Inside C++

VIRTUAL FUNCTIONS AND COMPILER OPTIMIZATIONS

Virtual functions - these functions are used for dynamic polymorphism of functions defined in both base and derived classes, whereby the correct function is chosen based on which class is a class pointer is pointing to. Without the virtual keyword, always the base class function is chosen when both base class and derived class contain the same named functions.

Virtual keyword is written before the function prototype in the base class and override keyword is written after the function prototype in the derived classes.

Without virtual functions

```
class Animal {
  public:
    void sound() {
      cout << "Animal sound\n";
    }
};

class Dog : public Animal {
  public:
    void sound() {
      cout << "Dog barks\n";
    }
};

int main() {
  Animal* a;  // Declare a pointer to the base class (Animal)
  Dog d;  // Create an object of the derived class (Dog)
  a = &d;  // Point the base class pointer to the Dog object
  a->sound(); // Call the sound() function using the pointer. Since
sound() is not virtual, this calls Animal's version
  return 0;
}
```

With virtual functions

```
class Animal {
  public:
```

```
    virtual void sound() {
      cout << "Animal sound\n";
    }
};

class Dog : public Animal {
  public:
    void sound() override {
      cout << "Dog barks\n";
    }
};

int main() {
  Animal* a;
  Dog d;
  a = &d;
  a->sound(); // Outputs: Dog barks
  return 0;
}
```

DOWNSIDE OF VIRTUAL FUNCTIONS

Since the compiler (gcc or clang) can't determine at compile time which function to execute, since virtual functions enforce dynamic polymorphism, therefore the compilers cant apply compiler optimizations on the virtual functions leading to slower code. This leads us to two related topics -

1. HOW DOES THE COMPILER PERFORM RUNTIME RESOLUTION? - https://www.geeksforgeeks.org/cpp/virtual-function-cpp/
2. WHAT ARE THE VARIOUS COMPILER OPTIMIZATIONS? - https://queue.acm.org/detail.cfm?id=3372264
3. WHAT IS THE DIFFERENCE BETWEEN GCC AND CLANG COMPILERS? - https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378

WHAT ARE THE VARIOUS COMPILER OPTIMIZATIONS?

First things first, compiler optimizations only make sense for compiler languages and not interpreted languages like Python. Lets first make sense of the differences between a compiler and an interpreter right before we delve deep into the topic -

Differences between a compiler and an interpreter - compiler compiles the entire code into machine code in one go while interpreter converts the code one line by line. Also interpreters do not replace compilation completely. Compilers can contain interpreters for performance reasons and smaller memory footprint.

C, C++, C#, Go, Rust and Swift are a few examples of compiled programming languages. Some languages, like Java and Python, are both compiled and interpreted.

Now lets come back to the topic.

```
$ g++ /tmp/test.cc -O2 -c -S -o - -masm=intel \
    | c++filt \
    | grep -vE '\s+\.'
```

This command compiles C++ code and displays the resulting assembly code in a readable format. Let me break it down:

1. **g++ /tmp/test.cc -O2 -c -S -o -** — Compiles the C++ file with optimization level 2, stops before linking (-c), and outputs assembly code (-S) to stdout (-o -).
2. **c++filt** — Demangles C++ symbol names, converting mangled names like _ZN3std... into readable names like std::vector<int>.
3. **grep -vE '\s+\.'** — Filters out lines that start with whitespace followed by a dot (these are typically assembler directives like .section, .align, .cfi_* that clutter the output).
4. **-masm=intel** — Uses Intel syntax for assembly (rather than AT&T syntax).

All the assembly code shown here is for 64-bit x86 processors. (**we need to click on 64 bit x86 processors to know more in detail about them**). Only Clang and GCC compilers are covered but equally clever optimizations show up on visual studio and intel. (**click on gcc and clang compilers to know more in detail about them).**

1. **FUNCTION INLINING**

Compiler inlining is one such way to optimize compiler performance where the call to a function is replaced by the entire body of the function. Lets see an example -

```
int count(const vector<int> &vec)
{
   int numPassed = 0;
   for (size_t i = 0; i < vec.size(); ++i)
         {
      if (testFunc(vec[i]))
         numPassed++;
         }
   return numPassed;
}
```

(d)

If the compiler has no information about testFunc, it will generate an inner loop like (e):

```
.L4:
 mov edi, DWORD PTR [rdx+rbx*4] ; read rbx'th element of vec
                          ; (inlined vector::operator [])
 call testFunc(int)                ; call test function
 mov rdx, QWORD PTR [rbp+0]    ; reread vector base pointer
 cmp al, 1                  ; was the result of test true?
 mov rax, QWORD PTR [rbp+8]    ; reread the vector end pointer
 sbb r12d, -1                ; add 1 if true, 0 if false
 inc rbx                    ; increment loop counter
 sub rax, rdx                ; subtract end from begin...
 sar rax, 2                 ; and divide by 4 to get size()
                          ; (inlined vector::size())
```

```
cmp rbx, rax                ; does loop counter equal size()?
jb .L4                      ; loop if not
```

(e)

To understand this code, it's useful to know that a std::vector<> contains some pointers: one to the beginning of the data; one to the end of the data; and one to the end of the storage currently allocated (f). The size of the vector is not directly stored, it's implied in the difference between the begin() and end() pointers. Note that the calls to vector<>::size() and vector<>::operator[] have been inlined completely.

CONST AND CONSTEXPR

Using const in C++ provides several important benefits:

> Safety and correctness. const helps catch bugs at compile time. If you accidentally try to modify something you shouldn't, the compiler will stop you before the code even runs. This prevents entire categories of mistakes.

> Intent and documentation. When you mark something const, you're explicitly telling other developers (and your future self) what can and can't be modified. This makes code easier to understand and reason about. Someone reading const int* ptr immediately knows they can't change what ptr points to.

> Enables compiler optimizations. The compiler can make stronger assumptions about const data. Since it knows a value won't change, it can optimize more aggressively—reusing values, eliminating redundant checks, or moving computations around.

> Allows calling functions on const objects. If you have a const object, you can only call const member functions on it. This forces a clear separation between functions that modify state and those that don't. Without const functions, you couldn't use many objects as const in the first place.

> Thread safety reasoning. const helps reason about thread safety. If data is const, multiple threads can safely access it simultaneously without synchronization, since nothing is modifying it.

> API contracts. When a function takes a const reference parameter, it guarantees to callers that the function won't modify that argument. This is a contract that makes APIs clearer and more predictable.

> Prevents accidental modifications. Even in your own code, const acts as a guardrail. It's easy to accidentally modify something when you're in a rush or tired—const prevents these mistakes.

In general, using const liberally throughout your code **makes it safer, clearer, and often faster**.

Object Oriented Programming in C++ and Python