C++ Course 7: Lambda expressions. IO streams.

2020 by Oleksiy Grechnyev

Functions as parameters/variables ?

Can we pass a function as an argument to another function?

Example 1: A binary operation :

```
int add(int x, int y) {          // Add two int numbers
        return x + y;
}
...
void printOp(??? op) {           // Apply binary operation op to 7 and 3
        cout << "printOp: op(7, 3) = " << op(7, 3) << endl;
}   // What type do we use instead of ??? ?
...
printOp(add);       // Pass function add as an argument to printOp
??? myOp = add;       // Variable to hold a function
printOp(myOp);      // Pass function myOp as an argument to printOp
```

What type *???* can hold a function?

Why do we need such things ???

Example 2: Operations for Algorithms : Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
int sum_10(??? fun){   // A trivial algorithm
        int sum = 0;
        for (int i = 1; i<=10; ++i)
                sum += fun(i);
        return sum;
}
```

Example 3: Callbacks: Errors, threads, events, signals, ...:

```
void errorCallback(const string & s){
        cerr << s;
        exit(1);
}
void setErrorCallback(??? cb) {...}
```

Solution 1: C-style function pointers (don't do this !)

Example 1: A binary operation :

```
int add(int x, int y) {return x+y;}
void printOp(int (*op) (int, int)) {...}
int (*myOp) (int, int) = add;     // Variable
printOp(add);     // Pass add to printOp
```

Example 2: Operations for Algorithms : Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
int sum_10(int (*fun) (int) ){ ... }
int result = sum_10(square);
```

Example 3: Callbacks: Errors, threads, events, signals, ...:

```
void errorCallback(const string & s){...}
void setErrorCallback(void (*cb)(const string &)) {...}
setErrorCallback(errorCallback);
```

Solution 2: std::function (Usually the best one)

Example 1: A binary operation :

```
int add(int x, int y) {return x+y;}
void printOp(function<int(int, int)> op) {...}
function<int(int, int)> myOp = add;     // Variable
printOp(add);     // Pass add to printOp
```

Example 2: Operations for Algorithms : Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
int sum_10(function<int(int)> fun){ ... }
int result = sum_10(square);
```

Example 3: Callbacks: Errors, threads, events, signals, ...:

```
void errorCallback(const string & s){...}
void setErrorCallback(function<void(const string &)> cb) {...}
setErrorCallback(errorCallback);
```

Solution 3: Templates (compile-time instantiation)

Example 1: A binary operation :

```
int add(int x, int y) {return x+y;}
template <typename T>
void printOp(T op) {...}
printOp(add);     // Pass add to printOp
```

Example 2: Operations for Algorithms : Sum a function applied to numbers 1 .. 10:

```
int square(int x) {return x*x;}
template <typename T> int sum_10(T fun){ ... }
int result = sum_10(square);
```

Example 3: Callbacks: Errors, threads, events, signals, ...:

```
void errorCallback(const string & s){...}
template <typename T>  void setErrorCallback(T cb) {...}
setErrorCallback(errorCallback);
```

Why is std::function better than function pointers?

1. **std::function** can be used with *functors* and *lambda expressions*.
2. **std::function** allows type conversions
3. Function pointer types look ugly

```
int addRef(const int & x, const int & y) {      // Different signature !
        return x + y;
}

...
function<int(int, int)> myOp = addRef;      // Works !
printOp(addRef);      // Works !
```

Functors

Functor class is a class with overloaded **operator()**

Functor object can be invoked as a function

```
struct FunctorAdd {       // struct is a class with default public: access
        int p = 0;                // Parameter
        int operator() (int x, int y) {
                return x + y + p;
        }
};
FunctorAdd fa{17};     // Sets p=17
cout << fa(1, 2);     // Can be called as a function, prints 20
printOp(fa);           // Can be passed to printOp (std::function or template version)
```

OpenCV example: **cv::Mat** is a functor!

```
Mat a = imread("my.png");
Mat b = a(Rect2i(100, 100, 300, 200));   // Crop a subimage defined by a Rect
```

Use functor to add functionality to *existing* class, otherwise use *lambda expressions* !

Lambda expressions

Lambda expression creates a functor object of an *anonymous class* :

```
printOp( [](int x, int y)->int{
        return x + y;
} );
```

Use **auto** (or **std::function**) to store it in a variable:

```
auto myOp = [](int x, int y)->int{
        return x + y;
};
printOp(myOp);      // Pass myOp to printOp
```

Lambda expression syntax:

```
[<capture list>] (<parameters list>) -> <return type> {<body>}
```

Lambda with **auto** (C++ 14), creates a functor template (sort of):

```
[](auto x, auto y) {return x + y;}
```

Example 1 with lambdas

```
void printOp(function<int(int, int)> op) {
        cout << "printOp: op(7, 3) = " << op(7, 3) << endl;
}

int main(){
        printOp( [](int x, int y)->int{
                return x + y;
        } );
        auto myOp = [](int x, int y)->int{
                return x + y;
        };
        printOp(myOp);      // Pass myOp to printOp
}
```

Terminology: Lambda expression vs Closure

Lambda expression defined an anonymous functor class and creates an object of it.

Captured variables are fields of the class.

Lambda expression :
The expression **[]()->{}** in the code.

Closure class :
The anonymous functor class defined by the lambda expression.

Closure :
The object of this class created by the lambda expression.

Capture

Lambdas can capture local variables from the outlying scope:

```
int a = 1, b = 2, d = 3, e = 4;
auto myLambda = [a, &b, c = d + e + 18, this] ()->void{
        cout << a << " " << b << " "  << c << endl;
};
```

Capture by value : **a**

Capture by reference : **&b**

Init capture (C++ 14) : **c = d + e + 18**

Class object capture : **this**
Only for lambdas defined within class methods.

Capture by value

Variables captured by value are *copied when the lambda is created*.
Note: if **a** is pointer, then the pointer is copied and not the data it points at!
Same with **std::shared_ptr** and **cv::Mat** (shallow copy).

```
int a = 13;     // Here a = 13

auto lam = [a]()->void{
        cout << "a = " << a << endl;
};

a = 666;     // Now a = 666

lam();        // What is printed ?
```

Capture by reference

Variables captured by reference are *stored as reference*.

```
int a = 13;     // Here a = 13

auto lam = [& a]()->void{
        cout << "a = " << a << endl;
};

a = 666;     // Now a = 666

lam();         // What is printed ?
```

Examples 2, 3 with lambdas

Example 2: Operations for Algorithms : Sum a function applied to numbers 1 .. 10:

```
int sum_10(function<int(int)> fun){
        int sum = 0;
        for (int i = 1; i<=10; ++i) sum += fun(i);
        return sum;
}
...
sum_10([](int x) -> int {return x*x;} );
```

Example 3: Callbacks: Errors, threads, events, signals, ...:

```
void setErrorCallback(function<void(const string &)> cb) {...} ...
setErrorCallback([](const string & s) noexcept  ->void{
        cerr << s;
        exit(1);
});
```

Capturing this

For lambdas defined inside a class method: Allows the use of class fields in the lambda.

this is captured *by value* as a *pointer*. The class object itself in *NOT copied* !

```
class MyClass{
        ...
        void run(){ // Capture this, not individual class fields !
                printOp([this](int x, int y)->int{
                                return x + y + p;
                });
        }
        ...
        int p;
};
MyClass mc;
...
mc.run();
```

Init capture or generalized lambda capture (C++ 14)

```
int d = 3, e = 4;
printOp( [p = d*d*2 ](int x, int y)->int{      // By value
        return x + y + p;
});
printOp( [&p = d ](int x, int y)->int{return x + y + p;});        // By reference
```

Init capture can *move* objects into the lambda :

```
auto ul = make_unique<int>(3);      // A unique_ptr cannot be copied, only moved !
printOp([u = move(ul)](int x, int y)->int{
        return x * y * *u;
});
```

Problem : such lambda does not work with **std::function** !!!

std::function requires lambdas to be *copyable* !

Extra slides: Capture all, capture in functors

Reducing the number of parameters

```
int add3(int x, int p, int y) {
        return x + p + y;
}   // One extra parameter !
```

How can we use it with **printOp**, which needs binary operation (2 parameters)?

Reducing the number of parameters

```
int add3(int x, int p, int y) {
        return x + p + y;
}   // One extra parameter !
```

How can we use it with **printOp**, which needs binary operation (2 parameters)?

With a lambda wrapper (preferred) :

```
printOp([](int x, int y)->int{      // Correct signature !
        return add3(x, 10, y);
});
```

With **std::bind** (returns a functor object) :

```
printOp(bind(add3, placeholders::_1, 10, placeholders::_2));
```

Use lambdas, not **std::bind** !

Extra slides: Using operators and class methods with std::function

Using lambdas in algorithms:

How to sort in C++ ?

```
vector <int> v{3, 17, 3, 81, -20, 0, 685, 185, -9, 37, 62};
sort(v.begin(), v.end());    // Sort in ascending order, uses operator<
// -20 -9 0 3 3 17 37 62 81 185 685
```

How to sort in *descending* order ?

Write a lambda:

```
sort(v.begin(), v.end(), [](int x, int y)->bool{
        return x > y;
});
```

Or use **std::greater<>** : a wrapper for **operator>**:

```
sort(v.begin(), v.end(), greater<int>());      // C++ 11
sort(v.begin(), v.end(), greater<>());      // C++ 14
```

for_each, count_if, transform

Run a lambda for every element:

```
for_each(v.cbegin(), v.cend(), [](int x)->void{
        cout << x << " : " << 2*x << endl;
});
```

Count negative elements:

```
int n = count_if(v.cbegin(), v.cend(), [](int x)->bool{
        return x < 0;
});
```

Apply a transformation function for each element:

```
transform(v.cbegin(), v.cend(), back_inserter(v2), [](int x)->int{
        return x*2;
});
```

std::back_inserter is a special iterator that does **push_back** (rather than overwrites !)

generate

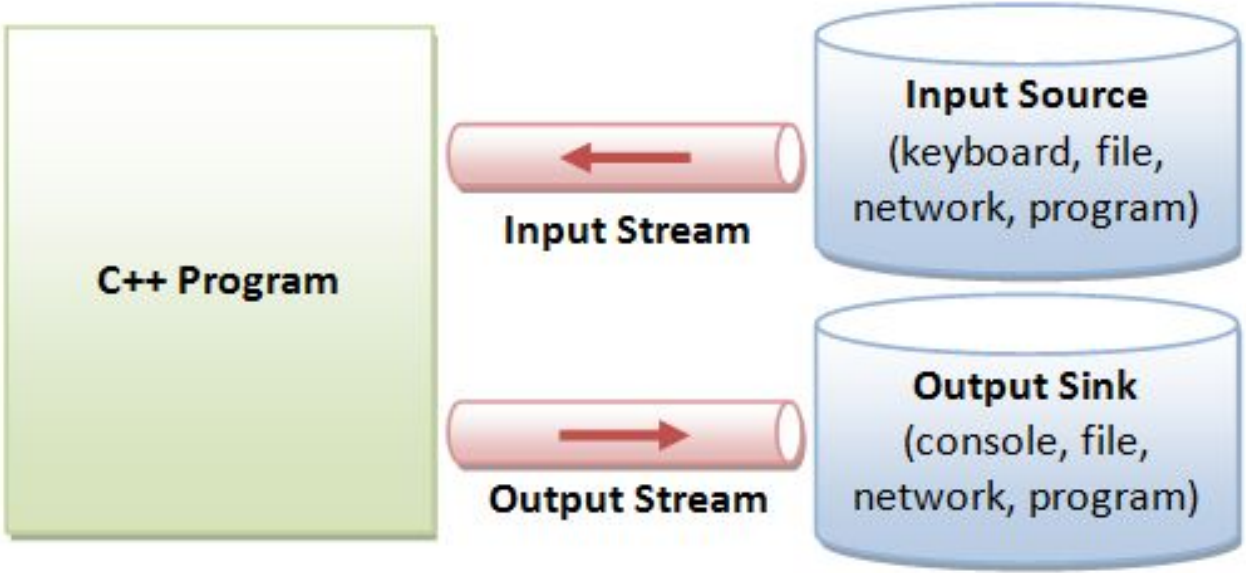
Generate a sequence of elements:

```
vector<int> v(10);        // Pre-allocate 10 elements
int n = 0;
generate(v.begin(), v.end(), [&n]()->int{
        return n++;
});
// 0 1 2 3 4 5 6 7 8 9
```

The same using number of elements and **std::back_inserter** :

```
vector<int> v;        // No  pre-allocation !
int n = 0;
generate_n(back_inserter(v), 10, [&n]()->int{
        return n++;
});
// 0 1 2 3 4 5 6 7 8 9
```

I/O streams



Internal Data Formats:

- Text: `char`, `wchar_t`
- `int`, `float`, `double`, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Standard streams cin, cout, cerr, clog

cin : Standard input (buffered)
cin : Standard output (buffered)
cerr : Standard error output (no buffer)
clog : Standard error output to **cerr** with buffer ?

operator **>>** : Input operator (overloaded bit shift operator !)
operator **<<** : Input operator

```
double c, d;
cout << "Enter c, d :" << endl;
cin >> c >> d;
cout << "c = " << c << " , d = " << d << " , c*d = " << c*d << endl;
cerr << "Fatal error : file " << fileName << " not found ! \n";
```

Operators **>>, <<** return the stream itself, which allows us to *chain* operators
For example: **cin >> c** returns **cin**.

Reading strings

```
string name;
cout << "Your full name ?" << endl;
cin >> name;
```

Type in the console: Sir Philip Anthony Hopkins

name == ????

Reading strings

```
string name;
cout << "Your full name ?" << endl;
cin >> name;
```

Type in the console: Sir Philip Anthony Hopkins

name == "Sir" !!!

The proper way to do it :

getline(cin, name);

Now **name** == "Sir Philip Anthony Hopkins"

getline() reads until EOL (End of Line)

Extra slides: stream classes and headers

Stream status and errors

Status bits:

good = Everything is OK
fail = Error (e.g. failed to red **int**, or EOF)
bad = Serious error
eof = End of file

Class methods: **cin.good()**, **cin.fail()**, **cin.bad()**, **cin.eof()**
Clear after error: **cin.clear()**
Cast to **bool** : same as (**!cin.fail()**) :

```
int a, b;
if (cin >> a >> b)
        cout << "a = " << a << ", b = " << b << endl;
else
        cerr << "Error !" << endl;
```

Exceptions in I/O streams

By default **istream**, **ostream** throw no exceptions (historical reasons ?)

How to turn on exceptions for **cin**:

```
cin.exceptions (ios::eofbit | ios::failbit | ios::badbit);
```

Now **cin** throws **ios_base::failure** if anything is wrong!

```
try {
        cin >> a >> b;
        ...
} catch (const ios_base::failure & e) {
        cerr << e. what() << endl;
}
```

Extra slides: Unicode issues, manipulators, C-streams

String streams: istringstream, ostringstream, stringstream

String streams allow us to use strings as streams (as files)
Use **str()** (getter and setter) to access the underlying string
This can be useful for formatting or parsing strings

```
istringstream iss("13.98  17.32");
ostringstream oss;
double a, b;
iss >> a >> b;
oss << "a = " << a << " , b = " << b << " , a*b = " << a*b << endl;
cout << "oss.str() = " << oss.str();    // Contents of oss
```

If we want to reuse **iss** -- Если мы хотим снова использовать **iss** :

```
iss.str("3.0 7.0");      // Change the string in iss
iss.clear();             // To avoid failure on EOF !
```

We need **clear()** to clear the EOF bit !

File streams : ifstream, ofstream, fstream

To open an input file:

ifstream in("in_file.txt");

or

ifstream in;

in.open("in_file.txt");

To create/overwrite output file:

ofstream out("out_file.txt");

To append at the end of file:

ofstream out("out_file.txt", ios::app | ios::out);

To close a file:

out.close();

Note: No need to do that, destructor calls **close()** !

Stream open modes

ios::app seek to the end of stream before each write
ios::binary open in binary mode (matters in Windows, EOL handling)
ios::in open for reading
ios::out open for writing
ios::trunc discard the contents of the stream when opening
ios::ate seek to the end of stream immediately after open

Joined by the **|** (bitwise OR) operator, for example

```
ofstream o1("out1.dat", ios::out | ios::binary);   // Out, replace, binary
ofstream o2("out2.dat", ios::out | ios::app | ios::binary);   // Out, append, binary
ifstream  i1("in1.dat", ios::in | ios::binary);   // In, binary
```

Stream methods and functions

Read/write **char**, C-strings, **streambuf**

get()

put()

Read/write **string**

getline() (method)

getline() (function)

<< (write)

Read/write buffers:

read()

write()

readsome()

Copy files : 2 ways

Using **get**, **put** by 1 character (slow !):

```
char c;
while (in.get(c))
    out.put(c);
```

Using **read**, **write** (efficient, serialize binary data in C++ like this !):

```
constexpr streamsize SIZE = 1024;   // Buffer size
char buf[SIZE];                     // Buffer
streamsize count;

do {
    in.read(buf, SIZE);         // Read up to SIZE chars to the buffer
    count = in.gcount();        // Get the actual count
    out.write(buf, count);       // Write count chars
} while (count > 0);            // Exit if count == 0
```

gcount() returns number of bytes actually read, up to **SIZE**.

std::filesystem (C++ 17)

Cross-platform filesystem operations (paths, directories, create/delete files etc.)

Based on: Boost Filesystem

Note: On Ubuntu there is a bug with gcc 8.3.0, use gcc-9 instead ! Configure like this:

```
cmake -DCMAKE_C_COMPILER=gcc-9 -DCMAKE_CXX_COMPILER=g++-9 ..
```

std::filesystem::path is the basic path object (the file doesn't have to exist !)

```
filesystem::path p("."); // Current directory

// Check that file/directory exists
cout << filesystem::exists(p) << endl;
// Check that it is a directory
cout << filesystem::is_directory(p) << endl;
// Convert p to absolute path
cout << filesystem::absolute(p) << endl;
// Convert p to std::string
cout << p.string() << endl;
```

std::filesystem (C++ 17)

Iterate over a directory **p**:

```
for (const filesystem::directory_entry &de :
filesystem::directory_iterator(p)) {
    cout << de.path() << endl;
}
```

Create a new path **p2** and create directory from this path:

```
filesystem::path p2 = p / "newdir";    // Overloaded / operator !

// Does not exist yet !
cout << filesystem::exists(p2) << endl;

// Create directory
filesystem::create_directory( p2);
```

Library of the day : gtkmm

Suppose you want a desktop GUI.
GUI is best in C++ (Electron, Java Swing, ... = SLOW)

Cross-platform C++ GUI libraries:
gtkmm = Nice
Qt = Not so nice
wxWidgets ...
Most other C++ GUI libraries are lightweight, outdated, or not cross-platform.

What NOT to do:
"GUI" with OpenCV. To enable buttons, build OpenCV with Qt.
!!! BAD IDEA !!! Approach is not scalable from demo to product !
If you want GUI, use a *real* GUI library !

Note: **gtkmm** is a wrapper to the C library **Gtk+**
GObject is a "object oriented" framework in C. **mm** stands for "--".

Using gtkmm in a CMake project

pkgconfig is a package-finding system for C.

It is typically used with **configure**/ **make**. Here we use it with CMake.

```
find_package(PkgConfig)      # Find PkgConfig CMake plugin
pkg_check_modules(GTKMM gtkmm-3.0)    # Find gtkmm-3.0 with pkgconfig

link_directories(${GTKMM_LIBRARY_DIRS})
include_directories(${GTKMM_INCLUDE_DIRS})

set(SRCS
    HelloWorld.h
    main.cpp
    )

# Link gtkmm libraries
add_executable(${PROJECT_NAME} ${SRCS})
target_link_libraries(${PROJECT_NAME} ${GTKMM_LIBRARIES})
```

gtkmm code example: main.cpp

```
#include <gtkmm/application.h>
#include "./HelloWorld.h"

int main(int argc, char** argv) {
    // Create the gtkmm application
    auto app = Gtk::Application::create(argc, argv, "org.hell");
    HelloWorld hw;          // Create the main window object
    return app->run(hw);    // Run application app with the main window hw
}
```

GUI applications do not follow sequential logic.

app->run(hw) runs the event loop and exits when the window is closed.

Everything else is done via *events*, *signals* and *callbacks*.

Here our main window is of our own class **HelloWorld**.

gtkmm code example: HelloWorld.h

```
#pragma once
#include <iostream>
#include <gtkmm/button.h>
#include <gtkmm/window.h>

class HelloWorld : public Gtk::Window {
public:
    HelloWorld() {     // Ctor
        set_title("Goblin Window");
        // Add a callback (lambda) to the button
        btn.signal_clicked().connect([]()->void {
            std::cout << "Goblin button pushed !!!" << std::endl;
        });
        add(btn);    // Add button to the window (used as container)
        btn.show(); // Make button visible
    }

protected:
    Gtk::Button btn{"Goblin Button"};     // A button
};
```

Thank you for your attention !

title

text

Capture with lambda and functor

```
int p = 3;

Capture p by a lambda :
printOp( [p](int x, int y)->int{
        return x + y + p;
}

Capture p by a functor :
struct {
        int pCap;       // Parameter
        int operator()(int x, int y) {
                return x + y + pCap;
        }
} functor{p};   // pCap captures p by value
printOp(functor);
```

Default capture (Don't do this !!!)

Suppose we have many variables :

int a = 1, b = 2, d = 3, e = 4;

Capture *everything* by value :

lam = [=]()->void{...};

Capture *everything* by reference :

lam = [&]()->void{...};

Using standard operators with std::function

```
void printOp(function<int(int, int)> op) {
        cout << "printOp: op(7, 3) = " << op(7, 3) << endl;
}
```

Can we try it with **operator+** ?

```
printOp(operator+);             // ERROR !!!
printOp(int::operator+);        // ERROR !!!
```

We cannot assign operators to **std::function** !!!
Use the **std::plus<>** wrapper :

```
printOp(plus<int>());           // Works !!!
printOp(plus<>());              // Works in C++ 14 !!!
```

Also : **minus, multiplies, negate, equal_to, less, greater_equal, logical_and, bit_xor, ...**

Using non-static class methods with std::function

```
struct Z{
        int p = 0;
        int op(int x, int y) {
                return x + y + p;
        }
};
Z z{20};
```

Can we assign to an **std::function** ?

```
printOp(z.op);              // ERROR !!!
```

Problem: class methods have an invisible first argument **this**:

```
function<int(Z*, int, int)> funny = &Z::op;    // This works !
```

Using non-static class methods with std::function

```
struct Z{
        int p = 0;
        int op(int x, int y) {
                return x + y + p;
        }
};
Z z{20};
```

With a lambda wrapper (preferred) :

```
printOp([&z](int x, int y)->int {      // Correct signature !
        return z.op(x, y);
});
```

With **std::bind** :

```
printOp(bind(&Z::op, &z, placeholders::_1, placeholders::_2));
```

mutable keyword in lambdas

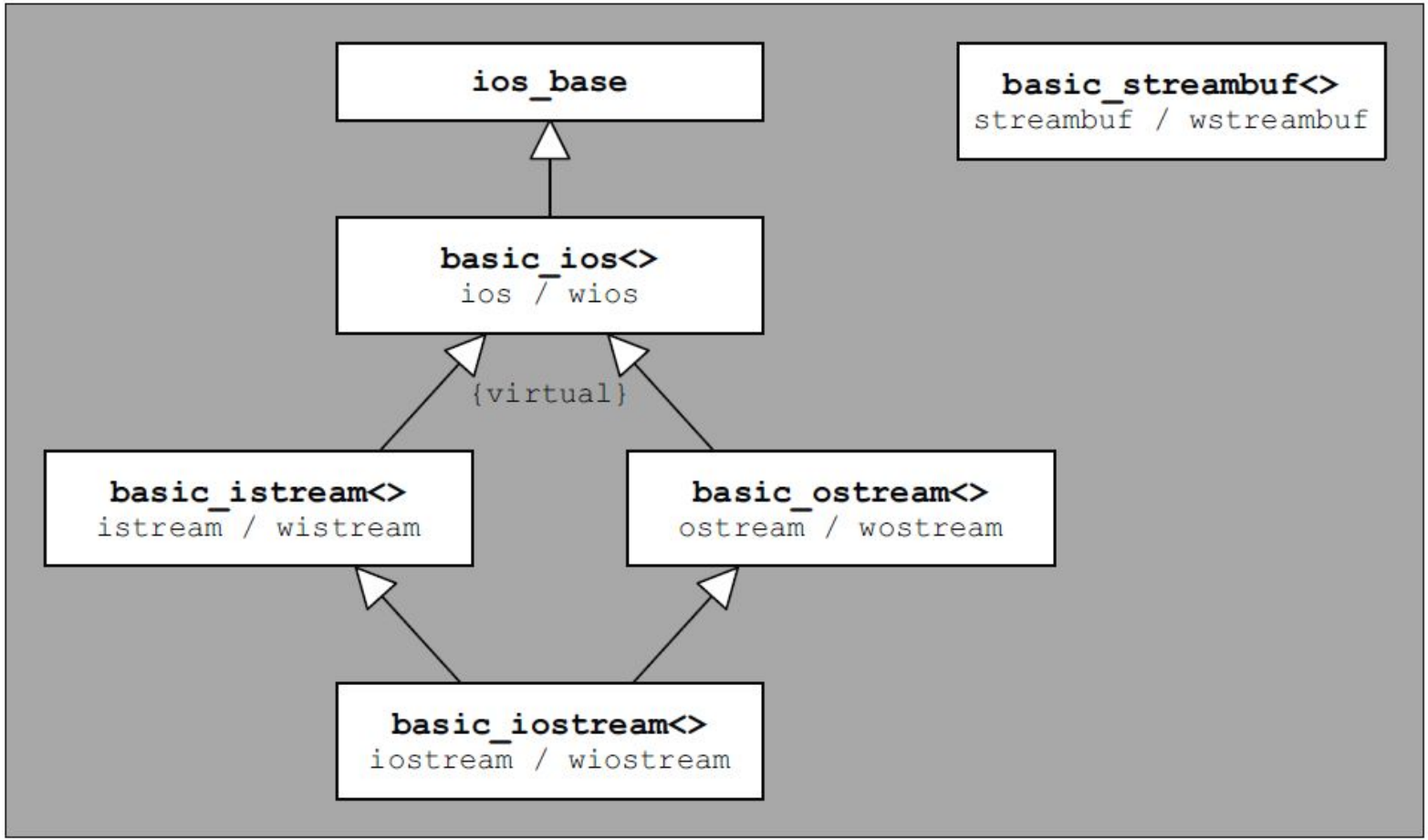
Normally variables captured by value are defined **const** :

```
int a = 10;
auto lam = [a]()->void {
        a += 5;          // ERROR !!!
        cout << a;
};
lam();
```

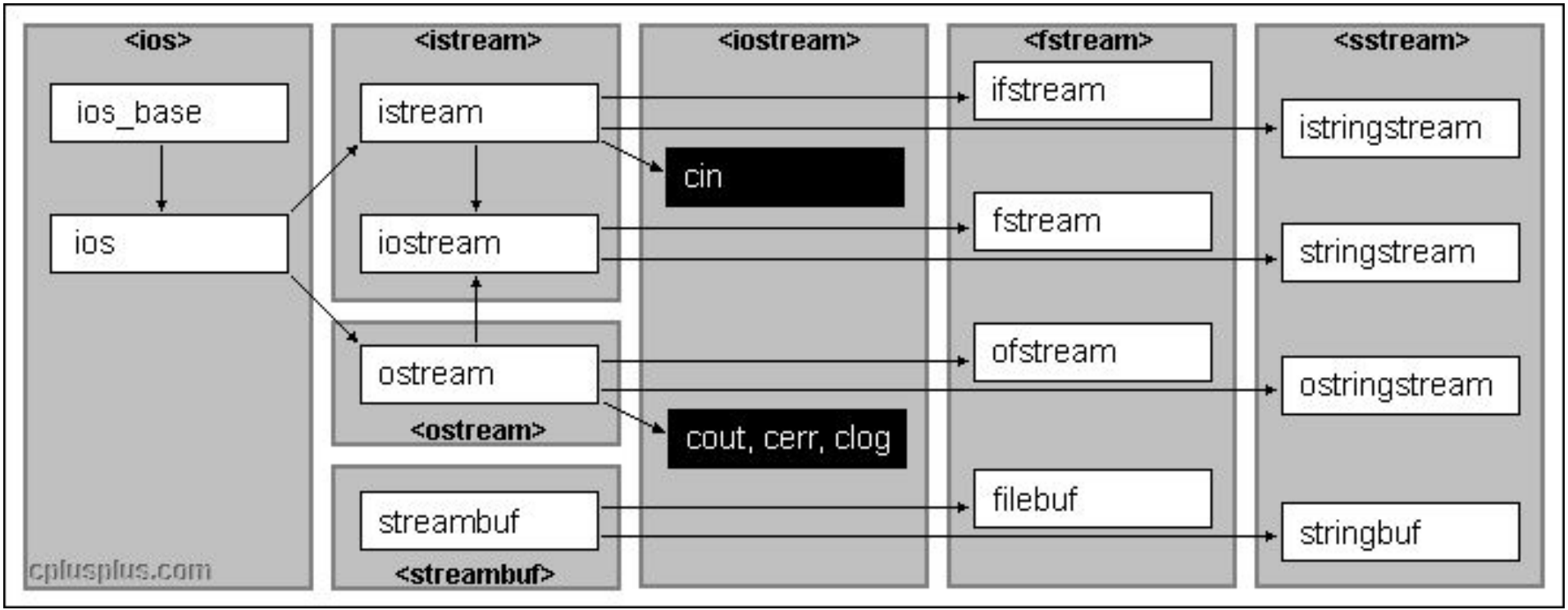
Use **mutable** modifier !

```
int a = 10;
auto lam = [a]() mutable  ->void {
        a += 5;          // OK !!!
        cout << a;
};
lam();
```

I/O stream classes (diamond problem !)



I/O stream classes and headers



C++ and unicode: Trouble with wchar_t, wstring, wcin, wcout

Idea: **wchar_t** is a wide (16 or 32 bit) character. **wstring**, **wcin**, **wcout**.
Why it is bad?

1. No guarantee it is actually UTF-16.
2. **wcin**/**wcout** depend on **locale**.

locale is compiler and OS-dependent! There is no guarantee you have utf-8 locale!

Probably works in : Linux, Windows + CL (Microsoft compiler)

MinGW gcc : only **C** and **POSIX** ! No UTF-8 !

Forget about **wchar_t**, **wstring**, **wcin**, **wcout** !

C++ and unicode : use UTF-8 ! And no locales !

1. Your code (*.h, *.cpp) must be in UTF-8 (string literals !).
2. Use **string** (not **wstring** !) for strings.
3. Use **cin**, **cout**, **ifstream**, **ofstream** with files in UTF-8.
4. Works fine with files, linux console.
5. Some trouble with windows console:
   - Output: type **chcp 65001** in the console
   - Input: I could not fix
6. Could be fixed with windows API if really needed.
7. GUI libraries have their own unicode support, e.g. **ustring** in **gtkmm**.
8. Use C++ 11 **u16string** and **char16_t** if needed. UTF8 <-> UTF16 conversion!

```
cout << "Український текст із літерами ґҐ !"  << endl;
cout << "Svenska bokstäver ÅåÖöÄä !" << endl;
cout << "Hiragana :  あ , い , う , え , お " << endl;
```

Manipulators 1 : Example 5.1

flush	Flush the buffer
endl	'**\n**' + flush the buffer
setw(i)	Set output width to **i**
left, right	Alignment left or right
setfill(c)	Set fill character
(**no**)**boolalpha**	Read/write **bool** as "**true**", "**false**" instead of **0, 1**
fixed, scientific	Notation for doubles
ws	Skip whitespaces (on read)

```
cout.precision(20);              // Set double precision
cout << setw(1) << 1 << setw(2) << 2 << setw(3) << 3 << setw(4) << 4
             << setw(5) << 5 << setw(6) << 6 << endl;
cout << left << setfill('A') << setw(10) << 1
              << right << setfill('B') << setw(10) << 2 << endl;
cout << scientific << uppercase << setfill('&') << setw(30) << 1.0/3 << endl;
```

```
1 2  3   4    5     6
1AAAAAAAAABBBBBBBBB2
&&&3.33333333333333314830E-001
```

Manipulators 2: dec, oct, hex

hex	Hexadecimal
oct	Octal
dec	Decimal
(**no**)**showbase**	Show base (e.g. 0x1a4 instead of 1a4)
(**no**)**uppercase**	Uppercase letters in hex numbers, double exp

```
int i = 45;
cout << "Dec : " << i << " " <<  showbase << i  << noshowbase << endl;
cout << "Oct : "<< oct << i <<" " << showbase << i <<dec << noshowbase<<endl;
cout << "Hex : "<< hex << i <<" " << showbase << i <<dec << noshowbase<<endl;
cout << "HEX : " << uppercase << hex << i << " "  << showbase << i  << dec
             << noshowbase << nouppercase << endl;
```

```
Dec : 45 45
Oct : 55 055
Hex : 2d 0x2d
HEX : 2D 0X2D
```

C I/O : printf(), scanf(), puts(), fgets()

C language has files and standard streams: **stdin, stdout, stderr**

```
puts("Enter a, b :");                                  // Print a string to stdout
double a, b;
scanf("%lf %lf", &a, &b);                              // Read a, b. Pointers !
getchar();                                             // Skip newline
printf(" %lf * %lf = %15.10lf\n", a, b, a*b);          // Print stuff

constexpr size_t SIZE = 100;
char s[SIZE];                                          // Buffer for a C-string
puts("Your full name ?");
fgets(s, SIZE, stdin);
printf("Hello %s !!!\n", s);
```

C files : see example ??

Using printf() formatting with C++ stream objects?

printf() is nice. Can we use it with C++ stream objects?

Simple example (a *variadic* template):

```
template <typename...  Params>
void print(std::ostream & os, const std::string & fmt, Params... p) {
    constexpr size_t SIZE = 1000;
    static char buffer[SIZE];                    // Hidden global buffer = ugly
    std::snprintf(buffer, SIZE, fmt.c_str(), p...);
    os << buffer;
}
```

Usage example:

```
print(cout, "8.1 = %10.13lf , 9 = %d \n", 8.1, 9);
```

This version:

1. Uses a global buffer of max size 1000, not thread-safe
2. Cannot work with C++ strings or any class objects

Better choice: Boost format?