

Conception Objets Avancée

Introduction to C++

Giuseppe Lipari

CRISAL - Université de Lille

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

Pre-requisites

To understand this course, you should at least know the basic C syntax

- ▶ functions declaration and function call,
- ▶ global and local variables
- ▶ pointers (will do again during the course)
- ▶ structures

Summary

1. First part of the course: classes

- ▶ Classes, objects, memory layout
- ▶ Pointer and references
- ▶ Copying
- ▶ Inheritance, multiple inheritance
- ▶ Access rules; public, protected and private inheritance
- ▶ Exceptions
- ▶ Templates, the STL

2. Second part: C++ 14

- ▶ What does it change
- ▶ lambda functions
- ▶ auto
- ▶ move semantic
- ▶ new STL classes
- ▶ Safety to exceptions

Summary II

3. Third part: patterns

- ▶ Some patterns in C++
- ▶ Function objects
- ▶ Template patterns
- ▶ Meta-programming with templates

4. Fourth part: concurrency and distribution

- ▶ Thread library, synchronisation
- ▶ Futures and promises
- ▶ The Active Object pattern

Conventions

When explaining a concept:

- ▶ if nothing is said, the concept is valid for all standard C++ language versions (e.g. starting from C++98)
- ▶ if C++11 is specified, the concept is valid starting from C++11
 - ▶ when compiling with g++ or clang++, you must specify the option `-std=c++11`
- ▶ if C++14 is specified, the concept is valid only from C++14 (and not for C++11, or C++98)
 - ▶ when compiling with g++ or clang++, you must specify the option `-std=c++14`
- ▶ if C++17 is specified, the concept is only valid for the latest version of C++, chances are this is not yet available in your compiler!

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

What is C++ ?

C is not a high-level language.

– Brian Kernighan, creator of C with D. M. Ritchie

What is C++ ?

C is not a high-level language.

– Brian Kernighan, creator of C with D. M. Ritchie

Actually I made up the term 'object-oriented', and I can tell you I did not have C++ in mind.

– Alan Kay, creator of Smalltalk, father of the OO paradigm

What is C++ ?

C is not a high-level language.

– Brian Kernighan, creator of C with D. M. Ritchie

Actually I made up the term 'object-oriented', and I can tell you I did not have C++ in mind.

– Alan Kay, creator of Smalltalk, father of the OO paradigm

There are only two kinds of languages: the ones people complain about and the ones nobody uses.

– Bjarne Stroustrup, creator of C++

Abstraction

Those types are not abstract: they are as real as int and float

– Doug McIlroy

- ▶ An essential instrument for OO programming is the support for data abstraction
- ▶ C++ permits to define new types and their operations
- ▶ Creating a new data type means defining:
 - ▶ Which elements it is composed of (*internal structure*);
 - ▶ How it is built/destroyed (*constructor/destructor*);
 - ▶ How we can operate on this type (*methods/operations*).

Data abstraction in C

- ▶ We can do data abstraction in C (and in almost any language)

```
typedef struct __complex {  
    double real_  
    double imaginary_  
} cmplx;  
  
void add_to(cmplx *a, cmplx *b);  
void sub_from(cmplx *a, cmplx *b);  
double get_module(cmplx *a);
```

- ▶ We have to pass the main data to every function
- ▶ name clashing: if another abstract type defines a function `add_to()`, the names will conflict!
- ▶ No information hiding: any user can access the internal data using them improperly

C++ version

```
class Complex {
    double real_;
    double imaginary_;
public:
    Complex();
    Complex(double a, double b);
    ~Complex();

    double real() const;
    double imaginary() const;
    double module() const;
    Complex &operator=(const Complex &a);
    Complex &operator+=(const Complex &a);
    Complex &operator-=(const Complex &a));
};
```

How to use it

```
Complex c1;           // default constructor
Complex c2(1,2);      // constructor
Complex c3(3,4);      // constructor

cout << "c1=(" << c1.real() << ","
      << c1.imaginary() << ")" << endl;

c1 = c2;               // assignment
c3 += c1;              // operator +=
c1 = c2 + c3;          // ERROR: operator + not yet defined
```

Using the new data type

- ▶ The new data type is used just like a predefined data type
 - ▶ it is possible to define new functions for that type:
 - ▶ `real()`, `imaginary()` and `module()`
 - ▶ it is possible to define new operators
 - ▶ `=` `+=` `-=`
- ▶ C++ is a strongly typed language
 - ▶ the compiler knows which function to invoke by looking at the type

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

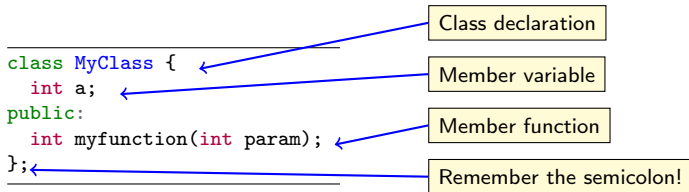
Copy Constructor

Static

Constants

Class

- ▶ Class is the main construct for building new types in C++
 - ▶ A class is almost equivalent to a struct with functions inside
 - ▶ In the C-style programming, the programmer defines structs, and global functions to act on the structs
 - ▶ In C++-style programming, the programmer defines classes with embedded functions



Object construction

- ▶ An **object** is an instance of a class
- ▶ An object is created by calling a special function called *constructor*
 - ▶ A constructor is a function that has the same name of the class and no return value
 - ▶ It may or may not have parameters;
 - ▶ It is invoked in a special way

```
class MyClass {  
public:  
    MyClass() {  
        cout << "Constructor"<<endl;  
    }  
};
```

Declaration of the constructor

```
MyClass obj;
```

Object construction

- ▶ An **object** is an instance of a class
- ▶ An object is created by calling a special function called *constructor*
 - ▶ A constructor is a function that has the same name of the class and no return value
 - ▶ It may or may not have parameters;
 - ▶ It is invoked in a special way

```
class MyClass {  
public:  
    MyClass() {  
        cout << "Constructor"<<endl;  
    }  
};
```

Declaration of the constructor

```
MyClass obj;
```

Invoke the constructor to create an object

Constructor - II

A class can have many constructors

```
class MyClass {  
    int a;  
    int b;  
public:  
    MyClass(int x);  
    MyClass(int x, int y);  
};
```

```
MyClass obj;  
MyClass obj1(2);  
MyClass obj2(2,3);
```

```
int myvar(2);  
double pi(3.14);
```

This is an error, no constructor without parameters

Calls the first constructor

Calls the second constructor

Same syntax is valid for primitive data types

Default constructor

- ▶ Rules for constructors
 - ▶ If you do not specify a constructor, a default one with no parameters is provided by the compiler
 - ▶ If you provide a constructor (any constructor) the compiler will not provide a default one for you
- ▶ Constructors are used to initialise members

```
class MyClass {  
    int a;  
    int b;  
public:  
    MyClass(int x, int y) {  
        a = x; b = 2*y;  
    }  
};
```

Initialization list

- ▶ Members can be initialised through a special syntax
 - ▶ This syntax is preferable (the compiler can catch some obvious mistake)
 - ▶ use it whenever you can (i.e. almost always)

```
class MyClass {  
    int a;  
    int b;  
public:  
    MyClass(int x, int y) :  
        a(x), b(y)  
    {  
        // other initialisation  
    }  
};
```

A comma separated list of constructors, following the :

Accessing members

- Members of one object can be accessed using the classical **dot** notation, similarly to structs in C and objects in Java

```
class MyClass {  
public:  
    int a;  
    int f();  
    void g(int i, int ii);  
};
```

```
MyClass x;  
MyClass y;
```

```
x.a = 5;  
y.a = 7;  
x.f();  
y.g(5, 10);
```

Assigning to a member variable of object x

Accessing members

- Members of one object can be accessed using the classical **dot** notation, similarly to structs in C and objects in Java

```
class MyClass {  
public:  
    int a;  
    int f();  
    void g(int i, int ii);  
};
```

```
MyClass x;  
MyClass y;
```

```
x.a = 5;  
y.a = 7;  
x.f();  
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Accessing members

- Members of one object can be accessed using the classical **dot** notation, similarly to structs in C and objects in Java

```
class MyClass {  
public:  
    int a;  
    int f();  
    void g(int i, int ii);  
};
```

```
MyClass x;  
MyClass y;
```

```
x.a = 5;  
y.a = 7;  
x.f();  
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Calling member function f() of object x

Accessing members

- Members of one object can be accessed using the classical **dot** notation, similarly to structs in C and objects in Java

```
class MyClass {  
public:  
    int a;  
    int f();  
    void g(int i, int ii);  
};
```

```
MyClass x;  
MyClass y;
```

```
x.a = 5;  
y.a = 7;  
x.f();  
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Calling member function f() of object x

Calling member function g() of object y

Implementing member functions

- ▶ You can implement a member function (including constructors) in a separate .cpp file

complex.hpp

```
class Complex {  
    double real_;  
    double img_;  
public:  
    ...  
    double module() const;  
    ...  
};
```

complex.cpp

```
double Complex::module()  
{  
    double temp;  
    temp = real_ * real_ +  
           img_ * img_;  
    return temp;  
}
```

- ▶ This is preferable most of the times
- ▶ put implementation in include files only if you hope to use *in-lining* optimisation

Accessing internal members

```
double Complex::module() const
{
    double temp;
    temp = real_ * real_ + img_ * img_;
    return temp;
}
```

- ▶ The `::` operator is called **scope resolution** operator
- ▶ member variables and functions can be accessed without *dot* or *arrow*

Accessing internal members

```
double Complex::module() const  
{  
    double temp;  
    temp = real_ * real_ + img_ * img_;  
    return temp;  
}
```

scope resolution

access to internal variable

- ▶ The `::` operator is called **scope resolution** operator
- ▶ member variables and functions can be accessed without *dot* or *arrow*

Uniform Initialization (C++11)

- ▶ Starting from C++11, it is possible to initialise an object with 3 different syntaxes:

```
int a = 0;
```

```
int b(0);
```

```
int c{0};
```

- ▶ In this case, the three are equivalent
- ▶ The last one is called **uniform initialization** because it is the most general
 - ▶ although it cannot be used everywhere, as we will see...

Uniform initialization (C++11)

- ▶ The reason for introducing uniform initialization is more apparent for object constructors


```
Widget w1(7);
```

```
Widget w2;
```

```
Widget w3();
```

```
Widget w4{7};
```

```
Widget w5{};
```



Calls the constructor taking one int argument

Uniform initialization (C++11)

- ▶ The reason for introducing uniform initialization is more apparent for object constructors

```
Widget w1(7);
```

```
Widget w2;
```

```
Widget w3();
```

```
Widget w4{7};
```

```
Widget w5{};
```

← Calls the constructor taking one int argument

← Calls the default constructor

Uniform initialization (C++11)

- ▶ The reason for introducing uniform initialization is more apparent for object constructors

```
Widget w1(7);
```

← Calls the constructor taking one int argument

```
Widget w2;
```

← Calls the default constructor

```
Widget w3();
```

```
Widget w4{7};
```

← Declares a function!

```
Widget w5{};
```

Uniform initialization (C++11)

- ▶ The reason for introducing uniform initialization is more apparent for object constructors

```
Widget w1(7);
```

Calls the constructor taking one int argument

```
Widget w2;
```

Calls the default constructor

```
Widget w3();
```

Declares a function!

```
Widget w4{7};
```

```
Widget w5{};
```

Calls the constructor taking one int argument

Uniform initialization (C++11)

- ▶ The reason for introducing uniform initialization is more apparent for object constructors

```
Widget w1(7);
```

Calls the constructor taking one int argument

```
Widget w2;
```

Calls the default constructor

```
Widget w3();
```

Declares a function!

```
Widget w4{7};
```

Calls the constructor taking one int argument

```
Widget w5{};
```

Calls the default constructor

Class member initialization (C++11)

- ▶ Since C++11 it is possible to initialize member variables directly in the declaration with default values

```
class A {  
    int a=0;  
    int b{0};  
    Widget obj{"ObjName"};  
public:  
    A() {}  
    A(int x) : a(x), b(x) {}  
};  
  
A obj1;  
  
A obj2{3};
```

- ▶ Notice the use of the uniform initialization syntax
- ▶ the value of `obj1.a` is equal to 0 because the object is initialised by the default constructor
- ▶ the value of `obj2.a` is equal to 3 if the object is initialised by the second constructor
- ▶ **NB:** You cannot use parenthesis in default initialisation of member variables

Access control keywords

- ▶ A member can be:
 - ▶ **private**: only member functions of the same class can access it; other classes or global functions can't
 - ▶ **protected**: only member functions of the same class or of derived classes can access it: other classes or global functions can't
 - ▶ **public**: every function can access it

```
class MyClass {  
private:  
    int a;  
public:  
    int c;  
};
```

```
MyClass data;  
  
cout << data.a;    // ERROR!  
cout << data.c;    // OK
```

Access control and scope

```
int xx;
```

global variable

```
class A {
```

```
    int xx;
```

```
public:
```

```
    void f();
```

```
};
```

member variable

```
void A::f()
```

```
{
```

```
    xx = 5;
```

```
    ::xx = 3;
```

```
    xx = ::xx + 2;
```

```
}
```

access member xx

access global xx

Private

- ▶ Some people think that private is synonym of secret
 - ▶ they complain that the private part is visible in the header file
- ▶ **private** means not accessible from other classes and does not mean secret
- ▶ The compiler needs to know the size of the object, in order to allocate memory to it
- ▶ If you want to hide completely the implementation, you must follow the **pimpl** idiom (more on this later on)

Friends

```
class A {  
    friend class B;  
    int y;  
    void f();  
public:  
    int g();  
};
```

B is friend of A ...

```
class B {  
    int x;  
public:  
    void f(A &a);  
};
```

... hence B can access private members of A

```
void B::f(A &a)  
{  
    x = a.y;  
    a.f();  
}
```

Friend operators

- ▶ Global functions and operators can be friend of a class
- ▶ Also, a single member function can be declared friend

```
class A {  
    friend B::f();  
    friend h();  
    int y;  
    void f();  
public:  
    int g();  
};
```

friend member function

friend global function

- ▶ It is better to use the *friend* keyword only when it is really necessary because it breaks the access rules .

"Friends, much as in real life, are often more trouble than they're worth." – Scott Meyers

Example of friend global function

```
class A;
```

Forward class declaration

```
void print(A obj);
```

Prototype of global function

```
class A {  
    int data;  
    friend void print(A obj);  
public:  
    A() : data(0) {}  
};  
void print(A obj) {  
    cout << obj.data << endl;  
}
```

Friend declaration

Using class private variable

Nested classes

- ▶ It is possible to declare a class inside another class
- ▶ Access control keywords apply

```
class A {  
    class B {  
        int a;  
    public:  
        int b;  
    }  
    B obj;  
public:  
    void f();  
};
```

- ▶ Class B is private to class A: it is not part of the interface of A, but only of its implementation.
- ▶ However, A is not allowed to access the private part of B!!
 - ▶ `A::f()` cannot access `B::a`.
- ▶ To accomplish this, we have to declare A as friend of B, or leave `B::a` public

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

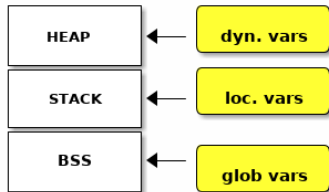
Constants

Memory layout

- ▶ Let us recapitulate the rules for the lifetime and visibility of variables
 - ▶ **Global variables** are defined outside of any function. Their lifetime is the duration of the program: they are created when the program is loaded in memory, and deleted when the program exits
 - ▶ **Local variables** are defined inside functions or inside code blocks (delimited by curly braces { and }). Their lifetime is the execution of the block: they are created before the block starts executing, and destroyed when the block completes execution
- ▶ Global and local variables are in different **memory segments**, and are managed in different ways

Memory segments

- ▶ The main data segments of a program are shown below
- ▶ The BSS segment contains **global variables**. It is divided into two segments, one for initialised data (i.e. data that is initialised when declared), and non-initialised data.
 - ▶ The size of this segment is statically decided when the program is loaded in memory, and can never change during execution
- ▶ The STACK segment contains **local variables**
 - ▶ Its size is dynamic: it can grow or shrink, depending on how many local variables are in the current block
- ▶ The HEAP segment contains **dynamic memory** that is managed directly by the programmer



Example

```
int a = 5; // initialised global data
int b;     // non initialised global data

int f(int i)    // i, d and s[] are local variables
{              // will be created on the stack when
    double d;   // function f() is invoked
    char s[] = "Lipari";
    ...
}


int main()
{
    int s, z;    // local variables, are created on the stack
                 // when the program starts

    f();         // here f() is invoked, so the stack for f() is created
}
```

Pointers

- ▶ A pointer is a variable that can hold a memory address

```
int a = 5;  
int b = 7;  
int *p;  
  
p = &a;  
  
cout << p << endl;  
  
cout << *p << endl;  
  
*p = 6;  
  
p = &b;  
  
cout << *p << endl;
```



Declaration of a pointer to an integer variable

Pointers

- ▶ A pointer is a variable that can hold a memory address

```
int a = 5;
```

```
int b = 7;
```

```
int *p;
```

```
p = &a;
```

```
cout << p << endl;
```

```
cout << *p << endl;
```

```
*p = 6;
```

```
p = &b;
```

```
cout << *p << endl;
```

Declaration of a pointer to an integer variable

p takes the address of a

Pointers

- ▶ A pointer is a variable that can hold a memory address

```
int a = 5;
```

```
int b = 7;
```

```
int *p;
```

Declaration of a pointer to an integer variable

```
p = &a;
```

```
cout << p << endl;
```

p takes the address of a

```
cout << *p << endl;
```

print the address

```
*p = 6;
```

```
p = &b;
```

```
cout << *p << endl;
```

Pointers

- ▶ A pointer is a variable that can hold a memory address

```
int a = 5;  
int b = 7;  
int *p;
```

Declaration of a pointer to an integer variable

```
p = &a;
```

p takes the address of a

```
cout << p << endl;
```

print the address

```
cout << *p << endl;
```

prints the value in a

```
*p = 6;
```

```
p = &b;
```

```
cout << *p << endl;
```

Pointers

- A pointer is a variable that can hold a memory address

```
int a = 5;  
int b = 7;  
int *p;
```

Declaration of a pointer to an integer variable

```
p = &a;
```

p takes the address of a

```
cout << p << endl;
```

print the address

```
cout << *p << endl;
```

prints the value in a

```
*p = 6;
```

changes the value in a = 6

```
p = &b;
```

```
cout << *p << endl;
```

Pointers

- A pointer is a variable that can hold a memory address

```
int a = 5;  
int b = 7;  
int *p;
```

Declaration of a pointer to an integer variable

```
p = &a;
```

p takes the address of a

```
cout << p << endl;
```

print the address

```
cout << *p << endl;
```

prints the value in a

```
*p = 6;
```

changes the value in a = 6

```
p = &b;
```

```
cout << *p << endl;
```

now p points to b

Pointers

- ▶ A pointer is a variable that can hold a memory address

```
int a = 5;  
int b = 7;  
int *p;
```

Declaration of a pointer to an integer variable

```
p = &a;
```

p takes the address of a

```
cout << p << endl;
```

print the address

```
cout << *p << endl;
```

prints the value in a

```
*p = 6;
```

changes the value in a = 6

```
p = &b;
```

now p points to b

```
cout << *p << endl;
```

prints the value in b

Arrays

- ▶ The name of an array is equivalent to a constant pointer to the first element
- ▶ With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";
```

```
cout << *name << endl;
```

```
char *p = name;
```

```
p++;
```

```
assert(p == name+1);
```

```
while (*p != 0)
```

```
    cout << *(p++);
```

```
cout << endl;
```



prints "G"

Arrays

- ▶ The name of an array is equivalent to a constant pointer to the first element
- ▶ With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";
```

```
cout << *name << endl;
```

prints "G"

```
char *p = name;
```

```
p++;
```

declares a pointer to the first element of the array

```
assert(p == name+1);
```

```
while (*p != 0)
    cout << *(p++);
cout << endl;
```

Arrays

- ▶ The name of an array is equivalent to a constant pointer to the first element
- ▶ With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";
```

```
cout << *name << endl;
```

prints "G"

```
char *p = name;
```

declares a pointer to the first element of the array

```
p++;
```

Increments the pointer, now points to "I"

```
assert(p == name+1);
```

```
while (*p != 0)
```

```
    cout << *(p++);
```

```
cout << endl;
```

Arrays

- ▶ The name of an array is equivalent to a constant pointer to the first element
- ▶ With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";
```

prints "G"

```
cout << *name << endl;
```

declares a pointer to the first element of the array

```
char *p = name;
```

```
p++;
```

Increments the pointer, now points to "I"

```
assert(p == name+1);
```

```
while (*p != 0)
```

```
    cout << *(p++);
```

```
cout << endl;
```

this assertion is correct

Arrays

- ▶ The name of an array is equivalent to a constant pointer to the first element
- ▶ With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";
```

```
cout << *name << endl;
```

```
char *p = name;
```

```
p++;
```

```
assert(p == name+1);
```

```
while (*p != 0)  
    cout << *(p++);  
cout << endl;
```

prints "G"

declares a pointer to the first element of the array

Increments the pointer, now points to "I"

this assertion is correct

zero marks the end of the string

Arrays

- ▶ The name of an array is equivalent to a constant pointer to the first element
- ▶ With non-const pointers we can do pointer arithmetic

```
char name[] = "Giuseppe";
```

```
cout << *name << endl;
```

```
char *p = name;
```

```
p++;
```

```
assert(p == name+1);
```

```
while (*p != 0)
    cout << *(p++);
cout << endl;
```

prints "G"

declares a pointer to the first element of the array

Increments the pointer, now points to "I"

this assertion is correct

zero marks the end of the string

prints the content of address pointed by p, and increments it

Dynamic memory

- ▶ Dynamic memory is managed by the user
- ▶ In C:
 - ▶ to allocate memory, call function `malloc`
 - ▶ to deallocate, call `free`
 - ▶ Both take pointers to any type, so they are not type-safe
- ▶ In C++
 - ▶ to allocate memory, use operator `new`
 - ▶ to deallocate, use operator `delete`
 - ▶ they are more type-safe

The new operator

- ▶ The new and delete operators can be applied to primitive types, and classes
- ▶ operator new automatically calculates the size of memory to be allocated

Allocates an integer pointed by p

```
int *p = new int(5);
```

```
class A { ... };
```

```
A *q = new A();
```

```
delete p;
```

```
delete q;
```

The new operator

- ▶ The new and delete operators can be applied to primitive types, and classes
- ▶ operator new automatically calculates the size of memory to be allocated

```
int *p = new int(5);
```

Allocates an integer pointed by p

```
class A { ... };
```

```
A *q = new A();
```

Does two things:

- 1) Allocates memory for an object of class A
- 2) calls the constructor of A()

```
delete p;
```

```
delete q;
```

The new operator

- ▶ The new and delete operators can be applied to primitive types, and classes
- ▶ operator new automatically calculates the size of memory to be allocated

```
int *p = new int(5);
```

Allocates an integer pointed by p

```
class A { ... };
```

Does two things:

- 1) Allocates memory for an object of class A
- 2) calls the constructor of A()

```
A *q = new A();
```

Deallocates the memory pointed by p

```
delete p;
```

```
delete q;
```

The new operator

- ▶ The new and delete operators can be applied to primitive types, and classes
- ▶ operator new automatically calculates the size of memory to be allocated

```
int *p = new int(5);
```

Allocates an integer pointed by p

```
class A { ... };
```

Does two things:

- 1) Allocates memory for an object of class A
- 2) calls the constructor of A()

```
A *q = new A();
```

Deallocates the memory pointed by p

```
delete p;
```

```
delete q;
```

Does two things:

- 1) Calls the *destructor* for A
- 2) deallocates the memory pointed by q

Destructor

- ▶ The destructor is called just before the object is deallocated.
- ▶ It is always called both for all objects (allocated on the stack, in global memory, or dynamically)
- ▶ If the programmer does not define a constructor, the compiler automatically adds one by default (which does nothing)

```
class A {  
    ...  
public:  
    A() { ... } // constructor  
    ~A() { ... } // destructor  
};
```

The destructor never
takes any parameter

Why a destructor ?

- ▶ A destructor is useful when an object dynamically allocates memory, so that it can deallocate it when the object is deleted

```
class A { ... };  
  
class B {  
    A *p;  
public:  
    B() {  
        p = new A();  
    }  
    ~B() {  
        delete p;  
    }  
};
```

- ▶ p is initialised when the object is created
- ▶ The memory is deallocated when the object is deleted

Example with destructor

`examples/destructor.cpp`

New and delete for arrays

- ▶ To allocate an array, use this form

```
int *p = new int[5]; // allocates an array of 5 int
...
delete [] p;         // notice the delete syntax

A *q = new A[10];    // allocates an array of 10
...                  // objects of type A
delete [] q;
```

- ▶ In the second case, the default constructor is called to build the 10 objects
- ▶ Therefore, this can only be done is a default constructor (without arguments) is available

Null pointer

- ▶ The address 0 is an invalid address
 - ▶ (no data and no function can be located at 0)
- ▶ therefore, in C/C++ a pointer to 0 is said to be a *null pointer*, which means a pointer that points to nothing.
- ▶ Dereferencing a null pointer is always a bad error (null pointer exception, or segmentation fault)
- ▶ In C, the macro `NULL` is used to mark 0, or a pointer to 0
 - ▶ however, 0 can be seen to be of integer type, or a null pointer
- ▶ In C++11, the null pointer is indicated with the constant `nullptr`
 - ▶ this constant cannot be automatically converted to an integer

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

Pointers

- ▶ We can define a pointer to an object

```
class A { ... };
```

```
A myobj;
```

```
A *p = &myobj;  ←
```

Pointer p contains the address of myobj

- ▶ As in C, in C++ pointers can be used to pass arguments to

```
void fun(int a, int *p)
```

{

```
a = 5;
```

```
*p = 7;
```

}

• • •

```
int x = 0, y = 0;
```

```
fun(x, &y);
```

x is passed by value (i.e. it is copied into a), so this assignment only modifies the copy

Pointers

- We can define a pointer to an object

```
class A { ... };
```

```
A myobj;
```

```
A *p = &myobj;
```

Pointer p contains the address of myobj

- As in C, in C++ pointers can be used to pass arguments to

```
void fun(int a, int *p)
```

```
{
```

```
    a = 5;
```

```
    *p = 7;
```

```
}
```

```
...
```

```
int x = 0, y = 0;
```

```
fun(x, &y);
```

x is passed by value (i.e. it is copied into a), so this assignment only modifies the copy

y is passed by address (i.e. we pass its address, so that it can be modified inside the function)

Pointers

- We can define a pointer to an object

```
class A { ... };
```

```
A myobj;
```

```
A *p = &myobj;
```

Pointer p contains the address of myobj

- As in C, in C++ pointers can be used to pass arguments to

```
void fun(int a, int *p)
```

```
{
```

```
    a = 5;
```

```
    *p = 7;
```

```
}
```

```
...
```

```
int x = 0, y = 0;
```

```
fun(x, &y);
```

x is passed by value (i.e. it is copied into a), so this assignment only modifies the copy

y is passed by address (i.e. we pass its address, so that it can be modified inside the function)

After the function call, x=0, y=7

Another example

`examples/pointerarg.cpp`

What happened:

- ▶ function `g()` takes an object, and makes a copy
 - ▶ `c` is a copy of `obj`
 - ▶ `g()` has no side effects, as it works on the copy
- ▶ Function `h()` takes a pointer to the object
 - ▶ it works on the original object `obj`, changing its internal value

Pointer to function

- ▶ It is also possible to define pointers to functions:
 - ▶ The portion of memory where the code of a function resides has an address; we can define a pointer to this address

```
void (*funcPtr)();           // pointer to void f();
int (*anotherPtr)(int)      // pointer to int f(int a);

void f(){...}

funcPtr = &f();              // now funcPtr points to f()
funcPtr = f;                  // equivalent syntax

(*funcPtr)();                // call the function
```

Pointers to functions - II

- ▶ To simplify notation, it is possible to use typedef:

```
typedef void (*MYFUNC)();  
typedef void* (*PTHREADFUN)(void *);  
  
void f() { ... }  
void *mythread(void *) { ... }  
  
MYFUNC funcPtr = f;  
PTHREADFUN pt = mythread;
```

- ▶ Arrays of function pointers

- ▶ It is also possible to define arrays of function pointers:

```
void f1(int a) {}  
void f2(int a) {}  
void f3(int a) {}  
...  
void (*funcTable []) (int) = {f1, f2, f3};  
...  
for (int i =0; i<3; ++i) (*funcTable[i])(i + 5);
```

Field dereferencing

- ▶ When we have to use a member inside a function through a pointer

```
class Data {  
public:  
    int x;  
    int y;  
};  
  
Data aa;           // object  
Data *pa = &aa;    // pointer to object  
pa->x;              // select a field  
(*pa).y;           // "        "
```

Pointer to member

- ▶ It is possible to have a pointer to a member of a class

```
class Car {  
public:  
    int speed;  
};  
...  
int Car::*pSpeed = &Car::speed;
```

- ▶ In this snippet, pSpeed is a pointer to member
- ▶ It is a special kind of pointer: in order to use it, you need to have an object of the class

```
Car obj;  
  
obj.speed = 1;  
cout << "content of pSpeed =" << obj.*pSpeed << endl;  
obj.*pSpeed = 2;  
cout << "content of speed = " << obj.speed << endl;
```

(a rare feature, see [This stackoverflow question](#) for some uses)

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

References

- ▶ In C++ it is possible to define a reference to a variable or to an object

```
int x;           // variable
int &rx = x;     // reference to variable

MyClass obj;     // object
MyClass &r = obj; // reference to object
```

- ▶ `r` is a reference to object `obj`
- ▶ **WARNING!**
 - ▶ C++ uses the same symbol `&` for two different meanings!
 - ▶ when used in a declaration/definition, it is a reference
 - ▶ when used in an instruction, it indicates the address of a variable in memory

References vs pointers

- There is quite a difference between references and pointers

```
MyClass obj;           // the object
MyClass &r = obj;       // a reference
MyClass *p;            // a pointer
p = &obj;               // p takes the address of obj

obj.fun();              // call method fun()
r.fun();                // call the same method by reference
p->fun();                // call the same method by pointer

MyClass obj2;          // another object
p = &obj2;              // p now points to obj2
r = obj2;               // compilation error! Cannot change a reference!
MyClass &r2;            // compilation error! Reference must be initialized
```

- First difference: once you define a reference to an object, the same reference cannot refer to another object later!

Reference vs pointer

- ▶ In C++, a reference is an *alternative name* for an object

Pointers

- ▶ May be uninitialised
- ▶ Pointers are like other variables
- ▶ Can have a pointer to void
- ▶ Can be assigned arbitrary values
- ▶ It is possible to do arithmetic

References

- ▶ Must be initialised
- ▶ Not a "variable", just an alias
- ▶ No references to void
- ▶ Cannot be assigned any value
- ▶ Cannot do arithmetic

Reference example

examples/referencearg.cpp

- ▶ Notice the differences:
 - ▶ Method declaration: `void h(MyClass &c);` instead of `void h(MyClass *p)`
 - ▶ Method call: `h(obj);` instead of `h(&obj)`
 - ▶ In the first case, we are passing a reference to an object
 - ▶ In the second case, the address of an object
- ▶ References are much less powerful than pointers
- ▶ However, they are **much safer** than pointers
 - ▶ The programmer cannot accidentally misuse references, whereas it is easy to misuse pointers

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

Copying objects

- ▶ In the previous example, function `g()` is taking a object by value

```
void g(MyClass c) {...}  
...  
g(obj);
```

- ▶ The original object is copied into parameter `c`
- ▶ The copy is done by invoking the *copy constructor*

```
MyClass(const MyClass &r);
```

- ▶ If the user does not define it, the compiler will define a default one for us automatically
 - ▶ The default copy constructor just performs a bitwise copy of all members

Example

- ▶ Let's add a copy constructor to MyClass, to see when it is called
`examples/copy1.cpp`
- ▶ Now look at the output
 - ▶ The copy constructor is automatically called when we call `g()`
 - ▶ It is not called when we call `h()`

Usage

- ▶ The copy constructor is called every time we initialise a new object to be equal to an existing object

```
MyClass ob1(2);    // call constructor
MyClass ob2(ob1);  // call copy constructor
MyClass ob3 = ob2; // call copy constructor
MyClass ob3{ob2};  // call copy constructor
```

- ▶ We can prevent a copy by making the copy constructor private:

```
// can't be copied!
class MyClass {
    MyClass(const MyClass &r);
public:
    ...
};
```

Copy constructor in C++11

- ▶ In the new standard C++11, the copy constructor can be "hidden" by using keyword "`= delete`" after the member declaration

```
// can't be copied!  
class MyClass {  
public:  
    MyClass();  
    MyClass(const MyClass &r) = delete ;  
    ...  
};
```

Hint

When starting the implementation of a class, disable copy constructor and assignment operator by default, since in most cases you will not need it. This will allow you to catch some additional errors.

Const references

- ▶ Let's analyse the argument of the copy constructor

```
MyClass(const MyClass &r);
```

- ▶ It means:
 - ▶ This function accepts a reference
 - ▶ however, the object will not be modified: it is treated as a *constant* within the scope of the function
 - ▶ The compiler checks that the object is not modified by checking the *constness* of the methods
 - ▶ As a matter of fact, the copy constructor does not modify the original object: it only reads its internal values in order to copy them into the new object
- ▶ If the programmer by mistake tries to modify a field of the original object, the compiler will give an error

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

Meaning of static

- ▶ In C/C++ static has several meanings
 - ▶ for **global variables**, it means that the variable is not exported in the global symbol table to the linker, and cannot be used in other compilation units
 - ▶ **local variables**, it means that the variable is not allocated on the stack: therefore, its value is maintained through different function instances
 - ▶ for class **data members**, it means that there is only one instance of the member across all objects
 - ▶ a static **function member** can only act on static data members of the class

Static members

- ▶ We would like to implement a counter that keeps track of the number of objects that are around
- ▶ We can use a static variable

```
class ManyObj {  
    static int count;  
    int index;  
public:  
    ManyObj();  
    ~ManyObj();  
  
    int getIndex();  
    static int howMany();  
};
```

```
int ManyObj::count = 0;  
  
ManyObj::ManyObj() {  
    index = count++;  
}  
ManyObj::~~ManyObj() {  
    count--;  
}  
  
int ManyObj::getIndex() {  
    return index;  
}  
  
int ManyObj::howMany() {  
    return count;  
}
```

Static members

```
int main()
{
    ManyObj a, b, c, d;
    ManyObj *p = new ManyObj;
    ManyObj *p2 = 0;
    cout << "Index of p: " << p->getIndex() << "\n";
    {
        ManyObj a, b, c, d;
        p2 = new ManyObj;
        cout << "Number of objs: " << ManyObj::howMany() << "\n";
    }
    cout << "Number of objs: " << ManyObj::howMany() << "\n";
    delete p2; delete p;
    cout << "Number of objs: " << ManyObj::howMany() << "\n";
}
```

Output:

```
Index of p: 4
Number of objs: 10
Number of objs: 6
Number of objs: 4
```


Static members

- ▶ There is only one copy of the static variable for all the objects
- ▶ All the objects refer to this variable
- ▶ How to initialize a static member?
 - ▶ cannot be initialized in the class declaration
 - ▶ the compiler does not allocate space for the static member until it is initialized
 - ▶ So, the programmer of the class must define and initialize the static variable

Static data members

- ▶ Static data members need to be initialized when the program starts, before the main is invoked
 - ▶ they can be seen as global initialized variables (and this is how they are implemented)

```
// include file A.hpp
class A {
    static int i;
public:
    A();
    int get();
};
```

```
// src file A.cpp
#include "A.hpp"

int A::i = 0;

A::A() {...}
int A::get() {...}
```

Initialization

It is usually done in the .cpp file where the class is implemented

```
int ManyObj::count = 0;

ManyObj::ManyObj() { index = count++;}
ManyObj::~~ManyObj() {count--;}
int ManyObj::getIndex() {return index;}
int ManyObj::howMany() {return count;}

```

- ▶ There is a famous problem with static members, known as the *static initialization order failure*

The static initialization fiasco

- ▶ When static members are complex objects, that depend on each other, we have to be careful with the order of initialization
 - ▶ initialization is performed just after the loading, and before the main starts.
 - ▶ Within a specific translation unit, the order of initialization of static objects is guaranteed to be the order in which the object definitions appear in that translation unit. The order of destruction is guaranteed to be the reverse of the order of initialization.
 - ▶ However, there is no guarantee concerning the order of initialization of static objects across translation units, and the language provides no way to specify this order. (undefined in C++ standard)
 - ▶ If a static object of class A depends on a static object of class B, we have to make sure that the second object is initialized before the first one

Solutions

- ▶ The **Nifty counter** (or Schwartz counter) technique
 - ▶ Used in the standard library, quite complex as it requires an extra class that takes care of the initialization
- ▶ The **Construction on first use** technique
 - ▶ Much simpler, use the initialization inside function

Construction on first use

- ▶ It takes advantage of the following C/C++ property
 - ▶ Static objects inside functions are only initialized on the first call
- ▶ Therefore, the idea is to declare the static objects inside global functions that return references to the objects themselves
- ▶ Access to the static objects happens only through those global functions (see Singleton)

Copy constructors and static members

What happens if the copy constructor is called?

```
void func(ManyObj a)
{
    ...
}

void main()
{
    ManyObj a;
    func(a);
    cout << "How many: " << ManyObj::howMany() << "\n";
}
```

- ▶ What is the output?
- ▶ Solution in [examples/manyobj.cpp](#)

Outline

Course Contents

Introduction to C++

Classes

Memory Layout

Pointers

References

Copy Constructor

Static

Constants

Const

- ▶ In C++, when something is const it means that it cannot change. Period.
- ▶ Then, the particular meanings of const are a lot:
- ▶ Don't to get lost! Keep in mind: const = cannot change
- ▶ Another thing to remember: constants must have an initial (and final) value!

Constants - I

- ▶ As a first use, `const` can substitute the use of `#define` in C
 - ▶ whenever you need a constant global value, use `const` instead of a `define`, because it is clean and it is type-safe

```
#define PI 3.14           // C style  
const double pi = 3.14; // C++ style
```

- ▶ In this case, the compiler does not allocate storage for `pi`
- ▶ In any case, the `const` object has an `internal linkage`

Constants - II

- ▶ You can use `const` for variables that never change after initialization. However, their initial value is decided at run-time

```
const int i = 100;  
const int j = i + 10;
```

Compile-time constants

```
int main()  
{  
    cout << "Type a character\n";  
    const char c = cin.get();  
    const char c2 = c + 'a';  
    cout << c2;  
  
    c2++;  
}
```

run-time constants

ERROR! c2 is const!

Const and pointers

- There are two possibilities
 1. the pointer itself is constant
 2. the pointed object is constant

```
int a  
int * const u = &a;  
  
const int *v;
```

the pointer is constant

the pointed object is constant (the pointer can change and point to another const int!)

- Remember: a const object needs an initial value!

Const function arguments

- ▶ An argument can be declared constant. It means the function can't change it
- ▶ this is particularly useful with references

```
class A {  
public:  
    int i;  
};  
  
void f(const A &a) {  
    a.i++;           // error! cannot modify a;  
}
```

- ▶ You can do the same thing with a pointer to a constant, but the syntax is messy.

Passing by const reference

- ▶ Remember:
 - ▶ we can pass argument by value, by pointer or by reference
 - ▶ in the last two cases we can declare the pointer or the reference to refer to a constant object: it means the function cannot change it
 - ▶ Passing by constant reference is equivalent, from the user point of view, to passing by value
 - ▶ From an implementation point of view, passing by const reference is much faster!!

Constant member functions

- ▶ A member function can be declared constant
 - ▶ It means that it will not modify the object
 - ▶ The compiler can call only const member functions on a const object

```
class A {  
    int i;  
public:  
    int f() const;  
    void g();  
};  
void A::f() const  
{  
    i++;           // ERROR!  
    return i;  
}
```

```
void myfun(const A &a)  
{  
    a.f();         // Ok  
    a.g();         // ERROR!!  
}
```

constexpr (c++11)

- ▶ Since C++11, we can declare a variable or a function as `constexpr`
 - ▶ `constexpr` means that the object can be computed at **compile time**
 - ▶ so, it is useful for optimization, substitutes and expands inline
 - ▶ for objects, `constexpr` is equivalent to `const`
 - ▶ for functions, it means that the function can be calculated at compile time

```
// C++11 constexpr functions use recursion rather than iteration
// (C++14 constexpr functions may use local variables and loops)
constexpr int factorial(int n)
{
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

int main()
{
    int k = factorial(5); // computed at compile time
    volatile j = 4;
    int h = factorial(j); // computed at run-time
}
```
