Dynamically Allocated Data Members, Overloading Operators

Shahram Rahatlou

Computing Methods in Physics

http://www.roma1.infn.it/people/rahatlou/cmp/

Anno Accademico 2018/19

SAPIENZA
UNIVERSITÀ DI ROMA

Today's Lecture

- More on dynamically allocated data members
- Operators in C++
- Overloading operators
- special pointer **`this`**
- Examples
  - Class Datum

Dynamically Allocated Data Members

```
#ifndef Worker_h
#define Worker_h

#include "Algo.h"

class Worker {
  public:
    Worker();
    Worker(const& Worker w);
    Worker(Algo* algo);
    Worker(const Algo& algo);

    ~Worker();

    void setAlgo(Algo* algo);

  private:
    Algo* alg_;
};
#endif
```

```
#ifndef Algo_h
#define Algo_h

class Algo {
 public:
  Algo() { params_ = 0; }
  Algo(const Algo& algo) {
   params_ = algo.params_;
  }

  double compute(const double& arg) const;

  private:
   int params_;
};
#endif
```

- Data member is a pointer!
- How would you implement class Worker ?
- Why so many different constructors?

Possible Implementation of `Worker`

```
#include "Worker.h"

Worker::Worker() {
  alg_ = new Algo();
}
Worker::Worker(Algo* algo) {
  alg_ = algo;
}

Worker::Worker(const& Worker w) {
  alg_ = w.alg_;
}

Worker::Worker(const Algo& algo) {
  alg_ = new Algo(algo);
}

Worker::~Worker() {
  delete alg_;
}

void Worker::setAlgo(Algo* algo) {
  cout << "Worker::setAlgo changed
alg_ from "
    << alg_
    << " to " << algo << endl;
    alg_ = algo;
}
```

- Implementation far from being OK
- Identify errors and suggest solution

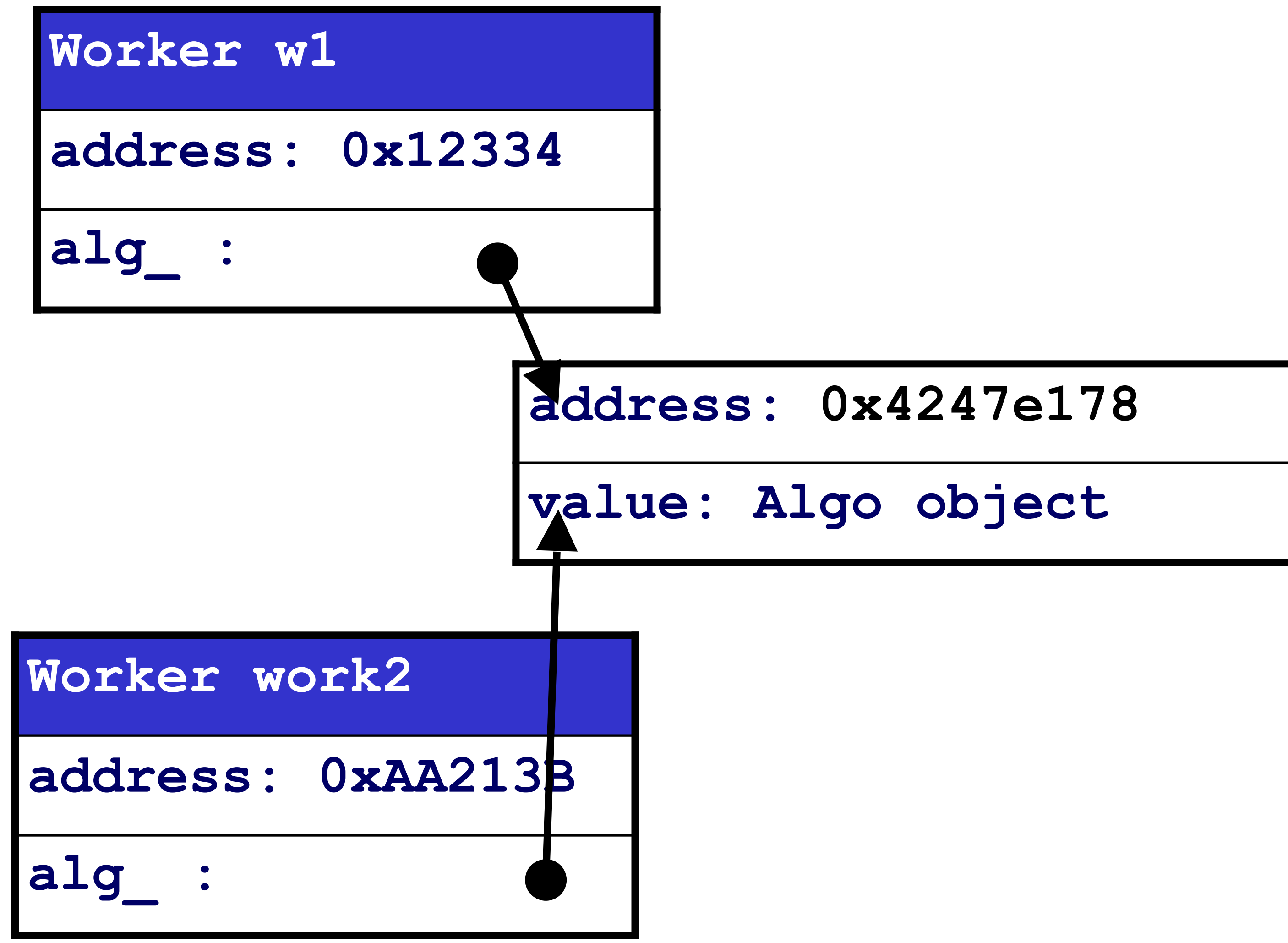
```
// app0.cpp
// testing Worker class

#include <iostream>
using namespace std;

#include "Worker.h"
#include "Algo.h"

int main() {

  Worker work1;
  // dynmic allocation
  Algo* alg1 = new Algo();
  work1.setAlgo( alg1 );
  work1.setAlgo( new Algo() );
  delete alg1;

  return 0;
}
```

```
$ g++ -Wall -o app0 app0.cpp Algo.cc Worker.cc
$ ./app0
Worker() alg_: 0x6a0290
Worker::setAlgo changed alg_ from 0x6a0290 to 0x6a06a8
Worker::setAlgo changed alg_ from 0x6a06a8 to 0x6a06b8
~Worker() deleting alg_: 0x6a06b8
```

Same data member for `w1` and `work2` : bug of feature?

```
#include "Worker.h"

Worker::Worker() {
  alg_ = new Algo();
  // even better: alg_ = 0;
}
Worker::Worker(Algo* algo) {
  alg_ = algo;
}

Worker::Worker(const& Worker w) {
  alg_ = w.alg_;
}

Worker::Worker(const Algo& algo) {
  alg_ = new Algo(algo);
}

Worker::~Worker() {
  delete alg_;
}

void Worker::setAlgo(Algo* algo) {
  alg_ = algo;
}
```

```
Worker w1  ( new Algo() );
Worker work2( w1 )
```



- Both object point to same dynamically allocated **`Algo`**
- Changing parameters of **`w1`** affects **`work2`**!

Possible Problem with Sharing pointers

```
#ifndef Worker_h
#define Worker_h

#include "Algo.h"

class Worker {
  public:
    Worker();
    Worker(const& Worker w);
    Worker(Algo* algo);
    Worker(const Algo& algo);

    ~Worker();

    void setAlgo(Algo* algo);

    Algo* algo()
         { return alg_;}

  private:
    Algo* alg_;
};
#endif
```

```
Worker w1 ( new Algo() );

// same algo used in work2
Worker work2( w1 );

// change params of algo of w1
w1.algo()->setParam(0, 1.23);
```

w1.algo() returns pointer to **w1::alg_**

Algo::setParam(i, value) is a method of class **Algo** to change value of i^{th} parameter

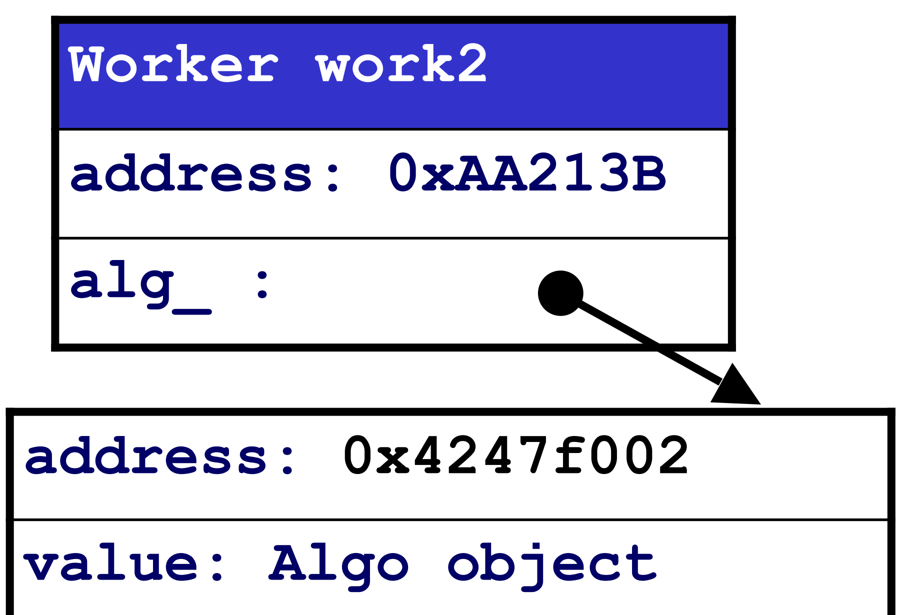
- Since both **w1** and **work2** point to same **Algo** object, the above code will change behavior for both **w1** and **work2**
- User of **work2** might not even know nor understand why his/her algorithm has changed!

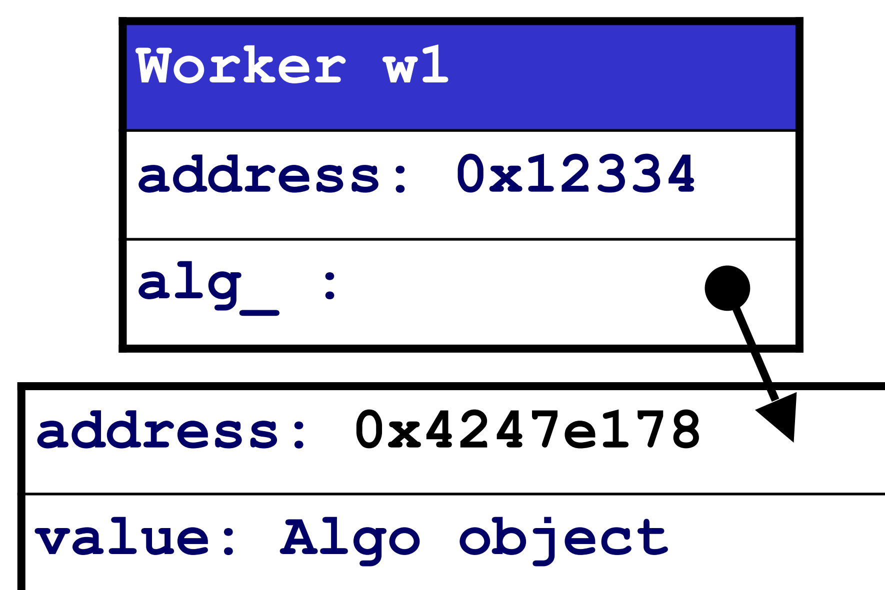
One Solution: one **Algo** for each **Worker**

```
#include "Worker.h"

Worker::Worker() {
  alg_ = new Algo();
}
Worker::Worker(Algo* algo) {
  alg_ = algo;
}

Worker::Worker(const& Worker w) {
  alg_ = new Algo( w.alg_ );
}

Worker::Worker(const Algo& algo) {
  alg_ = new Algo(algo);
}
```

```
Worker w1 ( new Algo() );
Worker work2( w1 )
```

- Same code as before but different behavior
- Instead of using the same object we clone `w1::alg_`
- `Work2::alg_` is a new dynamically allocated object that has the same parameters of `w1::alg_`
- Two independent object that can be configured separately





General guidelines for dynamically allocated members

- There are really no general solutions

- Very much depends on specific use case for individual classes

- If all workers MUST or should use the same algorithm then our first implementation was fine
  - But in general having object that can change without user explicitly calling any of its methods is a red flag pointing to weakness

- Very often objects must be fully independent from each other

Operators

Operation between Datum Objects

- Since Datum represents user data we could imagine having

```
Datum d1(-3.87,0.16);
Datum d2(6.55,2.1);

Datum d3 = d1.plus( d2 );

Datum d4 = d1.minus( d2 );

Datum d5 =
d1.product( d2 );
```

- These functions are easy to implement and provide behavior similar to doubles, ints, floats

- But they are functions not operators! They look different from what we are used to do with simple numbers

Operators

- C++ has a variety of built-in operators for built-in types

```
int i =8;
int j = 10;

int l = i + j;
int k = i * j;
```

- C++ allows you to implement such built-in operators also for user-defined types (classes!)

```
Datum d1(-3.87,0.16);
Datum d2(6.55,2.1);

Datum d3 = d1 + d2;
```

- This is called **overloading of operators**
  – We need to tell the compiler what to do when adding two Datum objects!

C++ Operators

- Binary operators require two operands
  - right-hand and left-hand operands

+	+=	<<=
-	-=	==
*	*=	!=
/	/=	<=
%	%=	>=
^	^=	&&
&	&=	||
|	|=	,
>	>>	()
<	<<	[]
=	>>=	->*

- Unary operators

+
-
*
&
->
~
!
++
--

Example of Overloaded Operator

```
class Datum {
  public:
    // interface same as before

    Datum operator+( const Datum& rhs ) const;

  private:
    // same data members
};
#endif
```

```
// app1.cpp
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );
  cout << "input data d1 and d2: " << endl;
  d1.print();
  d2.print();

  Datum d3 = d1 + d2;

  cout << "output d3 = d1+d2 " << endl;
  d3.print();

  Datum d4 = d1.operator+( d2 );
  d4.print();

  return 0;
}
```

```
#include "Datum.h"
#include <iostream>
#include <cmath>

// other member functions same as before

Datum Datum::operator+( const Datum& rhs ) const {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ +
                     (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
$ g++ -Wall -o app1 app1.cpp Datum.cc
$ ./app1
input data d1 and d2:
datum: 1.2 +/- 0.3
datum: -0.4 +/- 0.4
output d3 = d1+d2
datum: 0.8 +/- 0.5
datum: 0.8 +/- 0.5
```

Understanding Overloading of Operators: the syntax

```
Datum Datum::operator+( const Datum& rhs ) const {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ +
                     (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

- `operator+` is a member function of class Datum
  - it returns a Datum object in output by value
  - it has one argument called rhs
  - it is a constant function: can not modify the object it is applied to

- In this example we assume data points are not correlated
  - values are added
  - error on the sum is the sum in quadrature of the errors

Using Operators with Objects

- Operators can be called on objects exactly like any other member function of a class

```
Datum d1( 1.2, 0.3 );
Datum d2( -0.4, 0.4 );

Datum d4 = d1.operator+( d2 );
```

  - **`operator+`** is called on object **`d3`** with argument **`d2`** and result is stored in **`d4`**

- However, since they are operators, they can also be used like the operators for the built-in C++ types

```
Datum d1( 1.2, 0.3 );
Datum d2( -0.4, 0.4 );

Datum d3 = d1 + d2;
```

Operator versus Function

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
    Datum(double x=1.0, double y=0.0);
    Datum(const Datum& datum);
    ~Datum() { };

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    Datum operator+( const Datum& rhs ) const;
    Datum sum( const Datum& rhs ) const;


  private:
    double value_;
    double error_;
};
#endif
```

```
Datum Datum::operator+( const Datum& rhs) const {

  // sum of central values
  double val = value_ + rhs.value_;
  // assume data are uncorrelated. sum in quadrature of errors
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}


Datum Datum::sum( const Datum& rhs) const {

  // sum of central values
  double val = value_ + rhs.value_;
  // assume data are uncorrelated. sum in quadrature of errors
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  Datum d3 = d1 + d2;
  Datum d4 = d1.sum( d2 );
  d3.print();
  d4.print();

  return 0;
}
```

```
$ g++ -Wall -o app2 app2.cpp Datum.cc
datum: 0.8 +/- 0.5
datum: 0.8 +/- 0.5
```

Why is operator+ constant?

- As usual, if not declared constant you can't call it constant objects

```
Datum Datum::operator+( const Datum& rhs ) {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ +
                     (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
// app3.cpp
#include <iostream>
using namespace std;

#include "Datum1.h"


int main() {
  const Datum d1( 1.2, 0.3 );
  const Datum d2( -0.4, 0.4 );

  Datum d3 = d1 + d2;
  d3.print();

  return 0;
}
```

```
$ g++ -Wall -o app3 app3.cpp Datum1.cc
app3.cpp: In function `int main()':
app3.cpp:12: error: passing `const Datum' as `this' argument of `
Datum Datum::operator+(const Datum&)' discards qualifiers
```

- Adding constant objects is perfectly reasonable
  – Your mistake! operator+ MUST be constant!

Rules of the Game: What You Can or Cannot Do

- You can overload any of the built-in C++ operators for your classes

- Overload operators for classes should mimic functionality of built-in operators for built-in types
  - operator * should not be implemented as a division!
  - Purpose of overloading operators is to extend the C++ language for custom user types (classes)
    - Overload only operators that are meaningful
    - What is the meaning of ++ operator for class Datum ?

- You CANNOT
  - create new operators but only overload existing ones
  - change meaning of operators for built-in types
  - change parity of operators: a binary operator can not be overloaded to become a unary operator

Assignment Operator `Datum::operator=(const Datum& rhs)`

```
class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    ~Datum() { };

    double value() const { return value_; }
    double error() const { return error_; }
    double significance() const;
    void print() const;

    Datum operator+( const Datum& rhs ) const;
    Datum sum( const Datum& rhs ) const;

    const Datum& operator=( const Datum& rhs );

  private:
    double value_;
    double error_;
};
```

```
const Datum& Datum::operator=(const Datum& rhs) {
  value_ = rhs.value_;
  error_ = rhs.error_;

  return *this;
}
```

remember **this** ?

```
// app4.cpp
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
   const Datum d1( 1.2, 0.3 );
   Datum d2( -0.4, 0.4 );

   Datum d3 = d1;
   d3.print();

   Datum d4;
   d4.operator=(d2);
   d4.print();

   return 0;
}
```

```
$ g++ -Wall -o app4 app4.cpp Datum.cc
$ ./app4
datum: 1.2 +/- 0.3
datum: -0.4 +/- 0.4
```

This operator cannot be constant…
We need to modify the object it is applied to!

Another Example of Use of Assignment Operator

```
// app5.cpp
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  Datum d1( 1.2, 0.3 );
  const Datum d2 = d1; // OK.. init the constant

  Datum d3( -0.2, 1.1 );
  d2 = d3; // error!

  return 0;
}
```

```
$ g++ -Wall -o app5 app5.cpp Datum.cc
app5.cpp: In function `int main()':
app5.cpp:13: error: passing `const Datum' as `this' argument of
`const Datum& Datum::operator=(const Datum&)' discards qualifiers
```

Special Pointer **`this`** in a Class

- Special pointer provided in C++

- Allows an object to get a pointer to itself from within any member function of the class

- Useful when an object (instance of a class) has to compare itself with other objects

- Particularly useful for overloading operators
  - many operators are used to modify an object: =, +=, *=, etc.
  - All these operators should return an object of the type of the class
  - When overloading you want an object to modify itself AND return itself

One More Example of **`this`**

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

class Example {
  public:
   Example() { name_ = ""; }
   Example(const string& name);
   void printSelf() const;
  private:
   string name_;
};

Example::Example(const string& name) {
  name_ = name;
}


void
Example::printSelf() const {
  cout << "name: " << name_
       << "\t this: " << this
       << endl;
}
```

```
int main() {
  Example ex1("ex1");
  ex1.printSelf();

  cout << "&ex1: " << &ex1 <<
endl;

  return 0;
}
```

```
$ g++ -o this this.cpp
$ ./this
name: ex1         this: 0x23eef0
&ex1: 0x23eef0
```

`this` is the reference of ex1 accessible from within ex1

Exercise

- Complete class Datum with remaining operators and make sure errors are treated correctly (assuming no correlation)
  - for example * and /
  - add operator to multiply Datum by float

```
Datum d1(-1.1, 0.2);
Datum d2 = d1 * 3.5;
```

  - How can you take into account correlations between Datum objects?

- Write a new class Vector3D and implement following methods
  - + and - operators
  - = operator
  - operator to multiply or divide by a float
  - distance() and angle()
  - scalarProduct() and vectorProduct()