Separating Interface and Implementation of Classes
Header and Source Files
Dynamic Memory Management
Class Destructors

Shahram Rahatlou

Computing Methods in Physics

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

Anno Accademico 2019/20

SAPIENZA
UNIVERSITÀ DI ROMA

Reminder about g++

- g++ by default looks for a main function in the file being compiled unless differently instructed

- The main function becomes the program to run when the compiler is finished linking the binary application
  - Compiling: translate user code in high level language into binary code that system can use
  - Linking: put together binary pieces corresponding to methods used in the main function
  - Application: product of the linking process

- Source files of classes do not have any main method

- We need to tell g++ (and other compilers) no linking is needed

Compiling without Linking

- g++ has a `-c` option that allows to specify only compilation is needed

- User code is translated into binary but no attempt to look for main method and creating an application

```
$ ls -l Counter.*
-rw-r--r--  1 rahatlou users 449 May 15 00:55 Counter.cc
-rw-r--r--  1 rahatlou users 349 May 15 00:55 Counter.h

$ g++ -c Counter.cc

$ ls -l Counter.*
-rw-r--r--  1 rahatlou users  449 May 15 00:55 Counter.cc
-rw-r--r--  1 rahatlou users  349 May 15 00:55 Counter.h
-rw-r--r--  1 rahatlou users 1884 May 15 01:23 Counter.o
```

By default g++ creates a .o (object file) for the .cc file

Using Header Files in Applications

```
// app2.cpp
#include <iostream>
using namespace std;

#include "Counter.h"


Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: "
       << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: "
       << counter->value() << endl;
}
```

```
int main() {
  Counter counter;
  counter.increment(7);

  Counter* ptr = &counter;

  cout << "counter.value(): "
       <<counter.value() <<
endl;
  cout << "ptr = &counter: "
       << &counter << endl;
  cout << "ptr->value(): "
       << ptr->value() << endl;


  Counter c2 = makeCounter();
  c2.increment();

  printCounter( c2 );

  return 0;
}
```

```
$ g++ -o app2 app2.cpp
/tmp/ccJuugJc.o:app2.cpp:(.text+0x10d): undefined reference to `Counter::Counter()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x124): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x16e): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x1dc): undefined reference to `Counter::Counter()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x1ef): undefined reference to `Counter::increment(int)'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x200): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x272): undefined reference to `Counter::value()'
/tmp/ccJuugJc.o:app2.cpp:(.text+0x2b7): undefined reference to `Counter::increment()'
collect2: ld returned 1 exit status
```

Providing compiled Class Code at Link Time

- Including the header file is not sufficient!
  - It tells the compiler only about arguments and return type
  - But it does not tell him what to execute
  - Compiler doesn't have the binary code to use to create the application!

- We must use the compiled object file at link time
  - g++ is told to make an application called app2 from source code in app2.cpp and using also the binary file Counter.o to find any symbol needed in app2.cpp

```
$ g++ -o app2 app2.cpp Counter.o
$ ./app2
counter.value(): 7
ptr = &counter: 0x23ef10
ptr->value(): 7
counter value: 1
```

Problem: Multiple Inclusion of Header Files!

- What if we include the same header file several times?
  - This can happen in many ways

- Some pretty common ways are
  - **`App.cpp`** includes both **`Foo.h`** and **`Bar.h`**
  - **`Foo.h`** is included in **`Bar.h`** and **`Bar.cc`**

```
// Bar.h

#include "Foo.h"

class Bar {

  // class goes here
  Bar(const Foo& afoo, double x);


}`
```

```
// App.cpp

#include "Foo.h"
#include "Bar.h"

int main() {

  // program goes here
  Foo f1;
  Bar b1(f1, 0.3);

  return 0;
}
```

Example of Multiple Inclusion

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

Counter makeCounter() {
  Counter c;
  return c;
}

void printCounter(Counter& counter)  {
  cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter)  {
  cout << "counter value: " << counter->value() << endl;
}

#include "Counter.h"
int main() {
  Counter counter;
  counter.increment(7);

  Counter c2 = makeCounter();
  c2.increment();

  printCounter( counter );
  printCounter( c2 );

  return 0;
}
```

Line 19

```
// Counter.h
// Counter Class: simple counter class. Allows simp
// increments and also a reset function

// include header files for types and classes
// used in the declaration

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

  private:
    int count_;
}
```

Line 8

```
$ g++ -o app3 app3.cpp Counter.o
In file included from app3.cpp:19:
Counter.h:8: error: redefinition of `class Counter'
Counter.h:8: error: previous definition of `class Counter'
```

#define, #ifndef and #endif directives

- Problem of multiple inclusion can be solved at pre-compiler level

1: if Datum_h is not defined follow the instruction until #endif

2: define a new variable called Datum_h

3: end of ifndef block

```
#ifndef Datum_h
#define Datum_h
// Datum.h

class Datum {
  public:
    Datum();
    Datum(double x, double y);
    Datum(const Datum& datum);
    double value() { return value_; }
    double error() { return error_; }

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

Example: application using Datum

```
// app4.cpp
#include "Datum.h"
#include <iostream>

void print(Datum& input) {
  using namespace std;
  cout << "input: " << input.value()
       << " +/- " << input.error()
       << endl;
}

#include "Datum.h"

int main() {
  Datum d1(-1.4,0.3);
  print(d1);

  return 0;
}
```

```
$ g++ -c Datum.cc
$ g++ -o app4 app4.cpp Datum.o
$ ./app4
input: -1.4 +/- 0.3
```

Typical Errors

- Forget to use the scope operator :: in .cc files

```
#ifndef FooDatum_h
#define FooDatum_h
// FooDatum.h

class FooDatum {
  public:
    FooDatum();
    FooDatum(double x, double y);
    FooDatum(const FooDatum& datum);
    double value() { return value_; }
    double error() { return error_; }
    double significance();

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

```
#include "FooDatum.h"

FooDatum::FooDatum() { }

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

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

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

```
$ g++ -c FooDatum.cc
FooDatum.cc: In function `double significance()':
FooDatum.cc:17: error: `value_' undeclared (first use this function)
FooDatum.cc:17: error: (Each undeclared identifier is reported only once
 for each function it appears in.)
FooDatum.cc:17: error: `error_' undeclared (first use this function)
```

- Functions implemented as global
- error when applying function as a member function to objects
- No error compiling the classes but error when compiling the application

Reminder: Namespace of Classes

- C++ uses namespace as integral part of a class, function, data member

- Any quantity declared within a namespace can be accessed ONLY by using the scope operator :: and by specifying its namespace

- When using a new class, you must look into its header file to find out which namespace it belongs to
  - There are no shortcuts!

- When implementing a class you must specify its namespace
  - Unless you use the using directive

Another Example of Namespace

```
#ifndef CounterNS_h_
#define CounterNS_h_
#include <string>

namespace rome {
  namespace didattica {

    class Counter {
      public:
        Counter(const std::string& name);
        ~Counter();
        int value();
        void reset();
        void increment(int step =1);
        void print();

      private:
        int count_;
        std::string name_;
    }; // class counter

  } // namespace didattica
} //namespace rome
#endif
```

```
#include "CounterNS.h"

int main() {
  rome::didattica::Counter c1("c1");
  c1.print();
  return 0;
}
```

```
// CounterNS.cc
#include "CounterNS.h"

// include any additional heade files needed in the class
// definition
#include <iostream> // needed for input/output
using std::cout;
using std::endl;
using namespace rome::didattica;

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(int step) {
  count_ = count_+step;
}

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

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

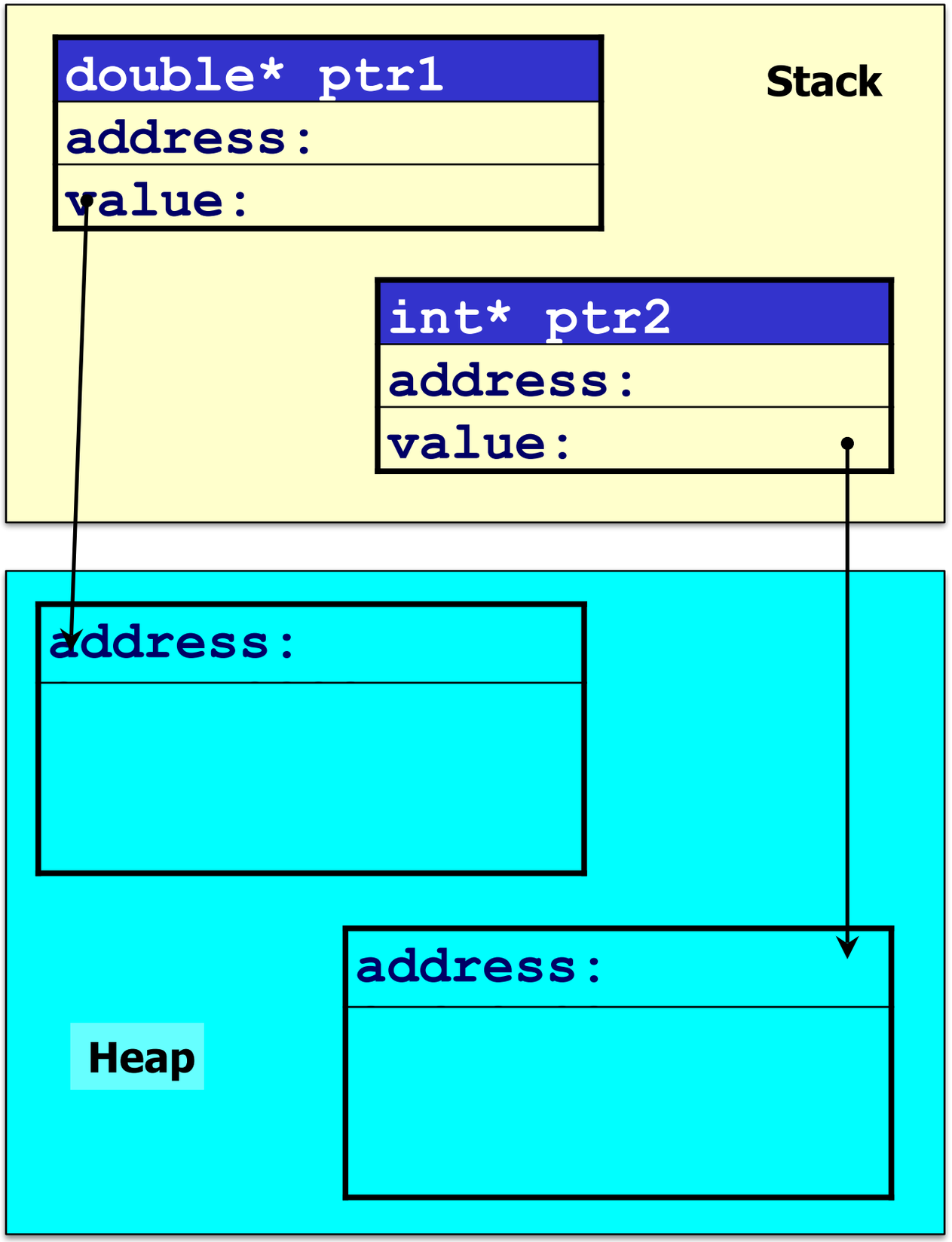
- Which memory is used by **`new`**?
  - **`new`** allocated objects in the free store also known as heap
  - This is region of memory assigned to each program at run time
  - Memory allocated by **`new`** is unavailable until we free it and give it back to system via **`delete`** operator

- **`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
```

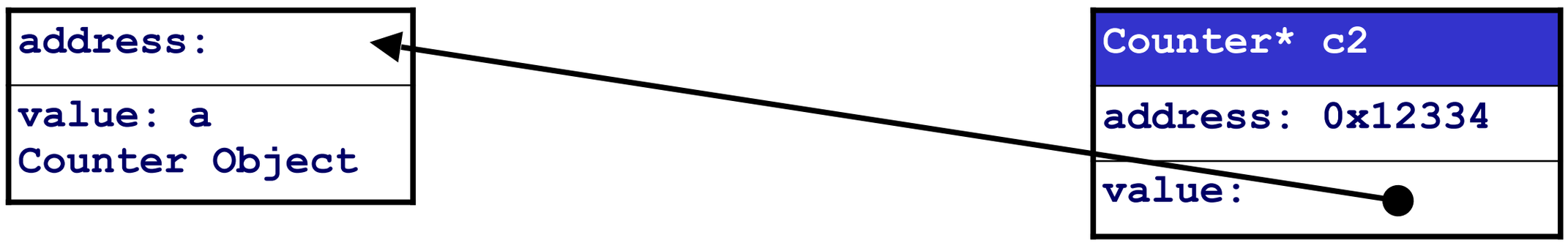


What does **new** do?

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
  - Do I really need to use new?
  - 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 un-usable 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;
}
```

```
$ 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)
```

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

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

```
$ 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
```

Create in order objects c1, c2, and c3

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
```