C++ essentials

Giuseppe Piero Brandino
and
Jimmy Aguilar Mena

March 9,2016

1 Basics

2 Object Oriented Programming in C++

What is C++?

- C++ is a imperative general-purpose programming language.
- It has object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.
- Created by Bjarne Stroustrup at AT&T Bell Labs in 1983.

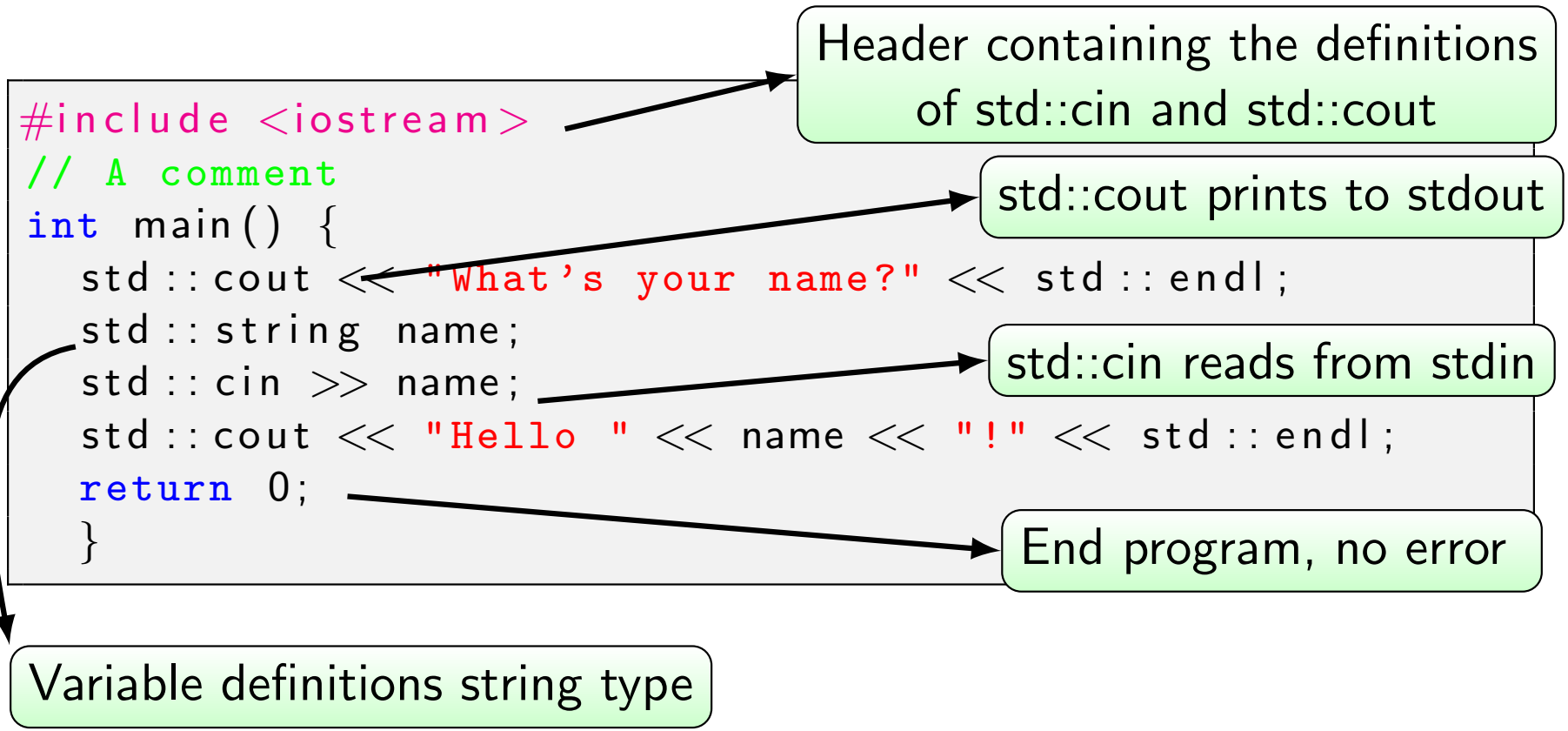
C++ features

- All C features plus:
- Objects
- Exceptions
- Polymorphism
- Run-time type checking
- Generic programming

Your first C++ program

```
#include <iostream>
// A comment
int main() {
  std::cout << "What's your name?" << std::endl;
  std::string name;
  std::cin >> name;
  std::cout << "Hello " << name << "!" << std::endl;
  return 0;
  }
```

Your first C++ program



Memory allocation

- Like in C, the dynamic arrays are pointers, that needs to be explicitly allocated and deallocated.
- Use the **new** operator to allocate arrays or objects.
- Use the **delete []** to deallocate arrays.

```
#include <iostream>
using namespace std;

int main(){
        int *p;
        p = new int[5];

        delete [] p;
        return 0;
    }
```

delete vs delete []

delete [] calls the destructor on every element, while **delete** does not, more on this later)

Function Overloading

In C++ is possible to have functions with the same name but different argument types.

```
#include <iosteam>
#include <iomanip>

void print(int i) {
  std::cout<<i<<std::endl;
  }
void print(double i) {
  std::cout<<std::setprecision(5)<<i<<std::endl;
  }
int main(){
  int a=4;
  double b=3.14159265359;
  print(a);
  print(b);
  return 0;
  }
```

Function Overloading

In C++ is possible to have functions with the same name but different argument types.

```
#include <iosteam>
#include <iomanip>

void print(int i) {
  std::cout<<i<<std::endl;
  }
void print(double i) {
  std::cout<<std::setprecision(5)<<i<<std::endl;
  }
int main(){
  int a=4;
  double b=3.14159265359;
  print(a);
  print(b);
  return 0;
  }
```

Uses print(int)

Uses print(double)

References

The reference is basically an alias of another object

A reference is something similar to a pointer, but it returns the value pointed to (the pointee) and not the address

```
#include <iostream>

using namespace std;
int main(){
  int a=5;
  int &aRef=a;
  cout<<"a = "<<a<<" aRef "<<aRef<<endl;
  a=4;
  cout<<"a = "<<a<<" aRef "<<aRef<<endl;
  aRef=3;
  cout<<"a = "<<a<<" aRef "<<aRef<<endl;
  return 0;
  }
```

output:

a = 5 aRef 5

a = 4 aRef 4

a = 3 aRef 3

References

Passing argument by reference

```
#include <iostream>

using namespace std;
void swap( int &a, int &b){
  int c=a;
  a=b;
  b=c;
  cout<<"Swapping by reference"<<endl;
  }

int main(){
  int a=1, b=2;
  cout<<"a "<<a<<" b "<<b<<endl;
  swap(a,b);
  cout<<"a "<<a<<" b "<<b<<endl;
  return 0;
  }
```

output:

a 1 b 2

Swapping by reference

a 2 b 1

```cpp
#include <iostream>
using namespace std;

void swap( int &a, int &b){
  int c=a;
  a=b;
  b=c;
  cout<<"Swapping by references"<<endl;
  }
void swap( int *a, int *b){
  int c=*a;
  *a=*b;
  *b=c;
  cout<<"Swapping by pointers"<<endl;
  }
int main(){
  int a=1, b=2;
  cout<<"a " <<a<<" b "<<b<<endl;
  swap(a,b);
  cout<<"a "<<a<<" b "<<b<<endl<< endl;
  int c=3, d=4;
  cout<<"c "<<c<<" d "<<d<<endl;
  swap(&c,&d);
  cout<<"c "<<c<<" d "<<d<<endl<< endl;
  return 0;
  }
```

output:
a 1 b 2
Swapping by references
a 2 b 1

c 3 d 4
Swapping by pointers
c 4 d 3

Tip:

Use references if you can
Use pointers if you must

C++ standard headers

- C++ Standard headers are included without the extension.
  So $< iostream >$ and not $< iostream.h >$
  The latter, if existing, is a compiler-provided version and may break portability
- ANSI C Header have a c prepended and the extension removed.
  $\#include < time.h > \rightarrow \#include < ctime >$

1 Basics

2 Object Oriented Programming in C++

Defining a class in C++

```
#include <iostream>

using namespace std;
class rectangle{
        public:
        float width, height;
        float area();
        };

float rectangle::area(){
        return width*height;
        }

int main() {
        rectangle a;
        a.width=5.0;
        a.height=2.0;
        cout << a.area()  << endl;
        return 0;
        }
```

Defining a class in C++

```
#include <iostream>

using namespace std;
class rectangle{
        public:
        float width, height;
        float area();
        };

float rectangle::area(){
        return width*height;
        }

int main() {
        rectangle a;
        a.width=5.0;
        a.height=2.0;
        cout << a.area()  << endl;
        return 0;
        }
```

All member data are privated by default

Member function definition

Accesing member data

Accesing member function

Public and Private

Public variables and functions are visible outside the class. Private variables are visible only inside the class.

```
class myclass{
    public:
        int a;
    private:
        int b;
    }

int main(){
    myclass m;
    m.a=4;
    m.b=3;
    }
```

OK, *"a"* is public

Compilation error *"b"* is private

Pointer to objects

```
include <iostream>
using namespace std;

class rectangle{
  public:
    float width, height;
    float area();
    };

float rectangle::area(){
  return width*height;
  }

int main() {
  rectangle a,*b, *c;
  a.width=5.0;
  a.height=2.0;
  c=&a;

  b= new rectangle;
  b->width=2.0;
  b->height=4.0;

  cout << a.area() << endl;
  cout << b->area() << endl;

  c->width=12.0;
  cout << a.area() << endl;
  delete b;
  return 0;
  }
```

Pointer to object rectangle

"c" points to "a"

b points to an object created by "new" (so in the heap). We use the -> operator

Modifying "c" we modify "a"

b needs to be deallocated, since it was created by new

Constructor and Destructor

- The constructor is the function that performs the steps necessary to correctly create an object
- The destructor performs the steps to correctly destroy it
- If not specified, they are created by the compiler (default constructor), that basically only allocates space for the member variables, and deallocates on destruction
- The constructor is called when a object variable is declared, while the destructor is called when the variable goes out of scope
- For a pointer to object, the constructor is called when using new, while the destructor is called when using delete
- It is possible to specify custom constructor, even containing parameters

Constructor and Destructor

```
#include <iostream>
using namespace std;

class rectangle{
  public:
    float width, height;
    float area();
    rectangle(int a, int b);
  private:
    rectangle(){};
  };

rectangle::rectangle(int a, int b){
  width=a;
  height=b;
  }

float rectangle::area(){
  return width*height;
  }

int main() {
  rectangle a(5.0,2.0);
  //rectangle b;
  cout << a.area() << endl;
  return 0;
  }
```

Custom Constructor

Default constructor as private, such that it is not possible to have an uninitialized rectangle

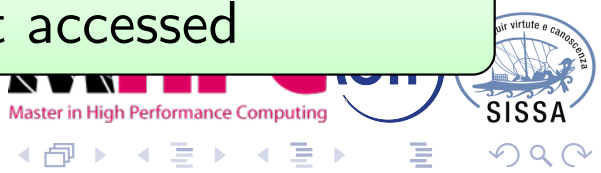
Constructor and Destructor

```
include <iostream>
#include <cmath>
using namespace std;

class myvector{
  private:
    int m_n;
    myvector();
  public:
    float *data;
    int get_n();
    myvector(int n);
    ~myvector();
    };

myvector::myvector(int n){
  m_n=n;
  data= new float[m_n];
  }

myvector::~myvector(){
  delete [] data;
  }

int myvector::get_n(){
  return m_n;
  }
```

Size of the vector is privated

Constructor allocates the data block, and destructor dealloates

The length cannot be modified, just accessed



Accessor, Mutator and Copy Constructor

Accessor and Mutator

Accessor a public method to access private data (a.k.a. "getter") example: **myvector::get_n()**

Mutator a public method to modify private data (typically, in a consistent manner) (a.k.a. setter).

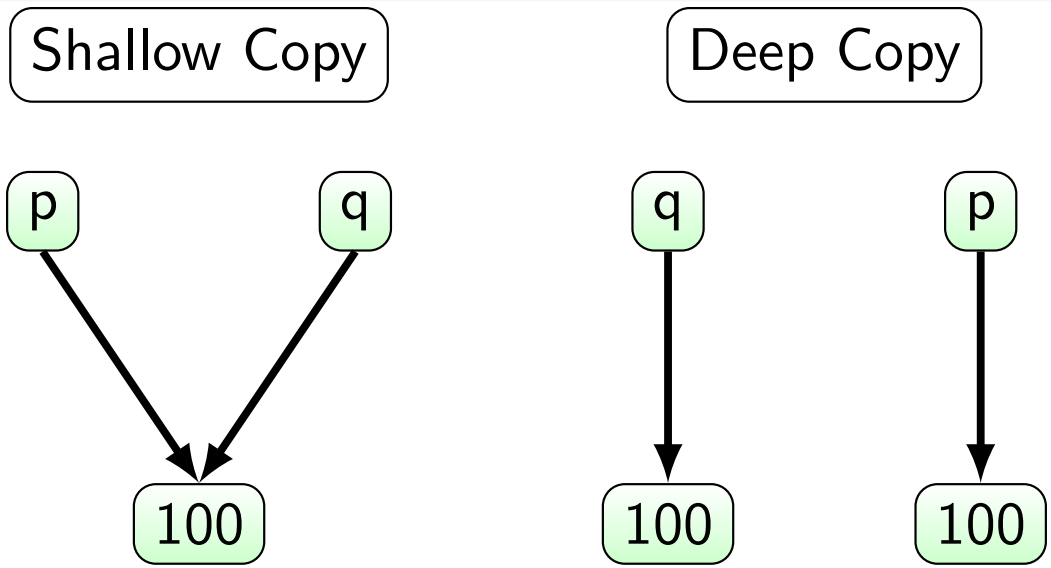
Copy constructor

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.
- myclass(const myclass &obj)
- As for constructor and destructor, the compiler creates a default copy constructor, if not defined.

Deep and shallow copy

Shallow copy If a class contains a pointer, the default copy construct (or a user defined one that just copies the pointer value).

Deep copy Refers instead to the complete duplication, including pointees



This pointer and Const member functions

This Pointer

- The this pointer is an implicit parameter containing the address of the object itself.
- Every call to a member function or member variables inside an object are equivalent to a call with `this->` prepended.
- Useful for example we you need to return a pointer or a reference in a member function.

Const member functions

- Member function declared as *const* cannot modify the object
- float myclass::func(int a, myobj b) const;
- Calling a non-const function on a const object will yield a compiler error (while a const function will work)
- It actually forces the this pointer to be a pointer to constant (const myclass *) for the function scope

Copy Constructor

A copy constructor is used to initialize a previously uninitialized object from some other object's data.

```
rectangle::rectangle(const rectangle &obj){
    cout<<"Invoked copy constructor "<<endl;
    width=obj.width;
    height=obj.height;
        }
```

Assignment Operator

An assignment operator is used to replace the data of a previously initialized object with some other object's data.

```
rectangle &rectangle::operator=(const rectangle &obj){
    cout << "Invoked assignment operator " << endl;
    width=obj.width;
    height=obj.height;
    return *this;
    }
```

Operator Overloading

In C++ the overloading principle applies not only to functions, but to operators too.
That is, operators can be extended to work not just with built-in types but also classes.

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

```
float myvector::operator*( const myvector& a){
    int range = ( m_n < a.get_n() ? m_n : a.get_n());
    float sum=0.0;
    for (int i=0 ; i < range; i++){
        sum+=data[i]*a.data[i];
        }
    return sum;
    }
```

Header files and source files

To keep a code tidy and to allow modularity it is good practice to split class definition and implementation in header file and source file

```
#ifndef MY_VECTOR_HEADER
#define MY_VECTOR_HEADER
class myvector{
  private:
        int m_n;
        myvector();
  public:
        float *data;
        int get_n();
        float norm();
        myvector(int n);
        ~myvector();
        };
#endif
```

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

using namespace std;

myvector::myvector(int n){
  m_n=n;
  data= new float[m_n];
  }

myvector::~myvector(){
  delete [] data;
  }

int myvector::get_n(){
  return m_n;
  }

float myvector::norm(){
  float sum=0.0;
  for (int i=0; i<m_n; i++)
    sum+=data[i]*data[i];
  return sqrt(sum);
  }
```

```
#include <iostream>
#include "myvector.h"
using namespace std;

int main() {

  myvector a(2);
  for (int i=0; i<a.get_n(); i-
    a.data[i]=1.0;
  cout << a.norm()
<< endl;
  return 0;
  }
```

The class header is included in both the class implementation file and the main

Inheritance

Inheritance

- Inheritance is the process of defining a new class based on the definition of another class
- It implements an is-a logic like: A dog is an animal

Protected keyword

- Protected variable and methods are visible from the class itself and derived class there of.
- They are still invisible from outside the class.

```
#ifndef MY_RECTANGLE_HEADER
#define MY_RECTANGLE_HEADER
class rectangle{
public:
  float width, height;
  float area();
  ~rectangle();
  rectangle(float a, float b);
  rectangle(const rectangle &obj);
protected:
  rectangle();
};
#endif
```

```
#ifndef MY_SQUARE_HEADER
#define MY_SQUARE_HEADER
#include "rectangle.h"
class square : public rectangle{
  public:
    ~square();
    square(float a);
    float inscribed_area();
  private:
    square();
  };
#endif
```

```
#include <iostream>
#include "square.h"
using namespace std;

square::square(){
  cout<<"square default constructor"<<endl;
  }

square::square(float a){
  cout<<"square custom constructor"<<endl;
  width=a;
  height=a;
  }

square::~square(){
  cout << " square destructor " << endl;
  }

float square::inscribed_area(){
  return 3.1415*width*width/4.0;
  }
```

Width, height and area() are inherithed. inscribed_area() is specific to the square

Inheritance

- Unless otherwise specified, the constructor of a derived class calls the default constructor of its parent classes (i.e., the constructor taking no arguments).
- The destructor of a derived class calls the destructor of its parent class
- Constructor are called in the order parent $\rightarrow$ derived
- Destructor are called in the order derived $\rightarrow$ parent

Polymorphism

Since a derived object is-a base object (a square is a rectangle), it is possible to to assign to a base pointer the address of a derived variable

```
rectangle *p_a=new square(1.0);

square b(3.0,5.0);
rectangle *p_b=&b;
```

```
float rectangle::area(){
  cout<<"Invoked rectangle area"<<endl;
  return width*height;
  }
```

```
float square::area(){
  cout<<"Invoked square area"<<endl;
  return width*width;
  }
```

```
int main() {
  square c(2.0);
  c.area();

  rectangle *d;
  d=&c;
  d->area();

  rectangle& e=c;
  e.area();
  return 0;
  }
```

What happens if when we call area() from a pointer to base class?

The base class version of area() is called

Static and dynamic dispatch

- The previous behavior is called **static dispatch**. At compile time, the method is associated to the pointer.
- It is possible to have **dynamic dispatch**. At run-time, the program chooses which version of the method to call
- This is done through the usage of the **virtual** keyword in the method declaration
- To call the base method, you need to use the scope resolution operator: *parent::method()*

Polymorphism

Dynamic dispatch

```
class rectangle{
  public:
    float width, height;
    virtual float area();
    ~rectangle();
    rectangle(int a, int b);
    rectangle(const rectangle &obj);
  protected:
    rectangle();
  };
```

```
#include "rectangle.h"
class square : public rectangle{
  public:
    ~square();
    square(int a);
    float inscribed_area();
    float area();
  private:
    square();
  };
```

```
int main() {
square c(2.0);
c.area();

rectangle *d;
d=&c;
d->area();

rectangle& e=c;
e.area();
return 0;
}
```

Now the correct method is resolved

Virtual destructor

If we delete a base class pointer, that it is actually pointing to a derived class, only the base class destructor is called, leading to possible undefined behaviour or memory leaks.

Virtual destructor

- The solution is to declare the destructor as virtual.
- In this way, the destructor of the derived class will be called, and from it the one of its parent, up to the base class.
- In general, it is good practice to declare destructor as virtual.