Discussion of an example: an array of floating point numbers

P. Ronchese
Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

An example

Same details of an example will be discussed:
an array of floating point numbers with improved functionalities

- Dynamic memory handling
- Operator usage
- `const` functions/operators
- Copy constructor and assignment

Discussion

Some weakness of the first simple implementation will be shown

- Crashes induced by unforeseen usage
- Analysis of the bug
- Solutions for the error

Constructor, destructor and data members

- Constructor: takes the size as argument
- Data members:
  - array size
  - pointer to array content

```
class FloatArray {
 public:
  FloatArray(unsigned int n);
  ~FloatArray();
  ...
 private:
  unsigned int eltn;
  float* cont;
};
```

Functions and operators

- A function to return the size
- Two operators to access content by index:
  - one `const` to be used with `const` arrays, returning a reference to `const float` ,
  - one non-`const` to be used with non-`const` arrays.
  - the compiler takes care of calling the proper one

```
class FloatArray {
 public:
  ...
  unsigned int size() const;
  const
  float& operator[](unsigned int i) const;
  float& operator[](unsigned int i);
  ...
};
```

Other elements

- An exception `class` to flag out-of-range access
- A function to check index
- A `static` flag to enable debugging

```
class FloatArray {
 public:
  ...
  class OutOfRange {
   public:
    OutOfRange() {}
    ~OutOfRange() {}
  };
  bool checkRange(unsigned int i) const;
  static bool debug;
  ...
};
```

Memory allocation and deallocation

- The constructor takes care of memory allocation
- Destructor takes care of its deallocation

```
FloatArray::FloatArray(unsigned int n):
 eltn(n),
 cont(new float[eltn]) {
  if(debug)cout << "created "
                << cont << endl;
}
FloatArray::~FloatArray() {
  if(debug)cout << "delete "
                << cont << endl;
  delete[] cont;
}
```

Size and range

- The array size is simply a number
- An index is in range if it's $0 \leq i < eltn$

```
unsigned int FloatArray::size() const {
  return eltn;
}
bool FloatArray::checkRange(unsigned
                            int i) const {
  return (i<eltn);
}
```

Access to content

- Access to content requires a prior check of index
- The array content are accessed by reference, so they can be modified (unless they're "`const`")

```
const
float& FloatArray::operator[](unsigned
                              int i) const {
  if(!checkRange(i))throw OutOfRange();
  return cont[i];
}
float& FloatArray::operator[](unsigned
                              int i) {
  if(!checkRange(i))throw OutOfRange();
  return cont[i];
}
```

Test program

A simple testing program can be easily written:
create a 5-element array and set them as 3.2 times their index

```
#include "FloatArray.h"
#include <iostream>
using namespace std;
int main() {
  unsigned int i;
  unsigned int n=5;
  FloatArray a(n);
  for(i=0;i<n;++i)a[i]=i*3.2;
  for(i=0;i<n;++i)cout << i << " "
                       << a[i] << endl;
  return 0;
}
```

Add a simple function

A simple function prints the array elements

```
void print(const FloatArray a) {
  unsigned int i;
  unsigned int n=a.size();
  for(i=0;i<n;++i)cout << i << " "
                       << a[i] << endl;
  cout << "***" << endl;
  return;
}
```

Test the function

The function can be called just after the construction and setting of "a"

```
#include "FloatArray.h"
#include <iostream>
using namespace std;
int main() {
  unsigned int i;
  unsigned int n=5;
  FloatArray a(n);
  for(i=0;i<n;++i)a[i]=i*3.2;
  print(a); // print a
  for(i=0;i<n;++i)cout << i << " "
                       << a[i] << endl;
  return 0;
}
```

Crash in the execution

Trying to run, the program crashes!

```
void print(const FloatArray a) {
  // "a" is passed by value:
  // a local copy is done, containing
  // a simple copy of "eltn" and "cont"
  unsigned int i;
  unsigned int n=a.size();
  for(i=0;i<n;++i)cout << i << " "
                       << a[i] << endl;
  return; // when the copy of "a" is deleted
}         // also "cont" is deleted,
          // and "a" gets a dangling pointer
```

- Simple copy of a pointer: “shallow copy”
- Copy of data referred to by a pointer: “deep copy”

Robustness improvement

- Default object copy performs a “shallow copy”
- Object copies could be hidden
  (mainly in function call and return)

Lowest-level protection: prevent object copying (in C++98/03)

```
class FloatArray {
 public:
   ...
 private: // any "hidden" copy will be
          // flagged by the compiler
          // as an error
  FloatArray(const FloatArray& a);
  FloatArray& operator=(const FloatArray& a);
   ...
};
```

Implicit constructor & assignment suppression

Remove copy constructor and assignment: (since C++11)

Default constructors and/or operators may be removed

```
class FloatArray {
 public:
   ...
   // any "hidden" copy will be
   // flagged by the compiler
   // as an error
   FloatArray(const FloatArray& a) = delete;
   FloatArray& operator=(const FloatArray& a)
                                    = delete;
   ...
};
```

Provide copy and assignment

- Object copies could be necessary
- Provide a copy constructor and assignment operator and make them public

```
class FloatArray {
 public:
   ...
   FloatArray(const FloatArray& a);
   FloatArray& operator=(const FloatArray& a);
   // object copying will be possible but
   // the corresponding constructor and
   // operator are to be provided
 private:
   ...
   void copy(const FloatArray& a);
};
```

Deep copy

Deep copy can be “encapsulated” in a specific function

```
void FloatArray::copy(const FloatArray& a) {
  if(cont==a.cont)return; // skip "a=a" cases
  delete[] cont; // delete old content
  cont=new float[eltn=a.eltn];
  // copy size and allocate new memory
  float* pr=a.cont+eltn; // copy elements
  float* pl=  cont+eltn; // one by one
  while(pl>cont)*--pl=*--pr;
  return;
}
```

Implement constructor and assignment

The "`copy`" function can be used in constructor and assignment

```
FloatArray::FloatArray(const FloatArray& a):
 cont(nullptr) { // set "cont" at nullptr so
  copy(a); // that a "delete[]" has no effect
}
FloatArray& FloatArray::operator=(
                          const FloatArray& a) {
  copy(a);
  return *this; // return the just-copied
}               // object as value
```

Temporary objects

Objects used in only one instruction

Some objects can be destroyed just after being created:

- returned by functions
- created only to be used as operands of function arguments

```
Pippo f();
void g(Pippo p);


  ...
  Pippo q = f();
  g(Pippo(...));
  ...
```

Construction and assignment from temporary objects

C++11 only

Resources owned by temporary objects can be simply “moved” to avoid the cost of copying: “move” constructors and assignments taking a “rvalue reference” as argument.

```
class FloatArray {
 public:
  ...
  FloatArray(FloatArray&& a);
  FloatArray& operator=(FloatArray&& a);
  // construct or assign
  // from temporary objects
 private:
  ...
  void move(FloatArray& a);
};
```

Resource transportation

Resource move can be “encapsulated” in a specific function

```
void FloatArray::move(FloatArray& a) {
  if(cont==a.cont)return; // skip "a=a" cases
  delete[] cont; // delete old content
  // copy pointer and size
  cont=a.cont;
  eltn=a.eltn;
  // reset pointer and size in source object
  a.eltn=0;
  a.cont=nullptr;
  return;
}
```

Implement move constructor and assignment (C++11 only)

The "move" function can be used in constructor and assignment

```
FloatArray::FloatArray(FloatArray&& a):
 cont(nullptr) { // set "cont" at nullptr
                 // so that a "delete[]" has
  move(a);       // no effect
}
FloatArray& FloatArray::operator=(
                          FloatArray&& a) {
  move(a);
  return *this; // return the just-moved
}               // object as value
```

Summary

Recommendation

- Object copies could hide inside an apparently simple program
- Default copy is often unreliable,
  mainly when objects contain pointers
- Unless an object is very simple
  (only native variables, no pointers), add protection
  - Declare (at least) copy constructor and assignment operator as `private` (or `deleted` in C++11)
  - Provide, if needed, an explicit implementation of them
  - "The rule of 3": if any one among
    destructor, copy constructor and assignment operator
    is defined, then all the 3 should be defined as well
- C++11 allows transferring the resources from temporary objects (move contructor and assignment): "rule of 5"

Appendix: RVO

Return Value Optimization

When possible, the compiler may generate code
to avoid copying (or moving) of objects:
temporary objects are elided in the creation
or assignment of objects in the client.
BEWARE: side effects in constructor/destructor
may be difficult to control!