**МЦСТ**

# Устройство современных компиляторов

Лекция 3.
Использование компилятора.

*Маркин А. Л.*
alexanius@gmail.com

# Часть 1.
# Опции компилятора.

# Включение оптимизаций

По умолчанию компилятор практически не применяет оптимизации, т.е. работает в неоптимизирующем режиме. Это упрощает отладку кода, однако не позволяет задействовать весь потенциал вычислительной системы.

Включение режима оптимизации может значительно ускорить работу приложения, однако не стоит забывать про потенциальные проблемы:

- Оптимизации могут проявить ошибки в коде, которые ранее не были видны.
- Определённые оптимизации могут изменять результат, например если речь идёт про плавающие вычисления.
- Включение более агрессивных оптимизаций не всегда приводит к ускорению кода.

# Включение оптимизаций

У компиляторов есть общепринятые опции для включения линеек оптимизаций (конкретную для данного компилятора семантику следует уточнять в документации):

- `-O1` — линейка оптимизаций, уменьшающих размер исполняемого файла и ускоряющий исполнение. Немного увеличивает время компиляции.
- `-O2` — набор оптимизаций `-O1` плюс дополнительные оптимизации производительности, увеличивающие время компиляции.
- `-O3` — набор оптимизаций `-O2` плюс дополнительные агрессивные оптимизации производительности.

# Включение оптимизаций

```
gcc t.c -O1
```

По данной опции будут в том числе применены оптимизации удаления мёртвого кода, частичной подстановки функций, предсказания профиля, свёртки констант, переупорядочивания блоков кода, некоторые межпроцедурные анализы, многие другие.

# Включение оптимизаций

```
gcc t.c -O2
```

По данной опции будут добавлены в том числе оптимизации сбора общих подвыражений, девиртуализации, выравнивания данных и кода, более агрессивной подстановки функций, анализ пересечения объектов в памяти и другие.

# Включение оптимизаций

```
gcc t.c -O3
```

По данной опции будут добавлены в том числе агрессивная оптимизация циклов, автоматическая векторизация и другие.

# Включение оптимизаций

Существуют некоторые более агрессивные опции, позволяющие получить серьёзное ускорение, но при этом несущие определённые риски.

- `-ffast-math` — включает алгебраические оптимизации над операциями с плавающей точкой. Может приводить к изменению точности вещественных вычислений. Требует обязательного ознакомления с документацией по опции.
- `-fstack-arrays` — оптимизация для языка Фортран, при которой компилятор кладёт все массивы извеснтого размера и времени жизни на стек. Для её использования может потребоваться модификация предельных значений стека в окружении (команда `ulimit`).

# Включение оптимизаций

Также для каждого компилятора существуют способы включать расширенные наборы оптимизаций:

- `-Ofast` — в **gcc** набор оптимизаций `-O3` плюс, оптимизации отменяющие строгое следование стандарту, например `-ffast-math` и `-fstack-arrays`
- `-O4` — в **clang** на данный момент эквивалент `-O3`, но в будущем может измениться. В **lcc** — полный набор оптимизаций без ограничения на время компиляции.
- `-ffast` — в **lcc** дополнительные оптимизации, требующие от программы полного следования стандарту.

# Включение межмодульных оптимизаций

Классическая помодульная схема сборки приложений не позволяет компилятору производить межмодульные оптимизации. Для решения этой проблемы была придумана сборка в режиме «вся программа» и её аналог — оптимизации времени компоновки.

Классическая схема сборки

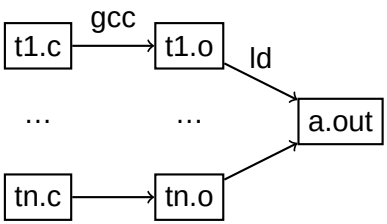
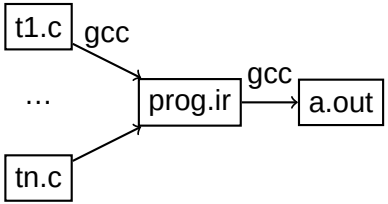



Схема сборки с оптимизациями времени компоновки



# Включение межмодульных оптимизаций

Включаются межмодульные оптимизации по опции — `-flto` для **gcc** и **clang** или `-fwhole` для **lcc**.

Данная технология соединяет все модули программы в один большой модуль, в котором видны все функции. Это позволяет выполнять межпроцедурные оптимизации для функций, которые не были возможны в обычном режиме

# Использование профиля программы

Граф потока управления содержит дополнительную информацию о профиле для более точной работы оптимизатора.

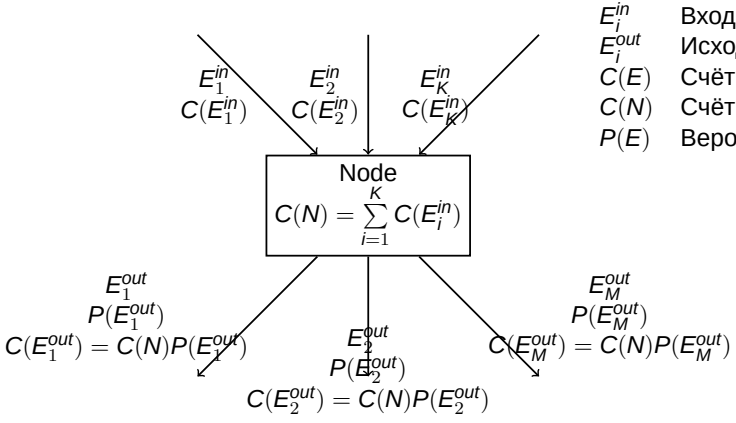
**Профиль программы** — информация о количестве проходов по данной дуге графа потока управления, а так же вероятность перехода на неё.

**Счётчики исполнения** — значения, указывающие на то сколько раз, программа исполнит данный линейный участок, или перейдёт по данной дуге.

**Вероятность перехода** — значения, указывающие на то с какой вероятностью будет совершён переход из данного линейного участка по данной дуге.

# Использование профиля программы

В представлении профиль выглядит следующим образом:




| | |
|---|---|
| $E_i^{in}$ | Входящая дуга |
| $E_i^{out}$ | Исходящая дуга |
| $C(E)$ | Счётчик дуги |
| $C(N)$ | Счётчик узла |
| $P(E)$ | Вероятность перехода на дугу |

# Использование профиля программы

Проблемой использования профильной информации в компиляторе является её отсутствие.

Компилятор умеет предсказывать профиль программы и оптимизировать на основе предсказанного профиля, однако такой механизм является неточным.

Для получения точной информации о вероятных маршрутах исполнения существует схема профилирующей компиляции.

# Использование профиля программы

Профилирующая компиляция требует двух запусков компилятора:

- `-fprofile-generate` — создаёт исполняемый файл со специальным инструментированием. После сборки программа запускается на наборе тестовых данных, которые записываются на диск.
- `-fprofile-use` — повторная компиляция с этой опцией использует полученные профильные данные и позволяет оптимизатору лучше ориентироваться в рассматриваемых участках кода.

# Использование профиля программы

Сборка с профилем имеет свои особенности:

- Если набор тестовых данных инициирует поведение отличное от поведения на боевых данных, то можно получить серьёзные деградации производительности.
- Профильные данные привязываются к графу потока управления программы, а это значит что любое изменение кода требует переполучения профиля. Если программа поменялась, то профиль примениться не сможет.

# Надёжность приложения

Любая сложная программа содержит ошибки. Исследование 2012 года показало что в среднем в каждой 1000 строк кода содержится хотя бы одна ошибка.

Стоимость даже одной ошибки может исчисляться миллиардами рублей или даже человеческими жизнями. Применение оптимизаций может проявлять ошибки, которые не были заметны ранее.

Поэтому ошибки следует находить и исправлять как можно раньше. Для этого существует богатый набор инструментов и подходов к разработке надёжного ПО.

# Диагностики времени компиляции

Профилактику ошибок следует начинать с аккуратного кода и широкого набора диагностик компилятора:

- `-std=<version>` — опция, проверяющая соответствие программы стандарту. Аргумент `<version>` может принимать, например, значение `c18` или `c++20`.
- `-Wall` — включение большого количества предупреждений (не всех), сигнализирующих о потенциальных проблемах в коде.
- `-Werror` — опция, превращающая предупреждение компилятора в ошибку. Необходима для недопустимости игнорирования проблем, выявленных при помощи `-Wall`.

# Диагностики времени исполнения

Далеко не все ошибки можно выявить на этапе компиляции. Поэтому был создан механизм выявления ошибок во время исполнения — **санитайзеры**.

- `-fsanitize=address` — анализ выходов за границы массивов и использования памяти после удаления.
- `-fsanitize=undefined` — анализ ситуаций, приводящих к неопределённому поведению.
- `-fsanitize=leak` — анализ утечек памяти.
- `-fsanitize=thread` — анализ состояния гонки в многопоточных приложениях.

Эти и некоторые другие санитайзеры можно включить по опции `-fsanitize=all`. Существуют и другие санитайзеры, о них можно почитать в документации к компилятору.

# Система тестирования

При разработке качественных программ для промышленного применения следует серьёзно подходить к тестированию надёжности и производительности ПО.

- Необходимо иметь набор данных, обеспечивающих максимально возможное покрытие кода. Код который не покрыт тестами можно считать нерабочим.
- Если для приложения подразумевается оптимизирующая компиляция, то эталонные результаты следует получать неоптимизированной версией и проверять что оптимизированная версия даёт эквивалентный результат.
- Следует проверять что оптимизации не ухудшают производительность, т.к. в ряде случаев такой эффект может наблюдаться.

# Пример итоговых опций сборки

В качестве примера можно рассмотреть несколько наборов опций, применимых для разработки приложения (опции для компилятора `gcc`):

- `-Wall -Werror -std=c++20 -Ofast -flto -fsanitize=all` — для поиска ошибок
- `-Wall -Werror -std=c++20 -Ofast -flto -fprofile-generate` — для получения профиля
- `-Wall -Werror -std=c++20 -Ofast -flto -fprofile-use` — для итоговой сборки приложения

Часть 2.
Подсказки компилятору.

# Расширения компиляторов

Языки программирования обычно имеют мало выразительных средств для передачи компилятору информации о программе. Поэтому разработчики компиляторов вводят свои расширения для удобства программистов.

Расширения компиляторов могут значительно упростить и улучшить жизнь разработчиков, однако следует помнить что все они зависят от конкретного компилятора.

Использовать расширения нужно таким образом чтобы была возможность собирать программу любым компилятором, поддерживающим классический стандарт языка.

Часть 2.
Секция 1. Атрибуты.

# Атрибуты

Атрибуты — это специальное расширение синтаксиса, которое позволяет указывать определённые свойства функций и объектов программы. Свойства могут быть как универсальными, так и платформо-специфичными.

Пример использования атрибута:

```
void fatal () __attribute__ ((noreturn));

void
fatal (/* ... */)
{
  /* ... */ /* Print error message. */ /* ... */
  exit (1);
}
```

# Атрибуты

Для оптимизации компилятору полезно знать какие функции будут вызываться часто, а какие — редко. Это поможет как в оптимизации самих функций, так и точек их вызова. Для этого существует два атрибута:

- `hot` — атрибут, показывающий что функция является «горячей», т.е. будет вызываться часто.
- `cold` — атрибут, показывающий что функция является «холодной», т.е. будет вызываться редко.

Функции, помеченные этими атрибутами помимо специфичных линеек оптимизаций даже складываются в отдельные подсекции исполняемого файла с целью увеличить их локальность.

# Атрибуты

Существуют атрибуты, помогающие в написании собственных выделителей памяти и функций копирования:

- `malloc` — говорит компилятору что функция является аналогом функции `malloc`, т.е. возвращает указатель на участок действительной памяти, который не пересекается с ранее выделенными или в редких случаях `NULL`.
- `returns_nonnull` — говорит компилятору что функция никогда не вернёт `NULL`.
- `nonnull` — говорит компилятору что аргументы с указанным в атрибуте номером никогда не окажутся нулевыми указателями.

Также существуют дополнительные атрибуты `alloc_size` и `alloc_align`, но мы не будем их здесь рассматривать.

# Атрибуты

Некоторые атрибуты бывают полезны для упрощения инициализации и очистки глобальных ресурсов программы:

- `constructor` — функции с данным атрибутом начинают вызываться до запуска функции `main`.
- `destructor` — функции с данным атрибутом будут вызываться после завершения функции `main` или при вызове функции `exit`.

Обоим атрибутам можно задавать аргумент, обозначающий приоритет вызова данной функции.

# Атрибуты

Бывает полезно пометить свойства функции, отвечающие за возврат управления и возможность работы с исключениями:

- `noreturn` — атрибут показывает что функция не вернёт управление вызывающей функции. Примерами таких функций могут быть `exit` и `abort`.
- `nothrow` — даёт компилятору обещание что функция не создаёт исключение. Она актуальна для функций языка Си, работающих в окружении C++.

# Атрибуты

Компилятору очень помогает указать отсутствие побочных эффектов в функциях:

- `const` — сообщает компилятору что функция не меняет состояние программы, а её выходное значение зависит только от аргументов и для одних и тех же аргументов оно будет одинаковым в любой точке вызова программы.
- `pure` — также сообщает компилятору об отсутствии побочных эффектов в функции, однако позволяет предоставлять ей в качестве аргументов указатели на области памяти, которые могут меняться между вызовами.

Данные атрибуты подразумевают довольно агрессивную оптимизацию вызовов, поэтому некорректно выставленный атрибут может привести к ошибке при исполнении.

# Атрибуты

В некоторых ситуациях бывает полезно указывать какие оптимизации можно или нельзя применять к данной функции:

- `noinline` — запрещает подставлять данную функцию куда-либо.
- `optimize` — позволяет задать конкретную линейку оптимизаций для данной функции. Этот атрибут имеет приоритет над линейкой, заданной в опциях запуска компилятора.

Существует атрибут `always_inline`, который заставляет компилятор выполнять оптимизацию подстановки функции. Его использование крайне **не рекомендуется**, т.к. обычно компилятор лучше знает что ему подставлять. Неаккуратное влияние на подстановку может привести к ухудшению производительности.

# Атрибуты

Современные компиляторы считают потенциально опасной ситуацию когда одна из ветвей конструкции `case` не заканчивается ключевым словом `break` и неявно переходит на следующую ветку. Поэтому был введён атрибут `fallthrough`, подсказывающий что это не ошибка:

```
switch (cond)
{
  case 1:
    bar (1);
    __attribute__((fallthrough));
  case 2:
  ...
}
```

Часть 2.
Секция 2. Встроенные функции.

# Встроенные функции

**Встроенные функции** (builtin) — это специальные функции компилятора, дающие дополнительные возможности программисту. Как и атрибуты, эти функции являются расширением компилятора и при их использовании следует помнить о портируемости.

# Встроенные функции

Сборка программы с профилем часто бывает проблематичной, поэтому в компиляторах появилась возможность явно указывать вероятности переходов по ветвлениям:

- `__builtin_expect(long exp, long c)` — говорит что с большой вероятностью `exp == c`.
- `__builtin_expect_with_probability(long exp, long c, double probability)` — более точная версия, позволяющая явно указать вероятность того что `exp == c`.

# Встроенные функции

Переход на редко вызываемую функцию:

```
if (__builtin_expect (x, 0))
  foo ();
```

Ещё один способ показать что чаще всего `ptr != NULL`:

```
if (__builtin_expect (ptr != NULL, 1))
  foo (*ptr);
```

В ядре для удобства используются макросы:

```
#define likely(x) __builtin_expect((x),1)

#define unlikely(x) __builtin_expect((x),0)
```

# Встроенные функции

Постепенно в такие возможность становятся частью языка. Например, в C++20 добавлен специальный синтаксис атрибутов, в котором есть атрибуты `[[likely]]` и `[[unlikely]]`:

```
void g(int);
  int f(int n) {
  if (n > 5) [[unlikely]] {
    g(0);
  return n * 2 + 1;
}
```

# Встроенные функции

Постепенно такие возможности становятся частью языка. Например, в C++20 добавлен специальный синтаксис атрибутов, в котором есть атрибуты `[[likely]]` и `[[unlikely]]`:

```
void g(int);

int f(int n) {
  if (n > 5) [[unlikely]] {
    g(0);

  return n * 2 + 1;
}
```

# Встроенные функции

Для минимизации времени ожидания данных в случае кеш-промаха можно указать компилятору какие данные следует заранее подкачать в кеш. Для этого существует функция предподкачки `__builtin_prefetch (const void *addr, ...)`:

```
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
  }
```

# Встроенные функции

У функции предподкачки есть два опциональных аргумента:

- `rw` — может принимать значения 0 или 1. 0 (по умолчанию) показывает что подкачиваемое значение используется для чтения, а 1 — что для записи.
- `locality` — принимает значения от 0 до 3 и указывает на временную локальность подгружаемого значения. 0 говорит о том что значение временная локальность у значения отсутствует, и его можно удалить из кеша после использования. 3 означает что значение имеет высокую временную локальность, и его следует сохранять на всех уровнях кеша. Значения 1 и 2 являются градациями временной локальности.

# Встроенные функции

Для упрощения автоматической векторизации можно подсказать компилятору величину выравнивания данных при помощи функции `__builtin_assume_aligned`:

```
void * x = __builtin_assume_aligned (arg, 16);
```

Первым аргументом функции будет указатель, который мы «выравниваем», а вторым аргументом — величина самого выравнивания. Вернёт эта функция указатель, который компилятором будет считаться выравненным.

Часть 2.
Секция 3. Ключевые слова.

# Ключевые слова

В языке Си и ранее можно было давать подсказки компилятору. Например, есть старые ключевые слова, описанные стандартом:

- `inline` — указывает компилятору что для данной функции имеет смысл выполнить подстановку.
- `register` — указывает компилятору что данную переменную имеет смысл поместить на регистр.

К счастью эти ключевые слова игнорируются современными компиляторами, т.к. компиляторы лучше понимают в каких случаях следует делать распределение на регистры и в каких случаях подставлять функцию.

# Ключевые слова

В некоторых случаях программисту требуется указать компилятору что значение объекта может быть изменено неявным образом (например в соседнем потоке). Для этого существует ключевое слово `volatile`:

```
volatile int a;
```

# Ключевые слова

Работа с указателями затрудняет качественное планирование и оптимизацию кода. Рассмотрим пример:

```
void                          0: t1  <- [val]
updatePtrs( int * a,          0: t2  <- [a]
            int * b,          3: t3  <- t1 + t2
            int * val )       4: [a] <- t3
{                             4: t4  <- [val]
   *a += *val;                4: t5  <- [b]
   *b += *val;                7: t6  <- t5 + t6
}                             8: [b] <- t6
```

Мы не знаем будут ли объекты a и b указывать на одну область памяти или на разные, т.е. будут они пересекаться или не будут.

# Ключевые слова

Программист может «пообещать» компилятору что указатели не имеют пересечений при помощи ключевого слова `restrict`:

```
void
updatePtrs(int * restrict a,
           int * restrict b,
           int * restrict val)
{
   *a += *val;
   *b += *val;
}
```

```
0: t1  <- [val]
0: t2  <- [a]
0: t5  <- [b]
3: t3  <- t1 + t2
3: t6  <- t1 + t5
4: [a] <- t3
4: [b] <- t6
```

Часть 2.
Секция 3. Указания.

# Указания

В компиляторах существует механизм «указаний» — `#pragma`. Данный механизм тоже обеспечивает подсказки компилятору.

Хотя документация gcc не рекомендует использовать данный механизм, некоторые указания не имеют аналогов среди более новых механизмов подсказок.

# Указания

В некоторых кодах до сих пор можно встретить попытки применения раскрутки цикла самим пользователем. Обычно это приводит к более медленному выполнению по сравнению с автоматической раскруткой. Более того, на некоторых платформах ручная раскрутка может ухудшать производительность.

Хотя документация gcc не рекомендует использовать данный механизм, некоторые указания не имеют аналогов среди более новых механизмов подсказок.

# Указания

Если программист всё-таки считает что его код следует раскрутить с определённым фактором раскрутки, то он может дать компилятору указание сделать это:

```
void foo (int n, int *a, int *b, int *c)
{
  int i, j;
#pragma GCC unroll 4
  for (i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
}
```

# Указания

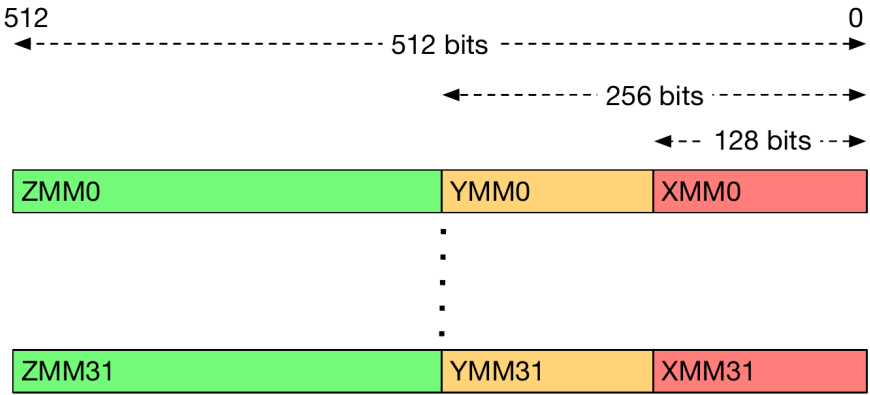
Иногда бывает полезно указать компилятору на отсутствие межитерационных зависимостей для потенциально векторизуемых операций. Тогда компилятор сможет провести безусловную векторизацию:

```
void foo (int n, int *a, int *b, int *c)
{
  int i, j;
#pragma GCC ivdep
  for (i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
}
```

# Часть 3.
# Векторизация.

# Векторизация

Современные процессоры помимо основных инструкций обычно обладают ещё и векторными расширениями, позволяющими реализовывать параллелизм уровня данных (SIMD — Single Instruction Multiple Data).



# Векторизация

В **gcc** автоматическая векторизация включается в линейке `-O3`, однако бывает полезно изучить отчёт векторизатора чтобы понять какие циклы он смог соптимизировать, а какие — нет:

```
$ gcc t.c -O3 -fopt-info-vec -fopt-info-vec-missed
...
linpack.c:858:9: optimized: loop vectorized using 16 byte vectors
linpack.c:858:9: optimized:  loop versioned for vectorization bec
...
```

# Векторизация

Компилятору можно задавать набор векторных инструкций, который ему следует пытаться применить, например:

- `-msse`
- `-msse4`
- `-mavx`
- `-mavx2`

При этом не стоит забывать проверять что векторизация действительно применилась к интересующему нас циклу.

# Векторизация

Автоматическая векторизация требует чтобы циклы были как можно более простыми.

На практике если в приложении существенную роль в производительности играет векторизация, лучше всего полагаться не на компилятор, а самостоятельно организовывать векторизуемые типы данных при помощи расширений компилятора.

Заключение.

# Заключение

Современные компиляторы обладают богатым набором оптимизаций и диагностик, доступным пользователю.

Включение большого количества агрессивных оптимизаций требует качественного тестирования приложения как на надёжность, так и на производительность.

Для качественной оптимизации компилятору нужна либо информация об исполнении программы, либо хорошие подсказки. Лучше всего когда есть и то и другое.