Ravesli Ravesli

- Уроки по C++
- OpenGL
- SFML
- Qt5
- RegExp
- Ассемблер
- Купить .PDF

Урок №101. Встроенные функции

Юрий |

- Уроки C++

|

Обновл. 24 Ноя 2020 |

30150

| 7

Использование функций имеет много преимуществ, в том числе:

➔ Код, находящийся внутри функции, может быть повторно использован.

➔ Гораздо проще изменить или обновить код в функции (что делается один раз), нежели искать и изменять все части кода в функции main() «на месте». Дублирование кода — хороший рецепт для ошибок и ухудшения производительности.

➔ Упрощение чтения и понимания кода, так как вам не нужно знать реализацию функции, чтобы её использовать (предполагается наличие информативного названия функции и **комментариев**).

➔ В функциях поддерживается проверка типов данных для гарантии того, что передаваемые аргументы соответствуют параметрам функции.

➔ Функции упрощают **отладку вашей программы**.

Однако, одним из главных недостатков использования функций является то, что при каждом её вызове происходит расход ресурсов, что влияет на производительность программы. Это связано с тем, что ЦП должен хранить адрес текущей команды (инструкции или стейтмента), которую он выполняет (чтобы знать, куда нужно будет вернуться позже), вместе с другими данными. Затем точка выполнения перемещается в другое место программы. Дальше все параметры функции должны быть созданы и им должны быть присвоены значения. И только потом, после выполнения функции, точка выполнения возвращается обратно. Код, написанный «на месте», выполняется значительно быстрее.

Для функций, которые являются большими и/или выполняют сложные задачи, расходы на вызов обычно незначительны по сравнению с количеством времени, которое отводится на выполнение кода этой

функции. Однако для небольших, часто используемых функций, время, необходимое для выполнения вызова, часто превышает время, необходимое для фактического выполнения кода этой функции. А это, в свою очередь, может привести к существенному снижению производительности.

Язык C++ предлагает возможность совместить все преимущества функций вместе с высокой производительностью кода, написанного «на месте». Речь идет о встроенных функциях. **Ключевое слово inline** используется для запроса, чтобы компилятор рассматривал вашу функцию как встроенную. При компиляции вашего кода, все **встроенные функции** (англ. *«inline functions»*) раскрываются «на месте», то есть вызов функции заменяется копией содержимого самой функции, и ресурсы, которые могли бы быть потрачены на вызов этой функции, сохраняются! Минусом является лишь увеличение компилируемого кода за счет того, что встроенная функция раскрывается в коде при каждом вызове (особенно если она длинная и/или её вызывают много раз). Рассмотрим следующий фрагмент кода:

```
#include <iostream>

int max(int a, int b)
{
    return a < b ? b : a;
}

int main()
{
    std::cout << max(7, 8) << '\n';
    std::cout << max(5, 4) << '\n';
    return 0;
}
```

Эта программа дважды вызывает функцию max(), т.е. дважды расходуются ресурсы на вызов функции. Поскольку max() является довольно таки короткой функцией, то это идеальный вариант для её конвертации во встроенную функцию:

```
inline int max(int a, int b)
{
    return a < b ? b : a;
}
```

Теперь, при компиляции функции main(), ЦП будет читать код следующим образом:

```
int main()
{
    std::cout << (7 < 8 ? 8 : 7) << '\n';
    std::cout << (5 < 4 ? 4 : 5) << '\n';
    return 0;
}
```

Такой код выполнится быстрее ценой несколько увеличенного объема.

Из-за возможности подобного «раздувания», встроенные функции лучше всего использовать только для коротких функций (не более нескольких строк), которые обычно вызываются внутри циклов и не имеют **ветвлений**. Также, обратите внимание, ключевое слово inline является лишь рекомендацией —

компилятор может игнорировать ваш запрос на встроенную функцию. Подобное произойдет, если вы попытаетесь сделать встроенной длинную функцию!

Наконец, современные компиляторы автоматически конвертируют соответствующие функции во встроенные — этот процесс автоматизирован настолько, что даже лучше ручной выборочной конвертации, проведенной программистом. Даже если вы не пометите функцию как встроенную, компилятор автоматически выполнит её как таковую, если посчитает, что это способствует улучшению производительности. Таким образом, в большинстве случаев нет особой необходимости использовать ключевое слово inline. Компилятор всё сделает сам.

Правило: Если вы используете современный компилятор, то нет необходимости использовать ключевое слово inline.

Встроенные функции освобождаются от «правила одного определения»

На предыдущих уроках мы не раз говорили, что вы не должны определять функции в **заголовочных файлах**, так как если вы подключаете один заголовок с определением функции в несколько файлов .cpp, то определение функции также будет скопировано несколько раз. Затем, на этапе линкинга, линкер выдаст ошибку, что вы определяете одну и ту же функцию больше одного раза.

Однако встроенные функции освобождаются от этого правила, так как дублирования в исходном коде не происходит — определение функции одно, и никакого конфликта при соединении линкером файлов .cpp возникнуть не должно.

Сейчас это всё может показаться неинтересными пустяками, но чуть позже, когда мы будем рассматривать новый тип функций (дружественные функции), эти *пустяки* пригодятся. И помните, что даже с использованием встроенных функций, вы НЕ должны определять глобальные функции в заголовочных файлах.

Урок №100. Возврат значений по ссылке, по адресу и по значению

Урок №102. Перегрузка функций

Комментариев: 7

1.  *Олег*:
[7 июля 2020 в 20:31](#)

Если я правильно понял, то функция , это адрес кода, на который переходит прога при ее вызове.. Насколько я помню ассемблер из времен палеозоя, то существовала инструкция JMP с адресом строки выполнения участка кода. Не могу понять, почему эти переходы так потребляют ресурсы процессора. Почему нужно при каждом вызове инициализировать сегмент данных для кода функции, когда он может быть выделен сразу во время компиляции??? Почему множество вызовов одной функции увеличивает выполняемый код???

[Ответить](#)

1.  *Максим*:
[15 августа 2020 в 22:15](#)

Отмечу, что при ответе на вопросы не знаю точных ответов, так что пользуюсь логикой и тем, что знаю.

Первый вопрос, про данные:
Предполагаю, что свою роль в увеличенном потреблении ресурсов процессора играют данные (параметры функции), которые необходимо передавать вместе с функцией (при передаче НЕ по ссылке данные копируются, а для этого необходимо выделить место для данных, перенести данные, а затем при выходе из функции освободить место).

На второй вопрос, про сегмент данных, не рискну отвечать.

А по поводу третьего вопроса, про то, почему множество вызовов одной функции увеличивают выполняемый код:
Полагаю, вы имеете ввиду inline функцию, в таком случае код увеличивается, потому вместо КАЖДОГО вызова функции подставляется код.
В статье в месте каждого вызова функции max вместо max(x, y) подставляется (x < y ? y : x).
То есть вместо

```
1 max(x, y)
2 max(x, y)
```

будет

```
1 (x < y ? y : x)
2 (x < y ? y : x)
```

И, разумеется, в скомпилированном файле объём кода будет больше, так как вместо двух jmpTo и одного куска кода будет просто два куска кода.
А два куска кода больше (в крайнем случае равно), чем два jmpTo и один кусок кода.

[Ответить](#)

2.  *Woland*:
[15 ноября 2019 в 22:32](#)

Интересно, в классе кратким версиям перегруженных функций ссылающимся на полную версию функции лучше сразу задавать inline?
Посмотрел правда, VS 2019 не меняет размер экзешника ни Debug ни Release.

[Ответить](#)

3.  *Константин*:
[1 сентября 2019 в 20:54](#)

А если определения, что ценой увеличения объёма кода можно повысить производительность не достаточно. Существует "вымя" за которое можно потрогать и понять что это действительно так, кроме экспериментов с кодом? На других системах ситуация может поменяться? (на каком-нибудь другом процессоре или в случае программирования микропроцессоров?)

[Ответить](#)

1.  *Steindvart*:
[10 мая 2020 в 17:49](#)

"Скорость" железа является одной из переменных, из которых рассчитывается общая скорость исполнения программы в единицах времени.

Например, представьте, что у вас `процессор_1` выполняет 100 инструкций кода в секунду, а `процессор_2` 1000 инструкций в секунду. У нас есть программа, которая состоит из 1000 инструкций.
Получается, что *одна и та же программа на втором процессоре исполнится в 10 раз быстрее, чем на первом*. Хотя это один и тот же код.

А теперь представьте, что благодаря той или иной оптимизации (например, тот же inline) ваша программа состоит из 500 инструкций, которые осуществляют точно такой же функционал. *Вне зависимости от процессора, скорость её выполнения выросла в два раза*.

Чтобы "пощупать" эти оптимизации попробуйте использовать `бенчмарки` (измерители эффективности кода) и сравните разницу по тому или иному подходу.

[Ответить](#)

4.  *Заир Узакович Узаков*:
[17 марта 2019 в 11:06](#)

Большое спасибо за статью.
З.Узаков,
Узбекистан

Ответить

1. *Юрий*:
   17 марта 2019 в 14:37

   Пожалуйста 🙂

   Ответить

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *

Email *

Комментарий

Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию

Получать уведомления о новых комментариях по электронной почте. Вы можете подписаться без комментирования.

Отправить комментарий

TELEGRAM КАНАЛ

ПАБЛИК

ТОП СТАТЬИ

- Словарь программиста. Сленг, который должен знать каждый кодер
- Урок №1. Введение в программирование
- 70+ бесплатных ресурсов для изучения программирования
- Урок №1: Введение в создание игры «SameGame» на C++/MFC
- Урок №4. Установка IDE (Интегрированной Среды Разработки)

- Ravesli
- - О проекте/Контакты -
- - Пользовательское Соглашение -
- - Все статьи -
-