

Dynamic Memory Management
Class Destructors
constant member functions

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

- ▷ Use of `std::vector<Type T>`
- ▷ Dynamic allocation of memory
- ▷ Destructors of a class
- ▷ Constant member functions
- ▷ Default arguments for member functions

Class `std::vector<T>`

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

int main() {

    std::vector<double> vals;
    vals.push_back(1.3);
    vals.push_back(-2.1);

    std::vector<double> errs;
    errs.push_back(0.2);
    errs.push_back(0.3);

    std::vector<Datum> data;
    data.push_back( Datum(1.3, 0.2) );
    data.push_back( Datum(-2.1, 0.3) );

    std::cout << "# dati:: " << data.size() << std::endl;

    // using traditional loop on an array
    int i=0;
    std::cout << "Using [] operator on vector" << std::endl;
    for(i=0; i< data.size(); ++i) {
        std::cout << "i: " << i
                    << "\t data: " << data[i].value() << " +/- " << data[i].error()
                    << std::endl;
    }

    // using vector iterator
    i=0;
    std::cout << "std::vector<T>::iterator " << std::endl;
    for(std::vector<Datum>::iterator d = data.begin(); d != data.end(); d++) {
        //std::cout << "d: " << d << std::endl;
        i++;
        std::cout << "i: " << i
                    << "\t data: " << d->value() << " +/- " << d->error()
                    << std::endl;
    }

    // using vector iterator
    i=0;
    std::cout << "C++11 extension feature " << std::endl;
    for(Datum dit : data) {
        i++;
        std::cout << "i: " << i
                    << "\t data: " << dit.value() << " +/- " << dit.error()
                    << std::endl;
    }

    return 0;
}
```

```
$ g++ -o app.exe vector1.cc Datum.cc
vector1.cc:45:17: warning: range-based for loop is a C++11 extension
      [-Wc++11-extensions]
    for(Datum dit : data) {
                    ^
1 warning generated.
$ ./app.exe
# dati:: 2
Using [] operator on vector
i: 0          data: 1.3 +/- 0.2
i: 1          data: -2.1 +/- 0.3
std::vector<T>::iterator
i: 1          data: 1.3 +/- 0.2
i: 2          data: -2.1 +/- 0.3
C++11 extension feature
i: 1          data: 1.3 +/- 0.2
i: 2          data: -2.1 +/- 0.3
```

Interface of `std::vector<T>`

<http://www.cplusplus.com/reference/vector/vector/>
<https://en.cppreference.com/w/cpp/container/vector>

Member functions

(constructor)	constructs the vector (public member function)
(destructor)	destructs the vector (public member function)
operator=	assigns values to the container (public member function)
assign	assigns values to the container (public member function)
get_allocator	returns the associated allocator (public member function)

Element access

at	access specified element with bounds checking (public member function)
operator[]	access specified element (public member function)
front	access the first element (public member function)
back	access the last element (public member function)
data (C++11)	direct access to the underlying array (public member function)

Iterators

begin cbegin	returns an iterator to the beginning (public member function)
end cend	returns an iterator to the end (public member function)
rbegin crbegin	returns a reverse iterator to the beginning (public member function)
rend crend	returns a reverse iterator to the end (public member function)

Capacity

empty	checks whether the container is empty (public member function)
size	returns the number of elements (public member function)
max_size	returns the maximum possible number of elements (public member function)
reserve	reserves storage (public member function)

Using `std::vector<T>` in functions

```
#include <vector>
Using std::vector;

Datum average(vector<float>& val,
vector<float>& err) {
    double mean = 0.;
    double meanErr(0.); // same as = 0.

    // loop over data
    // compute average

    Datum res(mean, meanErr);
    return res;
}
```

Constructor is called with arguments
Same behavior for `double` and `Datum`

Object `res` is like any other variable `mean` or `meanErr`
`res` simply returned as output to caller

```
#include <vector>
Using std::vector;

Datum average(vector<float>& val,
vector<float>& err) {
    double mean = 0.;
    double meanErr(0.); // same as =
0.

    // loop over data
    // compute average

    return Datum(mean, meanErr);
}
```

```
#include <vector>
Using std::vector;

double average(vector<float>& val) {
    double mean = 0.;
    // loop over data
    // compute average

    return mean;
}
```

Since `res` not really needed within function
we can just create it while returning the function
output

Dynamic Memory Allocation: **new** and **delete**

- C++ allows dynamic management memory at run time via two dedicated operators: **new** and **delete**
- **new**: allocates memory for objects of any built-in or user-defined type
 - The amount of allocated memory depends on the size of the object
 - For user-defined types the size is determined by the data members
- Which memory is used by **new**?
 - **new** allocated objects in the free store also known as heap
 - This is region of memory assigned to each program at run time
 - Memory allocated by **new** is unavailable until we free it and give it back to system via **delete** operator
- **delete**: de-allocates memory used by **new** and give it back to system to be re-used

Stack and Heap

```
// app7.cpp
#include <iostream>
using namespace std;

int main() {
    double* ptr1 = new double[100000];
    ptr1[0] = 1.1;

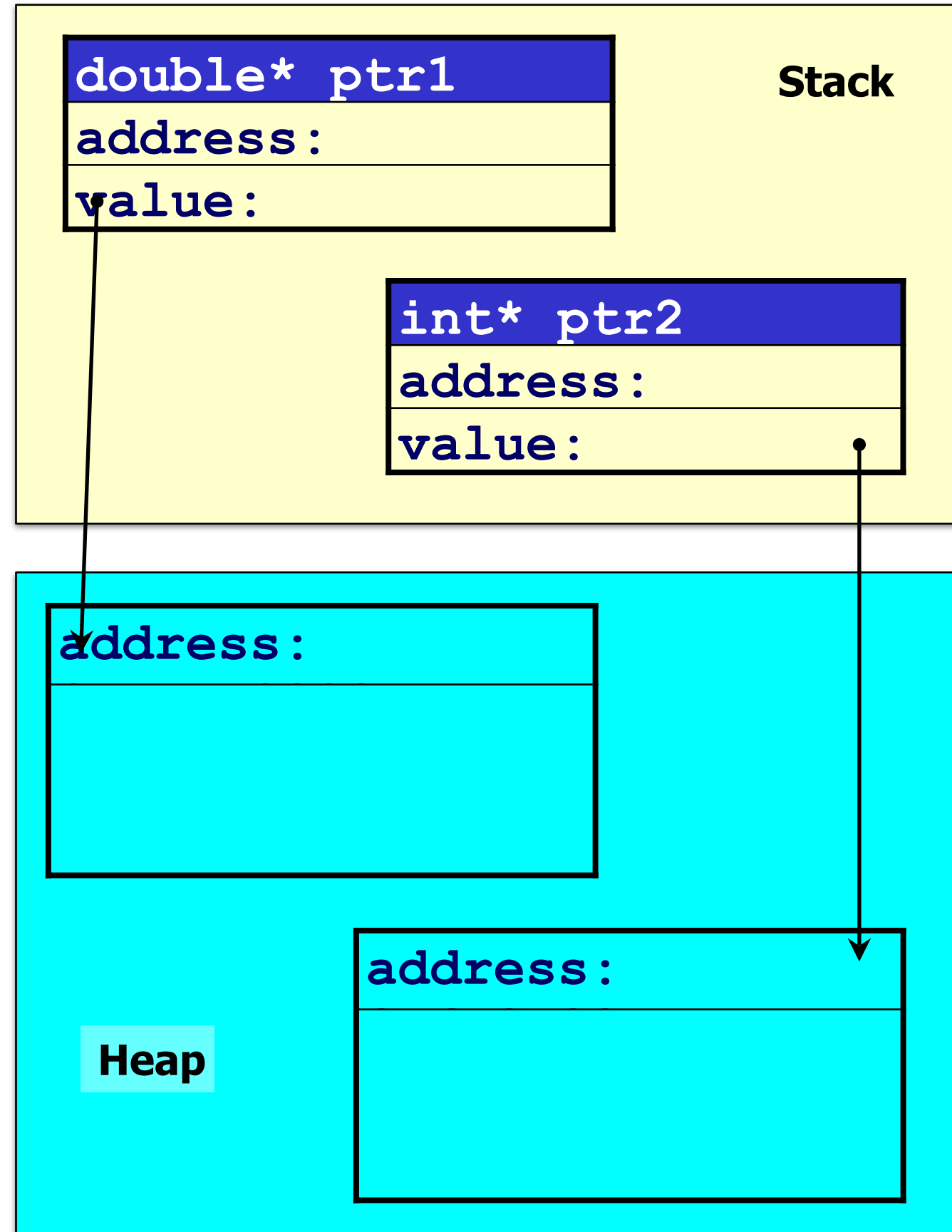
    cout << "ptr1[0]: " << ptr1[0]
          << endl;

    int* ptr2 = new int[1000];
    ptr2[233] = -13423;

    cout << "&ptr1: " << &ptr1
          << " sizeof(ptr1): " << sizeof(ptr1)
          << " ptr1: " << ptr1 << endl;

    cout << "&ptr2: " << &ptr2
          << " sizeof(ptr2): " << sizeof(ptr2)
          << " ptr2: " << ptr2 << endl;
    delete[] ptr1;
    delete[] ptr2;
    return 0;
}
```

```
$ g++ -Wall -o app7 app7.cpp
$ ./app7
ptr1[0]: 1.1
&ptr1: 0x22cce4  sizeof(ptr1): 4  ptr1: 0x7fee0008
&ptr2: 0x22cce0  sizeof(ptr2): 4
ptr2: 0x6a0700
```

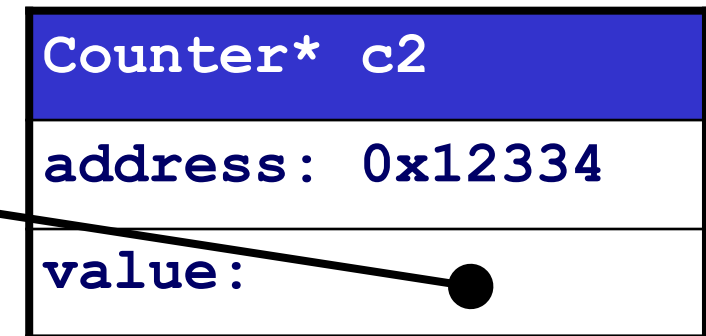
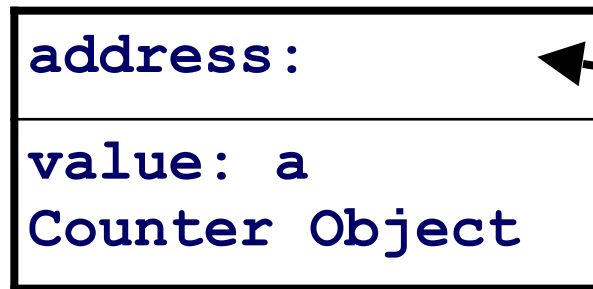


What does **new** do?

Dynamic object
in the heap

```
Counter* c2 = new Counter("c2");  
  
delete c2; // de-allocate memory!
```

Automatic variable
in the stack



- `new` allocates an amount of memory given by `sizeof(Counter)` somewhere in memory
- returns a pointer to this location
- we assign `c2` to be this pointer and access the dynamically allocated memory
- `delete` de-allocates the region of memory pointed to by `c2` and makes this memory available to be re-used by the program

Memory Leak: Killing the System

- Perhaps one of the most common problems in C++ programming
- User allocates memory at run time with `new` but never releases the memory – forgets to call `delete`!
- Golden rule: every time you call `new` ask yourself “where and when `delete` is called to free this memory” ?
- Even small amount of leak can lead to a crash of the system
 - Leaking 10 kB in a loop over 1M events leads to 1 GB of allocated and unusable memory!

Simple Example of Memory Leak

```
// app6.cpp
#include <iostream>
using namespace std;

int main() {

    for(int i=0; i<10000; ++i){

        double* ptr = new double[100000];
        ptr[0] = 1.1;

        cout << "i: " << i
              << ", ptr: " << ptr
              << ", ptr[0]: " << ptr[0]
              << endl;

        // delete[] ptr; // ops! memory
leak!
    }
    return 0;
}
```

- At each iteration ptr is a pointer to a new (and large) array of 100k doubles!
- This memory is not freed because we forgot the delete operator!
- At each turn more memory becomes unavailable until the system runs out of memory and crashes!

```
$ g++ -o leak1 leak1.cpp
$ ./leak1
i: 0, ptr: 0x4a0280, ptr[0]: 1.1
i: 1, ptr: 0x563bf8, ptr[0]: 1.1
...
i: 1381, ptr: 0x4247e178, ptr[0]: 1.1
i: 1382, ptr: 0x42541680, ptr[0]: 1.1
Abort (core dumped)
```

Advantages of Dynamic Memory Allocation

- No need to fix size of data to be used at compilation time
 - Easier to deal with real life use cases with variable and unknown number of data objects
 - No need to reserve very large but FIXED-SIZE arrays of memory
 - Example: interaction of particle in matter
 - How many particles are produced due to particle going through a detector?
 - Number not fixed a priori
 - Use dynamic allocation to create new particles as they are generated
- Disadvantage: correct memory management
 - Must keep track of **ownership** of **objects**
 - If not de-allocated can cause memory leaks which leads to slow execution and crashes
 - Most difficult part specially at the beginning or in complex systems

Destructor Method of a Class

- Constructor used by compiler to initialise instance of a class (an object)
 - Assign proper values to data members and allocate the object in memory
- Destructors are Special member function doing reverse work of constructors
 - Do cleanup when object goes out of scope
- Destructor performs termination house keeping when objects go out of scope
 - No de-allocation of memory
 - Tells the program that memory previously occupied by the object is again free and can be re-used
- Destructors are ***FUNDAMENTAL*** when using dynamic memory allocation

Special Features of Destructors

- Destructors have no arguments
- Destructors do not have a return type
 - Similar to constructors
- Destructor of class Counter MUST be called **~Counter()**

```
#ifndef Counter_h_
#define Counter_h_
// Counter.h
#include <string>

class Counter {
public:
    Counter(const std::string& name);
    ~Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);
    void print();

private:
    int count_;
    std::string name_;
};
#endif
```

Trivial Example of Destructor

Constructor initializes data members

```
#ifndef Counter_h_
#define Counter_h_
// Counter.h
#include <string>

class Counter {
public:
    Counter(const std::string& name);
    ~Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);
    void print();

private:
    int count_;
    std::string name_;
};
#endif
```

Destructor does nothing

```
#include "Counter.h"
#include <iostream> // needed for input/output
using std::cout;
using std::endl;

Counter::Counter(const std::string& name) {
    count_ = 0;
    name_ = name;
    cout << "Counter::Counter() called for Counter "
         << name_ << endl;
};

Counter::~~Counter() {
    cout << "Counter::~~Counter() called for Counter "
         << name_ << endl;
};

int Counter::value() {
    return count_;
}

void Counter::reset() {
    count_ = 0;
}

void Counter::increment() {
    count_++;
}

void Counter::increment(int step) {
    count_ = count_ + step;
}

void Counter::print() {
    cout << "Counter::print(): name: " << name_
         << " value: " << count_ << endl;
}
```

Who and When Calls the Destructor?

Constructors are called by compiler when new objects are created

```
// app1.cpp
#include "Counter.h"
#include <string>

int main() {

    Counter c1( std::string("c1") );
    Counter c2( std::string("c2") );
    Counter c3( std::string("c3") );

    c2.increment(135);
    c1.increment(5677);

    c1.print();
    c2.print();
    c3.print();

    return 0;
}
```

Destructors are called implicitly by compiler when objects go out of scope!

Destructors are called in reverse order of creation

Create in order objects c1, c2, and c3

```
$ g++ -c Counter.cc
$ g++ -o app1 app1.cpp Counter.o
$ ./app1
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::print(): name: c1 value: 5677
Counter::print(): name: c2 value: 135
Counter::print(): name: c3 value: 0
Counter::~~Counter() called for Counter c3
Counter::~~Counter() called for Counter c2
Counter::~~Counter() called for Counter c1
```

Destruct c3, c2, and c1

Another Example of Destructors

```
// app2.cpp
#include "Counter.h"
#include <string>

int main() {

    Counter c1( std::string("c1") );

    int count = 344;

    if( 1.1 <= 2.02 ) {
        Counter c2( std::string("c2") );

        Counter c3( std::string("c3") );
        if( count == 344 ) {
            Counter c4( std::string("c4") );
        }

        Counter c5( std::string("c5") );

        for(int i=0; i<3; ++i) {
            Counter c6( std::string("c6") );
        }
    }

    return 0;
}
```

```
$ g++ -o app2 app2.cpp Counter.o
$ ./app2
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
Counter::Counter() called for Counter c4
Counter::~~Counter() called for Counter c4
Counter::Counter() called for Counter c5
Counter::Counter() called for Counter c6
Counter::~~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~~Counter() called for Counter c6
Counter::Counter() called for Counter c6
Counter::~~Counter() called for Counter c6
Counter::~~Counter() called for Counter c5
Counter::~~Counter() called for Counter c3
Counter::~~Counter() called for Counter c2
Counter::~~Counter() called for Counter c1
```


Using **new** and **delete** Operators

```
// app6.cpp
#include "Counter.h"
#include "Datum.h"
#include <iostream>
using namespace std;

int main() {

    Counter c1("c1");

    Counter* c2 = new Counter("c2");
    c2->increment(6);

    Counter* c3 = new Counter("c3");

    Datum d1(-0.3,0.07);

    Datum* d2 = new Datum( d1 );
    d2->print();

    delete c2; // de-allocate memory!
    delete c3; // de-allocate memory!
    delete d2;

    return 0;
}
```

```
$ g++ -o app6 app6.cpp Datum.o Counter.o
$ ./app6
Counter::Counter() called for Counter c1
Counter::Counter() called for Counter c2
Counter::Counter() called for Counter c3
datum: -0.3 +/- 0.07
Counter::~~Counter() called for Counter c2
Counter::~~Counter() called for Counter c3
Counter::~~Counter() called for Counter c1
```

Order of calls to destructors has changed!

delete calls explicitly the destructor of the object to de-allocate memory

Vital for objects holding pointers to dynamically allocated memory

Why no message when destructing d2 ?

constant Member Functions

- Enforce principle of least privilege
 - Give privilege ONLY if needed
- `const` member functions cannot
 - modify data members
 - cannot be called on non-constant objects
- `const` member functions tell user, the function only 'uses' the input data or data members but makes **no changes to data members**
- Pay attention which function can be called on which objects
 - Objects can be constant
 - You can not modify a constant object
 - calling non-constant methods on constant objects does not make sense!

Datum Class and const Member Functions

```
class Datum {  
    public:  
        Datum();  
        Datum(double x, double y);  
        Datum(const Datum& datum);  
  
        double value() { return value_; }  
        double error() { return error_; }  
        double significance();  
        void print();  
  
        void setValue(double x) { value_ = x; }  
        void setError(double x) { error_ = x; }  
  
    private:  
        double value_;  
        double error_;  
};
```

Which methods
could become constant?

Datum Class with const Methods

All methods that only return values and do not change the attributes of an object!

All getters can be constant

```
#ifndef Datum1_h
#define Datum1_h
// Datum1.h
#include <iostream>
using namespace std;

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

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

    void setValue(double x) { value_ = x; }
    void setError(double x) { error_ = x; }

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

what about setter methods?

```
#include "Datum1.h"
#include <iostream>

Datum::Datum() {
    value_ = 0.; error_ = 0.;
}

Datum::Datum(double x, double y) {
    value_ = x; error_ = y;
}

Datum::Datum(const Datum& datum) {
    value_ = datum.value_;
    error_ = datum.error_;
}

double
Datum::significance() const {
    return value_/error_;
}

void Datum::print() const {
    using namespace std;
    cout << "datum: " << value_
         << " +/- " << error_ << endl;
}
```

Typical error with constant methods

```
#ifndef Datum2_h
#define Datum2_h
// Datum2.h
#include <iostream>
using namespace std;

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

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

    void setValue(double x) const { value_ = x; }
    void setError(double x) const { error_ = x; }

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

setters can never be constant!

Setter method is used to modify data members

Similarly constructors and destructors can not be constant

```
$ g++ -c Datum2.cc
```

```
In file included from Datum2.cc:1:
```

```
Datum2.h: In member function `void Datum::setValue(double) const':
```

```
Datum2.h:18: error: assignment of data-member `Datum::value_' in read-only structure
```

```
Datum2.h: In member function `void Datum::setError(double) const':
```

```
Datum2.h:19: error: assignment of data-member `Datum::error_' in read-only structure
```

Example of Error using non-constant functions

```
#ifndef Datum4_h
#define Datum4_h
// Datum4.h
#include <iostream>
#include <string>
using namespace std;
```

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

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

    void print(const std::string& comment) ;
    void setValue(double x) { value_ = x; }
    void setError(double x) { error_ = x; }
private:
    double value_;
    double error_;
};
#endif
```

print MUST have been constant!

bad design of the class!

```
void Datum::print(const std::string& comment) {
    using namespace std;
    cout << comment << ": " << value_
         << " +/- " << error_ << endl;
}
```

```
// app1.cpp
```

```
#include "Datum4.h"
```

```
int main() {
```

```
    Datum d1(-67.03, 32.12 );
```

```
    const Datum d2(-67.03, 32.12 );
```

```
    d1.print("datum");
```

```
    d2.print("const datum");
```

```
    return 0;
```

```
}
```

```
$ g++ -o app1 app1.cpp Datum4.o
app1.cpp: In function `int main()':
app1.cpp:12: error: passing `const Datum' as `this'
argument of `void Datum::print(const std::string&)'
discards qualifiers
```

Default Values for Methods

- Functions (not only member functions in classes) might be often invoked with recurrent values for their arguments
- It is possible to provide default values for arguments of any function in C++
 - Default arguments must be provided the first time the name of the function occurs
 - In declaration if separate implementation
 - In definition if the function is declared and defined at the same time
- Only the right-most argument can be omitted
 - Including all arguments to the right of omitted argument

Example of Default Values

```
// Counter.h

class Counter {
public:
    Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);

private:
    int count_;
};
```

Two increment() methods
but very similar functionality

increment() is a special case of
increment(int step) with step=1

Why two different methods?

```
// Counter.cc
// include class header files
#include "Counter.h"

// include any additional header files
// needed in the class
// definition
#include <iostream>
using std::cout;
using std::endl;

Counter::Counter() {
    count_ = 0;
};

int Counter::value() {
    return count_;
}

void Counter::reset() {
    count_ = 0;
}

void Counter::increment() {
    count_++;
}

void Counter::increment(int step) {
    count_ = count_ + step;
}
```


Default Value for Counter::increment(int step)

```
#ifndef Counter_Old_h_
#define Counter_Old_h_
// CounterOld.h
```

```
class Counter {
public:
    Counter();
    int value();
    void reset();
    void increment(int step = 1);
private:
    int count_;
};
#endif
```

Bad Practice!
Name of class
different from name
of file

```
// CounterOld.cc
#include "CounterOld.h"
#include <iostream>
using std::cout;
using std::endl;

Counter::Counter() {
    count_ = 0;
};

int Counter::value() {
    return count_;
}

void Counter::reset() {
    count_ = 0;
}

void Counter::increment(int step) {
    count_ = count_ + step;
}
```

```
// app3.cpp
#include "CounterOld.h" // old counter class
#include <iostream>
using namespace std;

int main() {

    Counter c1;

    c1.increment(); // no argument
    cout << "counter: " << c1.value() << endl;

    c1.increment(14); // provide argument, same function
    cout << "counter: " << c1.value() << endl;

    return 0;
}
```

```
$ g++ -c CounterOld.cc
$ g++ -o app3 app3.cpp
CounterOld.o
.$ ./app3
counter: 1
counter: 15
```

Ambiguous Use of Default Arguments

```
#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);
    double value() { return value_; }
    double error() { return error_; }
    double significance();

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

Does it make sense to have default value and error?

```
$ g++ -c Datum.cc
$ g++ -o app4 app4.cpp Datum.o
$ ./app4
datum: -0.23 +/- 0.05
datum: 5.23 +/- 0
datum: 1 +/- 0
```

```
#include "Datum.h"

Datum::Datum(double x, double y) {
    value_ = x;
    error_ = y;
}

Datum::Datum(const Datum& datum) {
    value_ = datum.value_;
    error_ = datum.error_;
}

double
Datum::significance() {
    return value_/error_;
}
```

```
#include "Datum.h"
int main() {

    Datum d1(-0.23, 0.05); // provide arguments
    d1.print();

    Datum d2(5.23); // default error ...
    d2.print();

    Datum d3; // default value and error!
    d3.print();
    return 0;
}
```

Don't Abuse Default Arguments!

- Default values must be used for functions very similar in functionality and with obvious default values
- If default values are not intuitive for user think twice before using them!
- Quite often different constructors correspond to DIFFERENT ways to create an object
 - Default values could be misleading
- If arguments are physical quantities ask yourself: is the default value meaningful and useful for everyone?