

C++ Course 5: IO streams. Type casts. enum class.

By Oleksiy Grechnyev

Using C++ compilers

gcc:

g++ -o myprog main.cpp file1.cpp file2.cpp

clang:

clang -o myprog main.cpp file1.cpp file2.cpp

cl (Microsoft):

cl /o myprog main.cpp file1.cpp file2.cpp

myprog = Name of the compiled program (EXE on Windows)

main.cpp file1.cpp file2.cpp = C++ source files

How to build a CMake project ?

```
mkdir build  
cd build  
cmake ..  
cmake --build .
```

Rebuild after you have edited some source files ...

```
cmake --build .
```

Using *generators* (Example: Windows, MinGW)

```
mkdir build  
cd build  
cmake -G "MinGW Makefiles" ..  
cmake --build .
```

CMake does not call the C++ compiler directly.

Generators use low-level build systems (**make**, **nmake**, **ninja**, ...) and IDEs (Visual Studio, Code.Blocks, xcode)

Enumerated type (enum)

Enumerated types : unscoped (**enum**) and scoped (**enum class**, C++ 11)

Перечисляемые типы: **enum**, **enum class**

```
enum Color {red, green, blue, cyan, magenta, yellow, orange};
```

Variable declaration - Объявление переменной

```
Color a = magenta;           // Used without any scope ! BAD !
```

```
Color b = a;
```

```
cout << "b = " << b << endl; // Can print, prints 4
```

```
int i = a;                   // Can assign to an int variable
```

```
cout << "i = " << i << endl; // Prints 4
```

Color can be used in the **switch** statement

```
switch (b) {
```

```
case (red):
```

```
...
```

enum class: scoped version of enum

enum class is scoped. It's NOT a class ! Can be cast to **int**

```
enum class Color {red, green, blue, cyan, magenta, yellow, orange};
```

```
Color a = Color::magenta;           // magenta is in scope Color:: ! GOOD!  
Color b = a;  
cout << "b = " << (int) b << endl; // Explicit cast, prints 4  
int i = (int) a;                     // Explicit cast  
out << "i = " << i << endl;        // Prints 4
```

You can use **enum** inside **class**, (small) **namespace**, function

Можно использовать **enum** внутри класса, (малого) **namespace**, функции

Otherwise use **enum class** !

Иначе использовать **enum class** !

enum: specify numerical values

```
enum Color {red = 17, green, blue, cyan, magenta, yellow, orange};  
cout << red << " " << green << " " << blue << " " << cyan << " " << magenta <<  
      " " << yellow << " " << orange << endl;
```

```
17 18 19 20 21 22 23
```

Danger ! Опасность !

```
enum Color {red, green, blue, cyan = 1, magenta, yellow, orange};
```

```
0 1 2 1 2 3 4
```

Values are repeated !!! Величины повторяются !!!

Using **enum** for constants, anonymous unscoped **enum**

```
enum {OK = 0, FILE_NOT_FOUND = 1234, BAD_DATA = 1235, IO_ERROR = 1236};
```

```
int i = readData();
```

```
switch (i) {
```

```
...
```

Enum underlying type

Тип, в котором хранится **enum**

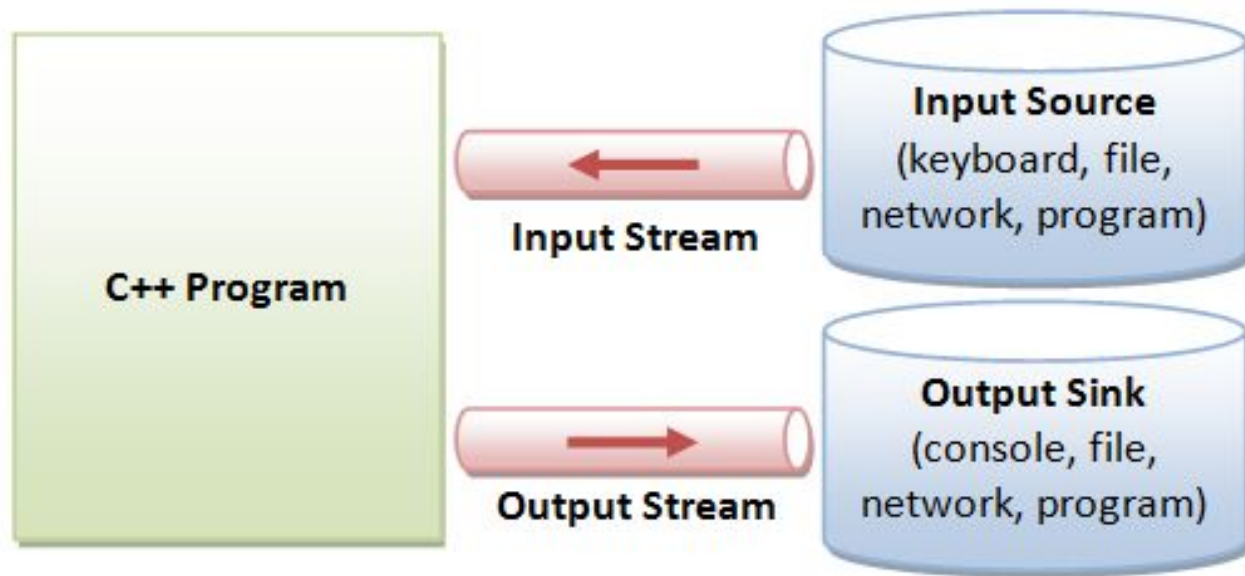
```
enum Color1 : int {red, green, blue, cyan , magenta, yellow, orange};  
enum class Color2 : unsigned char {red, green, blue, cyan , magenta, yellow, orange};  
enum class Color3 : long long {red, green, blue, cyan , magenta, yellow, orange};  
sizeof(Color1) == 4  
sizeof(Color2) == 1  
sizeof(Color3) == 8
```

The default type (тип по умолчанию) is **int** (4 bytes)

Use **unsigned char** (1 byte) to save memory !

Используйте **unsigned char** (1 байт), чтобы экономить память !

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 - стандартный ввод (buffer)

cout : Standard output - стандартный вывод (buffer)

cerr : Standard error output - стандартный вывод ошибок (no buffer)

clog : Standard error output to **cerr** with buffer ?

operator >> : Input operator - операция ввода

operator << : Output operator - операция вывода

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 (возвращают поток)

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" !!!
```

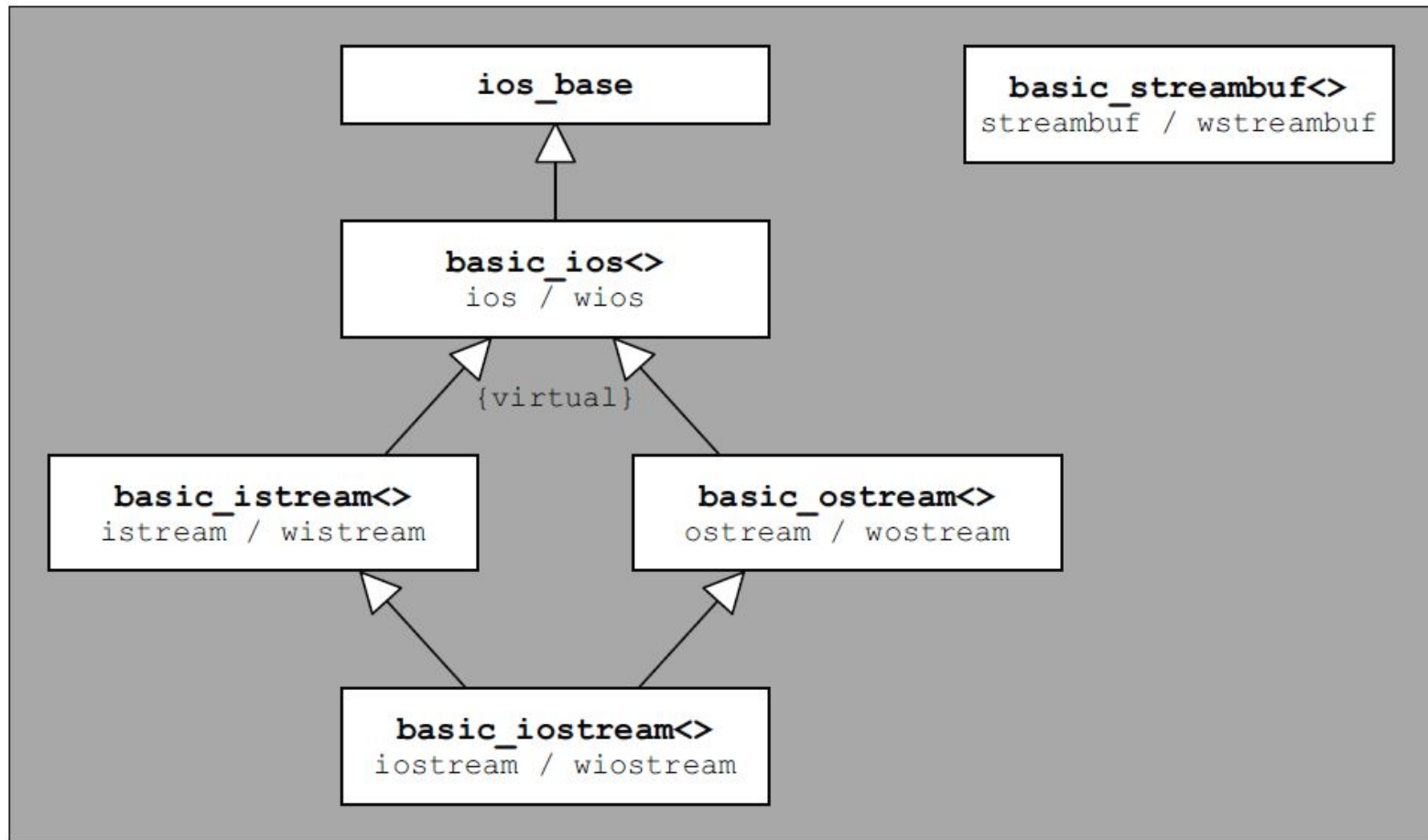
The proper way to do it :

```
getline(cin, name);
```

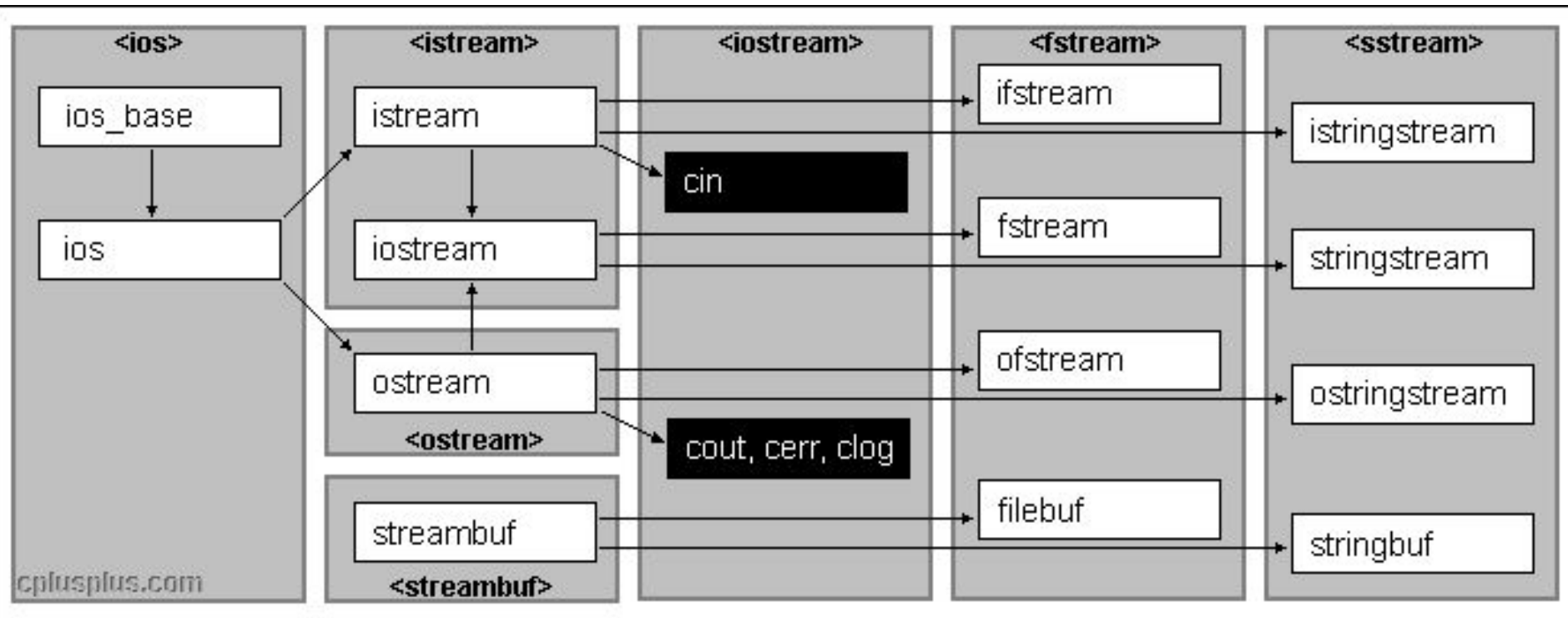
Now `name == "Sir Philip Anthony Hopkins"`

`getline()` reads until EOL (End of Line)

I/O stream classes (diamond problem !)



I/O stream classes and headers



Stream status and errors

Status bits:

good = Everything is OK

fail = Error (e.g. failed to read **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

По умолчанию **istream**, **ostream** не кидают исключения (исключения)

Turn on exceptions (включить исключения) for **cin**:

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

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

Теперь **cin** кидает **ios_base::failure** если какие-то проблемы!

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

String streams: istream, ostream, stringstream

To use strings as streams -- Использовать строки как потоки

Use **str()** (getter and setter) to access the underlying string

```
istream iss("13.98 17.32");  
ostream 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
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** :

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

Using **read**, **write** :

```
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**.

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? Почему это плохо?

1. No guarantee it is UTF-16. Никакой гарантии что это UTF-16.

2. **`wcin/wcout`** depend on **locale**. Используют **locale**.

locale is compiler and OS-dependent! **locale** зависят от ОС и компилятора!

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

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

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

Забудьте про **`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 rГ !" << 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
1AAAAAAAAABBBBBBBB2
&&&3.333333333333333314830E-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 5.2

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()**

Type conversions (type casts)

Преобразования типов (касты)

Implicit conversions. Неявные преобразования:

Primitive types, pointers to **void ***, pointer upcast, constructors, cast operators

C++-style casts. Conversion from type B to type A:

B b;

const_cast<A>(b) // Remove const from a pointer or reference

static_cast<A>(b) // Various type conversions

dynamic_cast<A>(b) // Safe polymorphic cast (pointer or reference)

reinterpret_cast<A>(b) // Reinterpret memory bytes as different type

C-style casts:

(A)b // Roughly speaking **const_cast**, **static_cast**, **reinterpret_cast**

A(b) // In that order

Implicit conversions

Неявные преобразования

A a;

B b;

b = a; // A is converted to B

myfunc(a); // Expects argument of type B

Primitive types:

float a = 777; // Possible loss of accuracy

int b = 3.5; // Loss of accuracy

char c = 1987; // Loss of higher bytes

Other types:

void * pV = &a; // Pointer to void *

string s = "Phoenix"; // Constructor : const char[8] to string

bool b(cout); // Cast operator: ostream to bool

Pointer and reference upcast:

```
struct Base{  
    virtual void print(){ cout << "Base" << endl;}  
};  
struct Derived : public Base{  
    void print() override { cout << "Derived" << endl;}  
};
```

Upcast: Derived * to Base *, Derived & to Base &

```
Derived derived;  
Base & baseR = derived;  
Base * baseP = &derived;  
baseR.print();           // Prints 'Derived' twice, polymorphism works  
baseP->print();
```

const_cast: Remove (cast away) const from ptrs, refs

Converts **const type *** to **type *** or **const type &** to **type &** :

```
int a = 17;
const int & crA = a;
int & rA = const_cast<int &>(crA);    // Remove const
rA = 20;
cout << "a = " << a << endl;        // Prints 20

double b = 1.1;
const double * cpB = &b;
double * pB = const_cast<double *>(cpB);    // Remove const
*pB = 2.2;
cout << "b = " << b << endl;        // Prints 2.2
```

static_cast: All "normal" casts

1. Implicit conversion made explicit (and remove warnings)

```
string s = static_cast<string>("Idiot !");
```

2. Enum::Class <-> int

```
int i = static_cast<int>(Num::Four);
```

3. void * to any pointer (No checks !)

4. Reference/pointer downcast: **Base *** to **Derived ***, **Base &** to **Derived &**. No checks!

Does not check that the object is of **Derived** class, unsafe !

```
Derived d;  
Base & rB = d;  
Base * pB = &d;  
Derived & rD = static_cast<Derived &>(rB);    // Downcast  
Derived * pD = static_cast<Derived *>(pB);    // No checks !  
rD.print();                                  // Prints "Derived" twice  
pD->print();
```

dynamic_cast: Pointer/reference downcast with checks

Like previous example, but with checks.

```
Derived d;  
Base & rB = d;  
Base * pB = &d;  
Derived & rD = dynamic_cast<Derived &> (rB);    // Downcast  
Derived * pD = dynamic_cast<Derived *> (pB);    // No checks !  
rD.print();                                     // Prints "Derived" twice  
pD->print();
```

throws **bad_cast** (for references) or returns **nullptr** (for pointers) if wrong type

```
Base b2;  
Base & rB2 = b2;  
Base * pB2 = &b2;  
Derived & rD2 = dynamic_cast<Derived &> (rB2);    // throws bad_cast  
Derived * pD2 = dynamic_cast<Derived *> (pB2);    // returns nullptr
```


reinterpret_cast, C-style casts

reinterpret_cast interprets memory bytes as a different type

reinterpret_cast Интерпретирует байты памяти как другой тип

```
int i = 17;
```

```
int *pl = &i;    // Pointer
```

```
long long j = reinterpret_cast<long long>(pl);
```

C-style casts:

const_cast, **static_cast** or **reinterpret_cast** in this order.

```
char c = (char) 2017;
```

```
int i = (int) 13.456789;
```

```
Derived & rD = (Derived &) rB;
```

```
Derived * pD = (Derived *) pB;
```

Thank you for your attention !

title

text