Информатика: семестр 3

Программирование на C++

Конспекты лекций

Лектор: Хирьянов Т.Ф.

24 октября 2017 г.

Оглавление

Лекция 8

Парадигмы программирования

8.1 Структурное программирование

Структурное программирование строится на следующей идее: разбивать сложную задачу на простые подзадачи. Достигается это путем решения подзадач в функциях. Таким подходом мы уже часто пользовались, но у него есть минус: со временем функций накапливается много и код становится менее читабельным.

8.2 Модульное программирование

В качестве решения указанной проблемы можно функции собрать в отдельные *модули*. Полученная парадигма носит название **модульного программирования**. Разумеется, при таком подход будет главный модуль – к примеру, main.cpp. Главный модуль имеет одну особенность: в main.cpp есть функция **main**, которая содержит вход в программу. Таким образом достигается иерархия модулей.

При построении взаимоотношений модулей очень важно правильно выстроить их зависимости. Нельзя допускать т.н. "межмодульную рекурсию": когда модуль **module1** инклюдит модуль **module2**, а модуль **module2** в то же время инклюдит модуль **module1**. Использование модулей заставляет думать об их организации.

Стоит понимать, что все парадигмы нужны лишь для удобной воспринимаемости кода, но сами по себе они никак не ускоряют работы программ (а иногда даже могут и замедлить).

8.3 Раздельная компиляция

Начнем с примера. Напишем структуру для дроби с возможностью использовать комплексные числа. Будем придерживаться следующей структуры:

1. **main.cpp** – единица трансляции (то есть, компилируется). Подключает все хидеры. Собирается в **main.o**
2. **fractions.hpp** – хидер для **fractions.cpp**
3. **fractions.cpp** – единица трансляции (то есть, компилируется). Вместе со своим хидером собирается в **fractions.o**
4. **frac_complex.hpp** – хидер для **frac_complex.cpp**
5. **frac_complex.cpp** – единица трансляции (то есть, компилируется). Подключает свой хидер **и** хидер **fractions.hpp** (потому что мы наверняка будем использовать его функции). При сборке соберется в **frac_complex.o**.

Код всего проекта можно смотреть здесь.

Помимо прочего, наш проект может иметь несколько *целей*: например, мы захотим собрать готовую программу или же тестовую; подключить файл-тестировщик или нет. Для этого применяется Makefile и система сборки **make**, однако она не будет обсуждаться в этой лекции. Если вас это заинтересовало, можете поискать в Гугле либо посмотреть исходники лекции, которую в данный момент читаете: она собрана с использованием **make**.

Разделяемые (динамические) библиотеки

Так называются библиотеки, которые содержатся в ОС. Распространяются в .dll в ОС Windows или же .so в Linux.

Отступление. Полезный материал на Фоксфорде

Подробности по C++ можно найти здесь (материал достаточно ознакомительный, но из него можно вынести много полезного).

8.4 Условная компиляция

Условная компиляция – компиляция с условиями. Условия компилятору сообщают *директивы* – команды, обрабатываемые на этапе *препроцессинга*. Часто используемая директива – это **#IFDEF**. Приведем пример ее использования:

Код 8.1: Директива IFDEF

```
#ifdef SOMETHING
some code
#else
other code
#endif
```

Код в someCode будет выполнен только тогда, когда был определен SOMETHING. Существует похожая директива: **#IFNDEF** – только он срабатывает, если *не* был определен аргументы.

Еще один т.н. "страж включения": **#define SOMETHING** – объявляет **SOMETHING**. После него **#ifdef** уже будет думать, что **SOMETHING** объявлено.

Все эти штуки используются для того, чтобы не инклюдить один и тот же файл по несколько раз, ведь в таких ситуациях компилятор будет кидать ошибки.

8.5 Макросы

Код 8.2: Пример использования макросов

```
1 #define max(a, b) (((a) > (b)) ? (a) : (b))
2
3 int main()
4 {
5  std::cout << max(5, 3) + 1;
6 }
```

Макросы не рекомендуется использовать – они создают обманчивую привлекательность, однако порождают непредвиденное поведение, поскольку **не** являются вызовами функции, а всего лишь подставляются препроцессором при текстовой обработке.

8.6 extern-переменные

Представим себе, что мы хотим создать переменную, которая будет доступна и в том, и другом модуле. Можно ее описать в **main.cpp**, однако тогда нельзя будет ей пользоваться в тех файлах, которые не подключают **main.cpp**.

У нас есть **fractions.cpp**, которого все подключают. В нее запихнем эту переменную, а в **fraction.hpp** запишем **extern int g;**. Это скажет компилятору, что переменная **g** будет, но стоит ее дожидаться ТОЛЬКО во время компоновки (а не создавать каждый раз при **include**). То есть, все файлы, которые будут инклюдить **fractions.hpp**, будут иметь возможность обращаться к переменной **g**, но при этом сама переменная **g** будет объявлена только в **fraction.cpp** (и никак не в файлах, которые индклюдят **fraction.hpp**). Это делает переменную **g** как глобальной, так и единой для всех файлов — то, чего мы и добиваемся.