", *d: " << *d <<endl;
   cout << "c = " << c << ", *c: " << *c <<endl;

   //int* e = c + d;  // not allowed

   cout << "c-d: " << c - d << endl;
   cout << "d-c: " << d - c << endl;

   //int* e = c-d; // wrong!

   int f = c - d;
   float g = c - d;

   cout << "f: " << f << " g: " << g << endl;

   int *  h = &v3[6] + (d-c);
   cout << "int *  h = &v3[6] + (d-c): " << h << " *h: " << *h << endl;

   return 0;
}
```

Arguments of Functions

- Arguments of functions can be passed in two different ways

```
// funcarg1.cc
#include <iostream>

using namespace std;

void emptyLine() {
  cout
  << "\n--------------------------------\n"
       << endl;
}
```

- By value
  - x is a local variable in f1()

```
void f1(double x) {
  cout << "f1: input value of x = "
       << x << endl;
  x = 1.234;
  cout << "f1: change value of x in f1(). x = "
       << x << endl;
}
```

- Pointer or reference
  - x is reference to argument used by caller

```
void f2(double& x) {
  cout << "f2: input value of x = "
       << x << endl;
  x = 1.234;
  cout << "f2: change value of x in f2(). x = "
        << x << endl;
}
```

Pointers and References in Functions

```
int main() {

   double a = 1.; // define a

   emptyLine();
   cout << "main: before calling f1, a = " << a<< endl;
   f1(a);  // void function
   cout << "main: after calling f1, a = " << a<< endl;

   emptyLine();
   cout << "main: before calling f2, a = " << a<< endl;
   f2(a);  // void function
   cout << "main: after calling f2, a = " << a<< endl;

   return 0;
}
```

f1 has no effect on variables in main

Because a is passed by value

x is a copy of a

f2 modifies the value of the variable in the main!

Because a is passed by reference

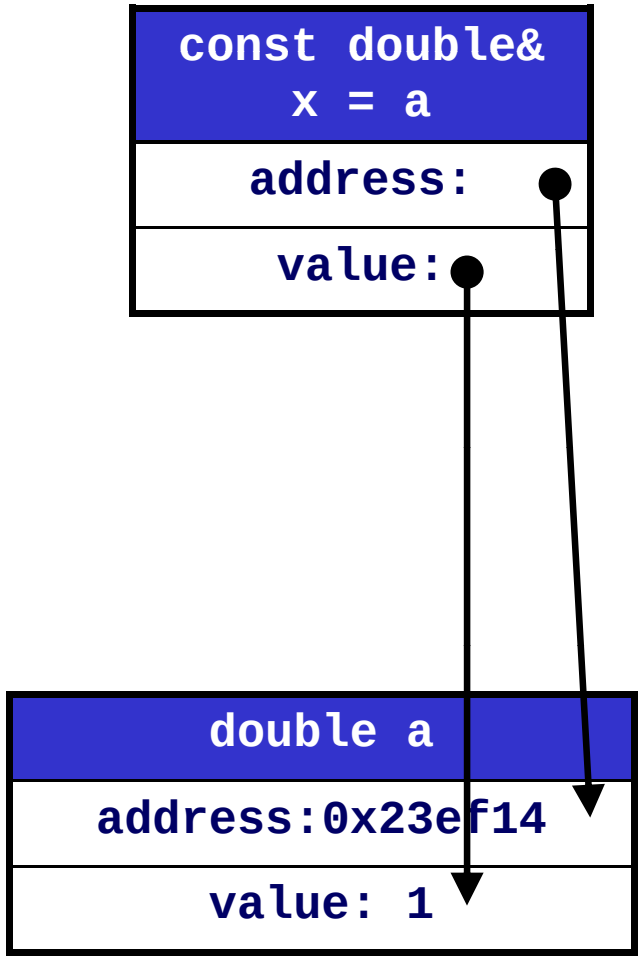
double& x = a;

```
$ ./funcarg1
---------------------------------
main: before calling f1, a = 1
f1: input value of x = 1
f1: change value of x in f1(). x = 1.234
main: after calling f1, a = 1
---------------------------------

main: before calling f2, a = 1
f2: input value of x = 1
f2: change value of x in f2(). x = 1.234
main: after calling f2, a = 1.234
```

Constant Pointers and References in Functions

- Useful in passing arguments of functions
  - Prevent undesired modification of input data

```
// funcarg2.cc
#include <iostream>

using namespace std;

void f2(const double& x) {
  cout << "f2: input value of x = "
       << x << endl;
  x = 1.234;
  cout << "f2: change value of x in f2(). x = "
        << x << endl;
}

int main() {

   double a = 1.;
   f2(a);

   return 0;
}
```




```
$ g++ -o funcarg2  funcarg2.cc
funcarg2.cc: In function `void f2(const double&)':
funcarg2.cc:9: error: assignment of read-only reference `x'
```

Pointers, References and Passing by Value in Functions

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

void computeMean(const double* data, int nData, double& mean) {
  mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
}

int main() {

   double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
   double average;

   computeMean( pressure, 7, average );

   cout << "average pressure: "
        << average << endl;
   return 0;
}
```

```
$ g++ -o mean mean.cc
$ ./mean
data: 0x23eed0, *data: 1.2
data: 0x23eed8, *data: 0.9
data: 0x23eee0, *data: 1.34
data: 0x23eee8, *data: 1.67
data: 0x23eef0, *data: 0.87
data: 0x23eef8, *data: 1.04
data: 0x23ef00, *data: 0.76
average pressure: 1.11143
```

Closer Look at **computeMean()**

```
void computeMean(const double* data, int nData, double& mean) {
  mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
}
```

- Input data passed as constant pointer
  - Good: can't cause trouble to caller! Integrity of data guaranteed
  - Bad: No idea how many data points we have!
- Number of data pointer passed by value
  - Simple int. No gain in passing by reference
  - Bad: separate variable from array of data. Exposed to user error
- Very bad: void function with no return type
  - Good: appropriate name. computeMean() suggests an action not a type

New implementation with Return Type

```
double mean(const double* data, int nData) {
  double mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
  return mean
}
```

- Make function return the computed mean
- New name to make it explicit function returns something
  - Not a rule, but simple courtesy to users of your code
- No need for variables passed by reference to be modified in the function
- Still exposed to user error...

Possible Problems with use of Pointers

```
// mean2.cc
#include <iostream>
using namespace std;

double mean(const double* data, int nData) {

  double mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
  return mean;
}

int main() {
   double pressure[] = { 1.2, 0.6, 1.8 }; // only 3 elements
   double average = mean(pressure, 4); // mistake!
   cout << "average pressure: " << average << endl;

   return 0;
}
```

```
$ g++ -o mean2 mean2.cc
$ ./mean2
data: 0x23eef0, *data: 1.2
data: 0x23eef8, *data: 0.6
data: 0x23ef00, *data: 1.8
data: 0x23ef08, *data: 8.48798e-314
average pressure: 0.9
```

Simple luck!
Additional value not changing the average!

No protection against possible errors!

What about computing other quantities?

- What if we wanted to compute also the standard deviation of our data points?

```
void computeMean(const double* data, int nData, double& mean, double& stdDev) {
 // two variables passed by reference to void function
 // not great. But not harmful.
}


double meanWithStdDev(const double* data, int nData, double& stdDev) {
  // error passed by reference to mean function! ugly!! anti-intuitive
}




double mean(const double* data, int nData) {
  // one method to compute only average
}


double stdDev(const double* data, int nData) {
  // one method to compute standard deviation
  // use mean() to compute average needed by std deviation
}
```

What if we had a new C++ type?

- Imagine we had a new C++ type called **`Result`** including data about both mean and standard deviation

- We could then simply do the following

```
Result mean(const double* data, int nData) {
  Result result;
  // do your calculation
  return result;
}
```

- This is exactly the idea of classes in C++!

Classes in C++

- A class is a set of data and functions that define the characteristics and behavior of an object
  - Characteristics also known as attributes
  - Behavior is what an object can do and is referred to also as its interface

Interface
or
Member Functions

Data members or attributes

```
class Result {
  public:

   // constructors
   Result() { }
   Result(const double& mean, const double& stdDev) {
     mean_ = mean;
     stdDev_ = stdDev;
   }

   // accessors
   double getMean() { return mean_; };
   double getStdDev() { return stdDev_; };

  private:
   double mean_;
   double stdDev_;
};
```

Don’t’s forget ; at the end of definition!

Using class **Result**

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

class Result {
  public:

   // constructors
   Result() { };
   Result(const double& mean, const double& stdDev) {
     mean_ = mean;
     stdDev_ = stdDev;
   }

   // accessors
   double getMean() { return mean_; };
   double getStdDev() { return stdDev_; };

  private:
   double mean_;
   double stdDev_;
};
```

```
int main() {

  Result r1;
  cout << "r1, mean: " << r1.getMean()
       << ", stdDev: " << r1.getStdDev()
       << endl;

  Result r2(1.1,0.234);
  cout << "r2, mean: " << r2.getMean()
       << ", stdDev: " << r2.getStdDev()
       << endl;

  return 0;
}
```

```
$ g++ -o results2 result2.cc
$ ./results2
r1, mean: NaN, stdDev: 8.48798e-314
r2, mean: 1.1, stdDev: 0.234
```

r1 is ill-defined. Why?

What is wrong with Result::Result() ?

C++ Data Types

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2 *or* 4 bytes	1 wide character

- Size is architecture dependent!
  - Difference between 32-bit and 64-bit machines
  - Above table refers to typical 32-bit architecture
- **`int`** is usually has size of 'one word' on a given architecture

- Four integer types: **`char`**, **`short`**, **`int`**, and **`long`**
  - Each type is at least as large as previous one
    - **`size(char) <= size(short) <= size(int) <= size(long)`**

- Long **`int == int`**; similarly **`short int == short`**

Size of Objects/Types in C++

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

int main() {
  char*       aChar = "c"; // char
  bool        aBool = true; // boolean
  short       aShort = 33; // short
  long        aLong  = 123421; // long
  int         anInt = 27; // integer
  float       aFloat = 1.043; // single precision
  double      aDbl = 1.243e-234; // double precision
  long double aLD = 0.432e245; // double precision

  cout << "char* aChar = " << aChar << "\tsizeof(" << "*char" << "): " << sizeof(*aChar) << endl;
  cout << "bool aBool = " << aBool << "\tsizeof(" << "bool" << "): " << sizeof(aBool) << endl;
  cout << "short aShort = " << aShort << "\tsizeof(" << "short" << "): " << sizeof(aShort) << endl;
  cout << "long aLong = " << aLong << "\tsizeof(" << "long" << "): " << sizeof(aLong) << endl;
  cout << "int aInt = " << anInt << "\tsizeof(" << "int" << "): " << sizeof(anInt) << endl;
  cout << "float aFloat = " << aFloat << "\tsizeof(" << "float" << "): " << sizeof(aFloat) << endl;
  cout << "double aDbl = " << aDbl << "\tsizeof(" << "double" << "): " << sizeof(aDbl) << endl;
  cout << "long double aLD = " << aLD << "\tsizeof(" << "long double" << "): " << sizeof(aLD) << endl;

  return 0;
}
```

```
$ g++ -o cpptypes cpptypes.cc
$ ./cpptypes
char* aChar = c                 sizeof(*char): 1
bool aBool = 1                  sizeof(bool): 1
short aShort = 33               sizeof(short): 2
long aLong = 123421             sizeof(long): 4
int aInt = 27                   sizeof(int): 4
float aFloat = 1.043            sizeof(float): 4
double aDbl = 1.243e-234        sizeof(double): 8
long double aLD = 4.32e+244     sizeof(long double): 12
```