# Эффективность различных конструкций языка C++

(по мотивам книги «Optimizing software in C++» by Agner Fog)

**Различные способы хранения переменных**

Переменные и объекты хранятся в различных частях памяти в зависимости от того, как они были объявлены в программе на C++. Это имеет влияние на работу кэша данных.

*Хранение на стеке*

Переменные и объекты, объявленные внутри функции, хранятся на стеке.

Стек – это специальная область памяти, которая организована… Стек используется для хранения адресов возврата, параметров функций, локальных переменных и для сохранения значений регистров. Каждый раз, когда функция вызывается, она выделяет место на стеке для всех этих целей, а при возврате освобождает.

Переменные и объекты, размещенные рядом на стеке, как правило, эффективно используют кэш-память данных 1-го уровня (если это не большие массивы).

Можно уменьшить область видимости с помощью { }, однако большинство компиляторов память освобождают только в конце функции. Но для переиспользования регистров эту конструкцию можно использовать.

*Глобальное или статическое хранение*

Переменные, объявленные за пределами всех функций, называются глобальными. Они доступны из любой функции. Глобальные переменные расположены в статической части памяти. Статическая память также используется для переменных, объявленных с ключевым словом static, для вещественных констант, строковых констант, списков инициализации массивов, таблиц переходов оператора switch и таблиц виртуальных функций.

Статическая часть обычно разделена на три части:

- одна для констант, которые никогда не меняются;
- одна для проинициализированных переменных, значения которых могут быть изменены в программе;
- одна для неинициализированных переменных, значения которых могут быть изменены в программе.

Преимущество статической области в том, что она может быть проинициализирована до того, как программа будет запущена. Недостаток в том, что это место в памяти остается занятым на протяжении всего исполнения программы, даже если переменные используются в небольшой части программы. Это делает кэширование данных менее эффективным.

Избегайте использования глобальных переменных. Глобальные переменные можно использовать для взаимодействия между потоками, но это единственная ситуация, где их нельзя избежать. Если вы хотите обеспечить доступ нескольких функция к одной переменной, более эффективно может быть создать функции, которые обращаются к членам одного класса и хранят разделяемые переменные внутри класса.

Часто предпочтительным будет объявить таблицу поиска статической. Например:

```
float SomeFunction (int x) {
      static float list[] = {1.1, 0.3, -2.0, 4.4, 2.5};
      return list[x];
}
```

Преимущество здесь в том, что список не должен инициализироваться каждый раз при вызове функции. Если таблицу сделать нестатической, то ее значения будут каждый раз копироваться из статической памяти. Однако в особых случаях, если эти данные будут много раз использоваться в цикле, то может быть более эффективным скопировать их на стек, чтобы разместить рядом с другими данными и повысить эффективность использования кэша данных 1 уровня.

Строковые и вещественные константы хранятся в статической памяти. Например:

```
a = b * 3.5;
c = d + 3.5;
```

Большинство компиляторов распознают, что константы одинаковые, и разместят в памяти только одну копию. Все одинаковые константы объединяются, чтобы повысить эффективность использования кэш-памяти.
Целочисленные константы обычно включаются в код команды.

*Хранение на регистрах*
Все оптимизирующие компиляторы автоматически выбирают наиболее часто используемые переменные для хранения на регистрах. Один и тот же регистр может использоваться для хранения нескольких переменных, если их диапазоны использования не пересекаются.
Количество регистров ограничено.
Вещественные переменные используют различные типы регистров: FPU, XMM.
Точность вычислений может зависеть от используемых регистров (32-битные, 64-битные, 80-битные).

*Volatile*
Volatile-переменные всегда имеют то значение, которое им в последний раз было присвоено любым потоком. Поэтому компилятор не может выполнять с ними все оптимизации и всегда размещает в памяти.
Volatile не означает atomic, т.е. не защищает переменную от одновременной записи несколькими потоками.

*Хранение локальное для потока*
Большинство компиляторов могут создавать локальное хранилище потока для статических и глобальных переменных, используя ключевое слово __thread или __declspec(thread). Такие переменные имеют по одному экземпляру на каждый поток. Локальное хранилище потока неэффективно, т.к. доступ к нему осуществляется через указатель, который хранится в блоке окружения потока. По возможности, нужно избегать его использования, а использовать хранение на стеке.

*Far*
В старых ОС использовалось для доступа к большим данным. Неэффективно. Используйте другую ОС.

*Динамическое выделение памяти*
Операторы new и delete и функции malloc и free работают долго. Часть памяти, называемая кучей, зарезервирована для динамического выделения памяти. Куча легко становится фрагментированной, когда объекты разного размера постоянно выделяются и освобождаются. Менеджер кучи может проводить много времени, расчищая место, которое больше не используется, и ища свободное место. Это называется сборкой мусора. Объекты, которые выделены последовательно во времени, не всегда лежат последовательно в памяти. Они могут быть разбросаны по разным частям кучи, если она фрагментирована. Это делает кэширование неэффективным.
Динамическое выделение памяти – источник многих ошибок, например:

- утечка памяти – освобождение выделенной памяти не во всех вариантах работы программы;
- обращение к уже освобожденной памяти.

Для предотвращения таких ошибок иногда нужно усложнять логику программы, что приводит к накладным расходам.
Некоторые языки программирования, например, Java, используют динамическое выделение памяти для всех объектов. Это, конечно, неэффективно.

*Переменные, объявленные внутри класса*
Переменные, объявленные внутри класса, хранятся в том порядке, в котором они объявлены в классе. Тип хранения зависит от типа хранения объекта класса. Объект не может находиться в регистре, за исключение простейших случаев, но его члены могут быть скопированы в регистры.
Член класса с ключевым словом static будет размещен в статической области памяти и будет иметь только один экземпляр. Нестатические члены одного класса хранятся с каждым экземпляром класса.
Хранение переменных в структуре или классе – это хороший способ быть уверенным, что переменные, которые используются в одной части программы и хранятся вместе.

**Целочисленные переменные и операторы**

*Целочисленные размеры*

Существую целочисленные переменные разных размеров. Целочисленные операции выполняются быстро в большинстве случаев, независимо от размера. Но неэффективно использовать переменные размера больше, чем размер регистра (особенно если используется умножение или деление).

Обычно int – наиболее эффективный размер на данной системе. Типы меньшего размера (char, short int) немного менее эффективны. Обычно компилятор конвертирует их в int, выполняет над ними операции, а затем берет младшую часть результата. Можно считать, что это преобразование занимает 0 или 1 такт.

На 64-битных системах есть минимальное различие между эффективностью 32-битных и 64-битных целых, если не использовать деление.

Рекомендуется использовать стандартный int, если нет опасности переполнения. Если есть (например, для доступа к большому массиву), то следует использовать больший минимально возможный тип.

Битовые поля другого размера, чем 8, 16, 32 и 64 бита, менее эффективны.

Беззнаковый целочисленный тип size_t занимает 32 бита на 32-битных системах и 64 бита на 64-битных. Он используется для хранения размеров и индексов массивов, которые помещаются в память, и гарантирует, что переполнения не будет.

При рассмотрении, достаточно ли большой конкретный целочисленный тип, необходимо учитывать, что переполнение могут вызвать промежуточные вычисления. Например, в выражении `a = (b*c)/d`, переполнение может случиться при вычислении `(b*c)`, даже если a, b, c и d меньше максимального значения.

Автоматической проверки на целочисленное переполнение нет.

*Знаковые и беззнаковые целые*

В большинстве случаев нет разницы между знаковыми и беззнаковыми целыми значениями. Но есть несколько случаев, когда разница есть:

- Деление (взятие остатка от деления) на константу: беззнаковое быстрее.
- Преобразование к вещественному значению: знаковое быстрее.
- Различное поведение при переполнении.

Преобразование между знаковым и беззнаковым значениями ничего не стоит – это только смена интерпретации.

```
int a, b;
double c;
b = (unsigned int)a / 10; // Convert to unsigned for fast division
c = a * 2.5; // Use signed when converting to double
```

Нельзя сравнивать знаковые и беззнаковые значения, т.к. результат неоднозначный.

*Целочисленные операторы*

Целочисленные операторы очень быстрые. Простые операции (сложение, вычитание, сравнение, битовые операции и сдвиги) занимают один такт на большинстве микропроцессоров.

Умножение и деление занимают более долгое время. Целочисленное умножение занимает 11 тактов на Pentium4 и 3-4 такта на большинстве остальных процессоров. Целочисленное деление занимает 40-80 тактов, в зависимости от процессора. На процессорах AMD целочисленное деление тем быстрее, чем меньше размер типа данных (на Intel – нет).

*Операторы инкремента и декремента*

Операторы инкремента так же быстры, как сложение. В простом случае пре-инкремент и пост-инкремент реализуются одинаково, и обычно нет разницы. Но когда результат выражения используется, разница есть. Например, `x = array[i++]` более эффективно, чем `x = array[++i]`, т.к. во втором случае вычисление адреса должно ждать нового значения i.

Еще пример: в случае `a = ++b;` компилятор распознает, что значения a и b совпадают, и может использовать для них один регистр. В случае `a = b++;` значения переменных разные, и один регистр использовать нельзя.

Все то же самое верно и для оператора декремента.

**Вещественные переменные и операторы**

Есть два типа регистров для хранения вещественных значений: FPU (80 бит) и XMM (32/64 бита). Преимущества использования регистров FPU:

- Все вычисления выполняются с точностью long double (80 бит).
- Преобразования между различными форматами точности не занимают время.
- Существуют команды процессора для вычисления математических функций, таких как логарифмы и тригонометрические функции.
- Код более компактен и занимает меньше места в кэше команд.

Недостатки использования регистров FPU:

- Компилятору сложно создать регистровую переменную из-за того, как организован регистровый стек.
- Вещественные сравнения медленные (если только не используется набор команд Pentium-II или более позднего процессора).
- Преобразования между целочисленными и вещественными значениями неэффективны.
- Деление, квадратный корень и математические функции занимают дольше времени при использовании типа long double.

Преимущества XMM/YMM:

- Легко создавать вещественные переменные.
- Доступны векторные операции.

Недостатки XMM/YMM:

- Не поддерживается тип long double.
- Вычисление выражений, использующих операнды различной точности, требуют использования команд преобразования формата, которые могут быть очень долгими.
- Математически функции реализованы в программной библиотеке, но это часто быстрее, чем встроенные аппаратные функции.

Очень немногие компиляторы могут использовать совместно два типа вещественных операций и выбирать тип, наиболее оптимальный в конкретном случае.

В большинстве случаев операции над double занимают не больше времени, чем над float. Когда используется FPU, разницы вообще нет, а тип long double занимает немногим больше времени. Когда используются регистры XMM, операции деления, вычисления корня и математических функций быстрее на float, чем на double, тогда как сложение, вычитание, умножение и т.п. одинаково для float и double на большинстве процессоров (без использования векторизации).

В большинстве случаев можно использовать double без страха временных затрат. Однако, для больших массивов можно использовать float, который занимает меньше памяти, и больше элементов помещается в кэш. Также float лучше использовать, если используется векторизация.

**Enum**

Enum – это integer.

**Логические значения**

*Порядок логических операндов*

Логические операции в выражениях вычисляются по принципу ленивых вычислений слева направо. Соответственно, их нужно упорядочивать так, чтобы наиболее часто вычислений было меньше. Если один из операндов более хорошо предсказуем или быстрее вычисляется, поместите его в выражении первым.

Однако при смене мест операндов в логических выражениях следует опасаться побочных эффектов, когда одно из подвыражений в определенных случаях не должно вычисляться.

*Логические переменные переопределены*

Логические переменные переопределены в том смысле, что корректным значением true считается любое ненулевое целочисленное значение, а результат всегда 0 или 1. Соответственно, в логических выражениях есть проверка на любое ненулевое значение. Например:

```
bool a, b, c, d;
c = a && b;
d = a || b;
```

обычно реализуется компилятором следующим образом:

```
bool a, b, c, d;
if (a != 0) {
      if (b != 0) {
            c = 1;
      }
      else {
            goto CFALSE;
      }
}
else {
      CFALSE:
      c = 0;
}
if (a == 0) {
      if (b == 0) {
            d = 0;
      }
      else {
            goto DTRUE;
      }
}
else {
      DTRUE:
      d = 1;
}
```

Это неоптимально, т.к. есть ветвления. Можно придумать более эффективный код, если быть уверенным, что на вход придут значения только 0 или 1. Это предположение можно сделать, если значения переменных явным образом проинициализированы, или они являются результатом другого логического выражения. Оптимизированный код может выглядеть следующим образом:

```
char a = 0, b = 0, c, d;
c = a & b;
d = a | b;
```

Здесь char используется вместо bool, чтобы использовать однотактные побитовые операторы. Необходимо заметить, что здесь нельзя использовать другие значения, кроме 0 и 1. Также нельзя использовать b=~a. Вместо него нужно использовать b=a^1. Также следует иметь ввиду, что ленивое вычисление теперь не используется.

*Логические векторные операции*

Целочисленные значения могут быть использованы как булевы. Соответственно, для них можно использовать векторные побитовые операции.

**Указатели и ссылки**

*Указатели или ссылки*

*Эффективность*

Указатели могут занимать регистры, которых мало. Либо указатель лежит в памяти, и нужно тратить время на его доставание.

*Арифметика с указателями*

Указатели – целые числа. Операции с ними выполняются быстро.

Эффективнее перед обращением по указателю вычислять его значение пораньше, а не сразу перед обращением.

**Указатели на функции**

Если адрес может быть предсказан, то вызов функции по указателю занимает на несколько тактов больше времени, чем обычный вызов. Адрес предсказывается правильно, если в прошлый раз в этом же месте вызывалась та же функция. Если значение указателя сменилось, то неверное предсказание вызовет долгую задержку.

**Указатели на методы**

**Умные указатели**

**Массивы**

**Преобразования типов**

*Преобразование знаковые/беззнаковые*
*Преобразование целочисленного размера*
*Преобразование вещественной точности*
*Преобразование целочисленного значения в вещественное*
*Преобразование вещественного значения в целочисленное*
*Преобразование типа указателя*
*Переинтерпретация типа объекта*
*Const_cast*
*Static_cast*
*Reinterpret_cast*
*Dynamic_cast*
*Преобразование объектов класса*

**Ветвления и операторы switch**
Ветвления исполняются быстро, если хорошо предсказываются.
В некоторых случаях плохо предсказуемое ветвление можно заменить выбором из таблицы.
В некоторых случаях компилятор может автоматически заменить ветвление условным присваиванием.

**Циклы**

*Раскрутка циклов*

*Условие выхода из цикла*

*Копирование или очистка массива*

**Функции**

*Избегайте ненужных функций*

*Используйте встроенные функции*

*Избегайте вложенных вызовов функций в самом внутреннем цикле*

*Используйте макросы вместо функций*

*Используйте fastcall-функции*

*Делайте функции локальными*

*Используйте оптимизацию программы целиком*

*Используйте 64-битный режим*

**Параметры функций**

**Возвращаемые типы функций**

**Структуры и классы**

**Данные-члены классов**

**Методы**

**Виртуальные методы**

**Runtime type identification**

**Наследование**

**Конструкторы и деструкторы**

**Union**

**Битовые поля**

**Перегруженные функции**

**Перегруженные операторы**

**Шаблоны**

*Использование шаблонов для полиморфизма*

**Потоки**

**Исключения и обработка ошибок**

*Избегание цены обработки исключений*

*Создание кода безопасного для исключений*

**Другие случаи раскрутки стека**

**Директивы препроцессора**

**Пространства имен**