Very Brief Introduction to C++

Computer Vision 2018

P. Zanuttigh

Some slides M. Bowdoin, F. Bari, Pearson Education

What is C++ ?

- C++ is an enhanced version of the C language
- C++ adds support for Object-Oriented Programming (OOP) without sacrificing any of C’s power, elegance, or flexibility
- C++ was invented in 1979 by Bjarne Stroustrup at Bell Laboratories in Murray Hill, New Jersey, USA
- This is the short version, see the complete presentation if you need more details on C++

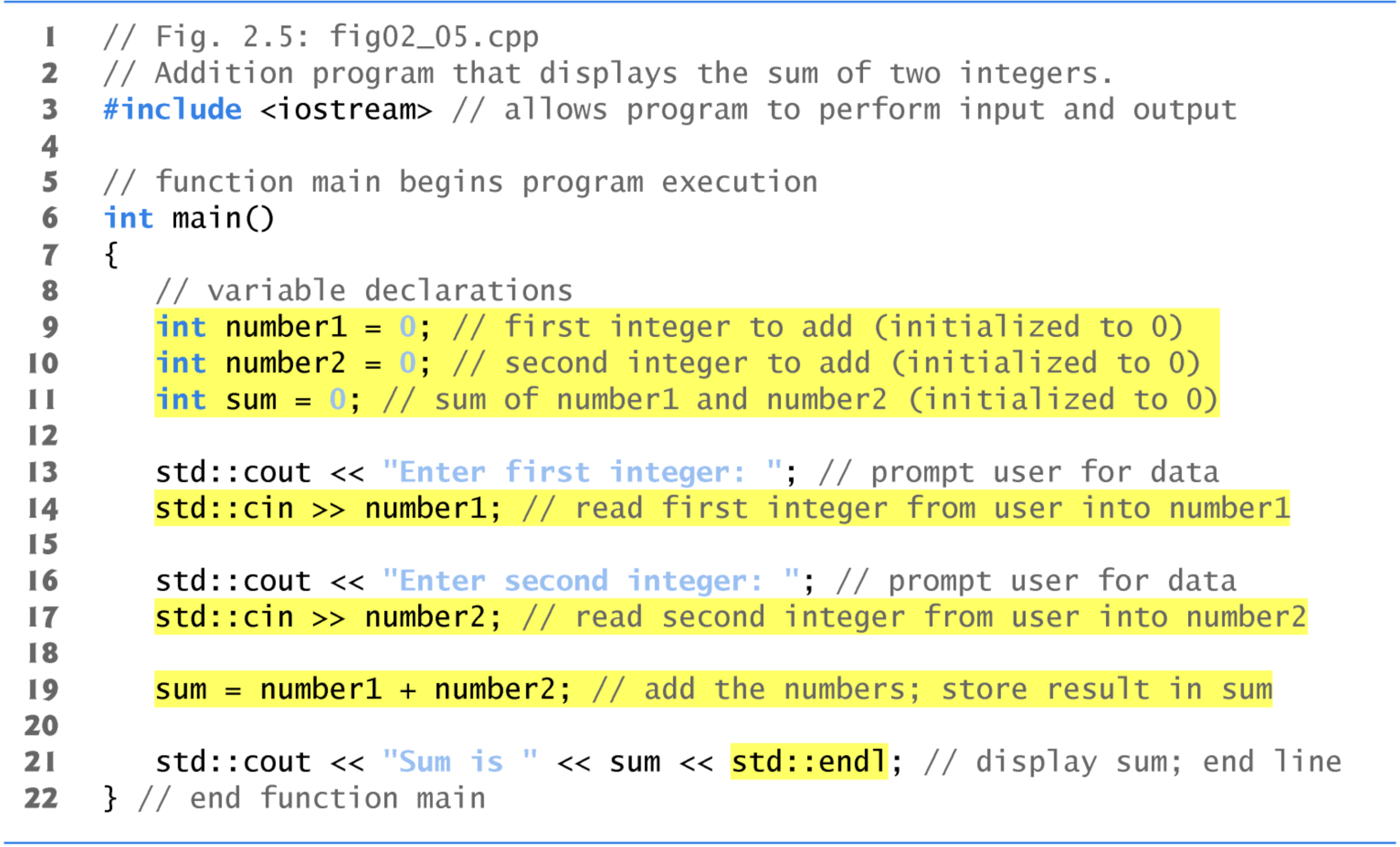
A Simple C++ Program

```
//include headers (modules that include functions used by your program)
#include <iostream>

int main() {

    std::cout << “Hello world!”;
    return 0;
}
```

After you write a C++ program you compile it; that is, you run a program called **compiler** that checks whether the program follows the C++ syntax

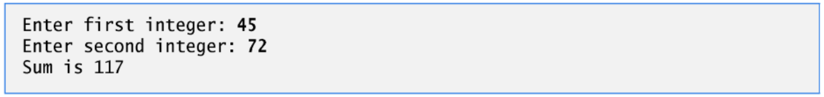
- □ if it finds errors, it lists them
- □ If there are no errors, it translates the C++ program into a program in machine language which you can execute

Simple Sum Program



```cpp
// Fig. 2.5: fig02_05.cpp
// Addition program that displays the sum of two integers.
#include <iostream> // allows program to perform input and output

// function main begins program execution
int main()
{
    // variable declarations
    int number1 = 0; // first integer to add (initialized to 0)
    int number2 = 0; // second integer to add (initialized to 0)
    int sum = 0; // sum of number1 and number2 (initialized to 0)

    std::cout << "Enter first integer: "; // prompt user for data
    std::cin >> number1; // read first integer from user into number1

    std::cout << "Enter second integer: "; // prompt user for data
    std::cin >> number2; // read second integer from user into number2

    sum = number1 + number2; // add the numbers; store result in sum

    std::cout << "Sum is " << sum << std::endl; // display sum; end line
} // end function main
```

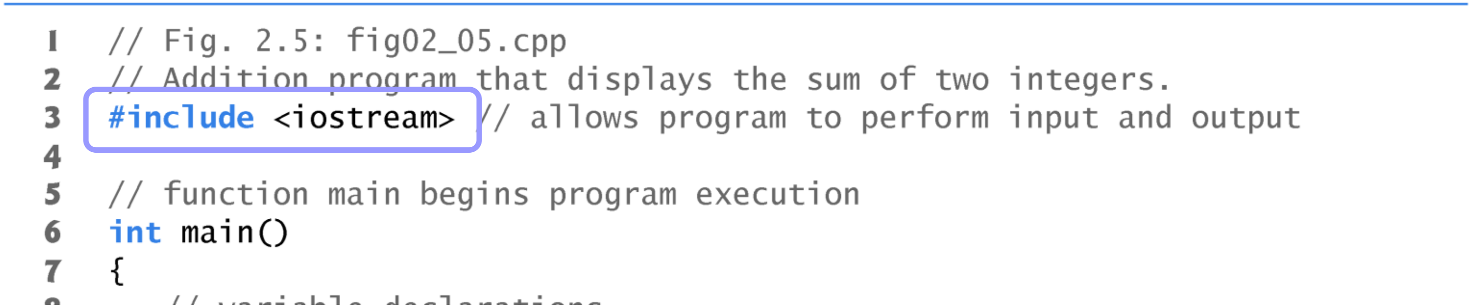
Fig. 2.5 | Addition program that displays the sum of two integers.



```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.5 | Addition program that displays the sum of two integers.

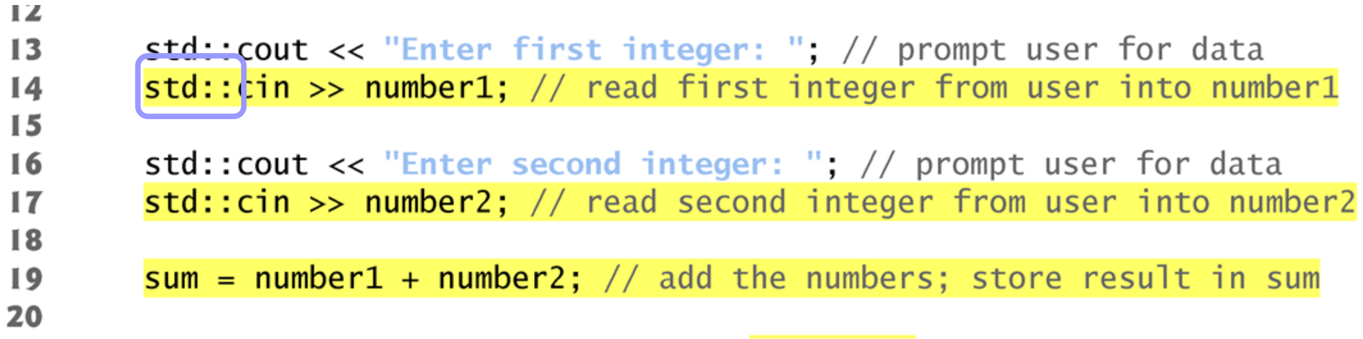
Include Headers



```
// Fig. 2.5: fig02_05.cpp
// Addition program that displays the sum of two integers.
#include <iostream> // allows program to perform input and output

// function main begins program execution
int main()
{
   // variable declarations
```

- A preprocessing directive is a message to the C++ preprocessor
- Lines that begin with # are processed by the preprocessor before the program is compiled
- The new-style headers do not specify filenames
- They simply specify standard identifiers that might be mapped to files by the compiler, but they need not be
  - □ <iostream>, <vector>
  - □ ….
- Programmer defined header files should end in “.h”
  - □ #include "myheader.h"

Namespaces



```
std::cout << "Enter first integer: "; // prompt user for data
std::cin >> number1; // read first integer from user into number1

std::cout << "Enter second integer: "; // prompt user for data
std::cin >> number2; // read second integer from user into number2

sum = number1 + number2; // add the numbers; store result in sum
```

- A namespace is a declarative region
- It localizes the names of identifiers to avoid name collisions
- The Binary Scope Resolution Operator (::) is used to associate a member function with its class
- The contents of new-style headers are placed in the **std** namespace
- using declarations eliminate the need to repeat the std:: prefix
  - example: using namespace std; → ~~std::~~cout
- A newly created class, function or global variable can be put in an existing namespace, a new namespace, or it may not be associated with any namespace
  - In the last case the element will be placed in the global unnamed namespace

Memory Concepts

- Variable names such as *number1*, *number2* and *sum* actually correspond to locations in the computer's memory
- Every variable has a name, a type, a size and a value
- When a value is placed in a memory location, the value overwrites the previous value in that location; thus, placing a new value into a memory location is said to be destructive
- When a value is read out of a memory location, the process is nondestructive

Pointers (1)

- Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them
- Every variable is a memory location and every memory location has its address defined which can be accessed using the *and* (&) operator which denotes an address in memory

Consider the following which will print the address of the variables defined −

```
#include <iostream>
using namespace std;
int main () {
        int var1 = 5;
        cout << "Address of var variable: ";
        cout << &var1 << endl;
        cout << "Value of var variable: ";
        cout << var1 << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: 0xbfebd5c0
Value of var variable: 5
```

Pointers (2)

- A **pointer** is a variable whose value is the address of another variable
- Like any variable or constant, you must declare a pointer before you can work with it
  - The general form of a pointer variable declaration is type *var-name;
  - Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable.
- The asterisk is being used to designate a variable as a pointer
  - Example: int *ip; // pointer to an integer
- The *actual data type* of the value of all pointers, whether integer, float, character, or otherwise, *is the same*, a long hexadecimal number that represents a memory address
- The only difference between pointers of different data types is the data type of the variable that the pointer points to

Pointers (3)

- There are few important operations, which we will do with the pointers very frequently
  a) We define a pointer variable
  b) Assign the address of a variable to a pointer
  c) Finally access the value at the address available in the pointer variable
- This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Pointers: Example

```
#include <iostream>
using namespace std;
int main () {
    int var = 20; // actual variable declaration.
    int *ip; // pointer variable
    ip = &var; // store address of var in pointer variable
    cout << "Value of var variable: ";
    cout << var << endl;
    // print the address stored in ip pointer variable
    cout << "Address stored in ip variable: ";
    cout << ip << endl;
    // access the value at the address in pointer
    cout << "Value of *ip variable: ";
    cout << *ip << endl;
    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

from https://www.tutorialspoint.com/cplusplus/cpp_pointers.htm

References

- ❑ A reference variable is an alias, that is, another name for an already existing variable
- ❑ Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable

References vs Pointers

References are often confused with pointers but three major differences between references and pointers are:

1. You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage
2. Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time
3. A reference must be initialized when it is created. Pointers can be initialized at any time

Creating References in C++

- Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location
- You can access the contents of the variable through either the original variable name or the reference
- References are usually used for function argument lists and function return values.

```
int i = 17;
//We can declare reference variables for i as follows.
int& r = i;
// Read the & in this declaration as a reference. Thus, read the  declaration
as "r is an integer reference initialized to i"
```

Example: References

```
#include <iostream>
using namespace std;
int main () {
        // declare simple variables
        int i = 5;
        // declare reference variables
        int& r = i;
        // init standard variable to the value of i
        int j = i;

        cout << "Value of i : " << i << endl;
        cout << "Value of i reference : " << r << endl;
        cout << "Value of j: " << j << endl;

        i = 7;
        cout << "Value of i : " << i << endl;
        cout << "Value of i reference : " << r << endl;
        cout << "Value of j: " << j << endl;
}
```

When the above code is compiled together and executed, it produces the following result

```
Value of i : 5
Value of i reference : 5
Value of j: 5

Value of i : 7
Value of i reference : 7
Value of j: 5
```

Classes: A First Look

General syntax:

```
class class-name
{
            // private functions and variables
public:

            // public functions and variables
}object-list (optional);
```

- A class declaration is a logical abstraction that defines a new type
- It determines what an object of that type will look like
- An object declaration creates a physical entity of that type
  - An object occupies memory space, a type definition does not
- Each object of a class has its own copy of every variable declared within the class, but they all share the same copy of member functions

Constructors

- Every object we create will require some sort of initialization

- A class constructor is automatically called by the compiler each time an object of that class is created

- A constructor function has the ***same name*** as the class and has ***no return type***

- There is no explicit way to call the constructor

```
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

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

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Destructors

- The complement of a constructor is the destructor
  - It does any necessary clean up before the object is removed from memory.
- It is automatically called by the compiler when an object is destroyed
  - When an object goes out of scope normally
  - When a dynamically allocated object is explicitly deleted using the delete keyword
- The name of a destructor is the ***name of its class***, preceded by a **~**
- For classes that just initialize the values of normal member variables a destructor is not needed, C++ will automatically clean up the memory
- If the class object is holding any resources (e.g. dynamic memory, or a file or database handle), or if you need to do any kind of maintenance before the object is destroyed, the destructor is needed

```
#include <iostream>
#include <cassert>

class IntArray
{
private:
  int *m_array;
  int m_length;

public:
  IntArray(int length) // constructor
                    {
                    assert(length > 0);

                    m_array = new int[length];
                    m_length = length; //destr. not needed
                    }

  ~IntArray() // destructor
                    {
                    // Dynamically delete the array
                    delete[] m_array ;
                    }

   void setValue(int ind, int val) { m_array[ind] = val; }
   int getValue(int ind) { return m_array[ind]; }
   int getLength() { return m_length; }
};
```

Constructors & Destructors

- For global objects, an object’s constructor is called once, when the program first begins execution
- For local objects, the constructor is called each time the declaration statement is executed
- Local objects are destroyed when they go out of scope
- Global objects are destroyed when the program ends