Использование языка программирования Julia для решения математических и инженерных задач

Часть I

Автор: **Чуракова Юлия Романовна**
Студентка факультета математики и компьютерных наук
Кубанского Государственного Университета

Автор: **Алексеев Евгений Ростиславович**
Кандидат технических наук, доцент
Доцент кафедры информационных технологий
Кубанского Государственного Университета

Полная версия пособия доступна в репозитории:
https://github.com/JuliaChurakova/Julia

Глава 1 Общие сведения о языке Julia

1.1 Установка Julia и Jupyter

1.1.1 Официальный сайт Julia

Официальный сайт языка программирования Julia можно найти по адресу https://julialang.org/.

На этом сайте можно найти всю необходимую информацию о языке, его особенности, руководство, документацию, а также новости и обновления языка.

1.1.2. Страница загрузки

Страница для загрузки различных версий языка программирования Julia доступна по адресу https://julialang.org/downloads/.

Здесь представлена информация о том, как скачать язык для различных операционных систем, таких как Linux, macOS и Windows.

1.1.3 Установка Julia на Linux

```
curl -fsSL https://install.julialang.org | sh
```

1.1.4 Установка Jupyter

1. **Установка pip**:

   ```
   sudo apt update
   sudo apt install python3-pip
   ```

2. **Обновление pip**:

```
python3 -m pip install --upgrade pip
```

3. **Установка Jupyter Notebook**:

```
pip3 install notebook
```

4. **Дополнительная установка JupyterLab**(более современная и функциональная среда, развивающая идеи классического Notebook):
   `pip3 install jupyterlab`

1.1.5 Особенности ввода в REPL (Read-Eval-Print Loop) языка Julia

Вход в REPL Julia: для начала работы с Julia откройте терминал и введите команду julia. Это запустит интерактивную оболочку REPL (Read-Eval-Print Loop), где можно вводить и выполнять команды на языке Julia.

1. **Системная оболочка**. Режим системных команд предоставляет доступ к командной оболочке операционной системы для выполнения системных операций. Для активации этого режима введите точку с запятой **;** в начале строки.

Чтобы вернуться в основной режим, нужно нажать клавишу BackSpace.

2. **Режим справки**. В REPL доступна система помощи. Для получения справочной информации о функции или пакете можно использовать команду с вопросительным знаком **?**. Например, чтобы узнать, что делает функция **sqrt**, можно ввести:

   ```
   ?sqrt
   ```
   Это откроет справочную информацию о функции.

Чтобы вернуться в основной режим, нужно нажать клавишу BackSpace.

3. **Управления пакетами**. Для активации режима управления пакетами в REPL Julia введите символ **]**. В этом режиме можно управлять пакетами (устанавливать, обновлять, удалять).

Можно также управлять пакетами через API, импортируя модуль Pkg командой **using Pkg**, а затем вызывая команды, например, **Pkg.add("имя пакета")**.

Полезные команды диспетчера пакетов:

- **status:** показывает список установленных пакетов с их версиями;
- **update:** обновляет локальный индекс пакетов и устанавливает последние версии всех пакетов;
- **add <имя пакета>:** устанавливает новый пакет. Для нескольких пакетов используйте **add <имя пакета 1> <имя пакета 2>**;
- **free <имя пакета>:** возвращает пакет к последней стабильной версии;
- **rm <имя пакета>:** удаляет пакет и все его зависимости;
- **add https://github.com/<имя репозитория>/<имя пакета>.jl:** устанавливает пакет с GitHub по URL.

Чтобы вернуться в основной режим, нужно нажать клавишу BackSpace.

4. **Использование установленных пакетов**:

- using: предоставляет прямой доступ ко всем функциям пакета;

```
using MyPackage # Прямой доступ к функциям пакета
my_function()  # Функция вызывается без указания имени пакета
```

- import: требует использования полных имен функций пакета, помогает избежать конфликтов имен.

  ```
  import MyPackage # Для вызова функции нужно использовать полное имя пакета
  MyPackage.my_function()  # Нужно явно указать MyPackage.my_function
  ```

1.1.6 Подключение Julia к Jupyter

Чтобы использовать Julia в Jupyter Notebook, необходимо установить пакет IJulia, который добавляет ядро Julia в Jupyter.

1. Запустите REPL Julia (введя julia в терминале).
2. В режиме REPL выполните следующие команды:

   ```
   using Pkg
   Pkg.add("IJulia")
   Pkg.build("IJulia")
   ```
3. После установки вы можете запустить Jupyter Notebook с поддержкой Julia, выполнив в REPL Julia:

   ```
   using IJulia
   notebook()
   ```
   Либо просто запустите **jupyter notebook** или **jupyter lab** из терминала – при создании нового блокнота в списке доступных ядер появится вариант "Julia".

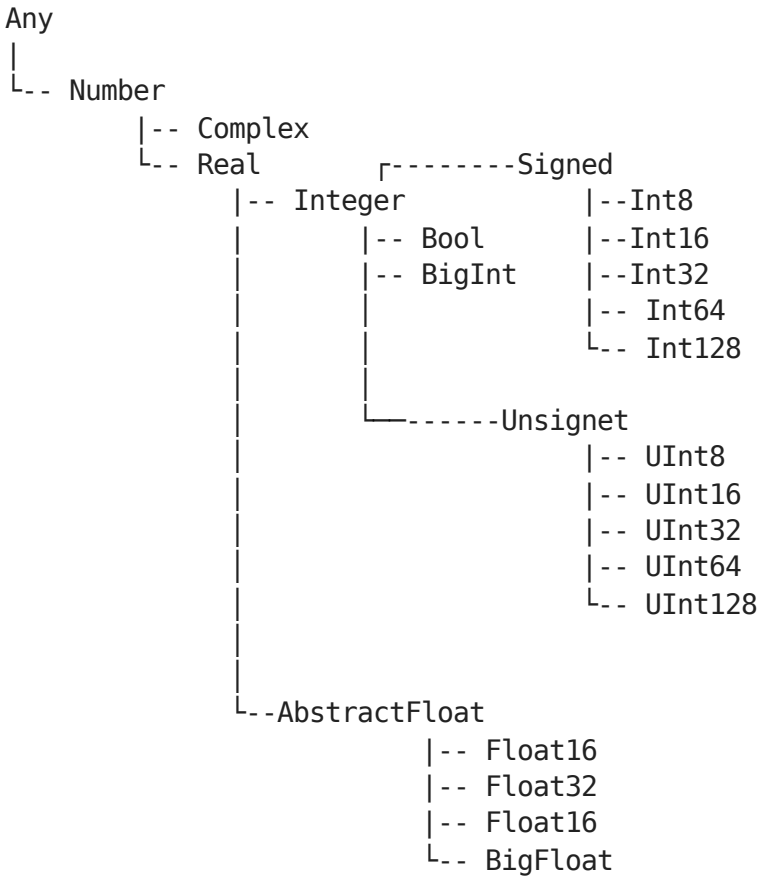
1.1.7 Список библиотек, которые используются в пособии

1. библиотеки для построения графиков: **Plots.jl**, **PyPlot.jl**, **Graphics.jl**;
2. библиотека для создания графических интерфейсов: **Gtk.jl**;
3. библиотека для функций вывода в стиле C: **Printf.jl**;
4. библиотека для работы с линейной алгеброй: **LinearAlgebra.jl**;
5. библиотека для генерации случайных чисел: **Random.jl**;
6. библиотека для работы с датами и временем: **Dates.jl**;
7. библиотека для решения дифференциальных уравнений: **DifferentialEquations.jl**;
8. библиотека для работы с простыми числами и выполнения задач теории чисел: **Primes**;
9. библиотека для решения полиномиальных уравнений: **PolynomialRoots.jl**;

10. библиотека для решения алгебраических и трансцендентных уравнений: **Roots.jl**;
11. библиотека для решения системы уравнений: **NLsolve**;
12. библиотека для решения одномерных интегралов: **QuadGK**;
13. библиотека для решения многомерных интегралов: **Cubature**;
14. альтернатива Cubature с оптимизациями: **HCubature**.

1.2 Числовые типы данных в Julia

Иерархия типов данных в языке программирования Julia организована таким образом, что каждый тип наследует характеристики от более общего типа. Давайте подробно рассмотрим, как устроена эта иерархия.



1. **Any** – самый общий тип в Julia, от которого наследуются все другие типы данных. Это супертип для всех типов.
2. **Number** – включает все числовые типы. Он делится на два подтипа:

- **Complex** – тип для комплексных чисел.
- **Real** включает в себя:
  - **Integer** – тип для целых чисел:
    - **Signed** – знаковые целые числа, такие как:
      - **Int8** – 8-битное целое число.
      - **Int16** – 16-битное целое число.
      - **Int32** – 32-битное целое число.
      - **Int64** – 64-битное целое число.
      - **Int128** – 128-битное целое число.
    - **Bool** – булев тип (может быть только `true` или `false`).
    - **BigInt** – произвольной точности целое число, не ограниченное стандартными размерами.
  - **Unasigned** – тип для целых чисел без знака:
    - **UInt8** – 8-битное целое число без знака.
    - **UInt16** – 16-битное целое число без знака.
    - **UInt32** – 32-битное целое число без знака.
    - **UInt64** – 64-битное целое число без знака.
    - **UInt128** – 128-битное целое число без знака.
  - **AbstractFloat** – тип для вещественных чисел:
    - **Float16** – 16-битное вещественное число.
    - **Float32** – 32-битное вещественное число.
    - **Float64** – 64-битное вещественное число.
    - **BigFloat** – произвольной точности вещественное число, не ограниченное стандартными размерами.

Теперь давайте более подробно рассмотрим каждый из типов данных, чтобы лучше понять их особенности в языке программирования Julia.

1.2.1 Целочисленные данные в Julia

В языке программирования Julia существует несколько типов данных для целых чисел, которые различаются по размеру и диапазону значений. Эти типы включают как знаковые (**Int8**, **Int16**, **Int32**, **Int64**, **Int128**), так и беззнаковые целые числа (**UInt8**, **UInt16**, **UInt32**, **UInt64**, **UInt128**). Каждый из этих типов имеет свои ограничения по диапазону значений, которые могут быть представлены с использованием определённого количества бит.

Для наглядности и понимания, давайте выведем значения, которые могут быть представлены каждым из этих типов, для получения минимального и максимального значений используются функции **typemin** и **typemax**:

In [2]:
```
for T in
[Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
```

```
println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
end
```

```
   Int8: [-128,127]
  Int16: [-32768,32767]
  Int32: [-2147483648,2147483647]
  Int64: [-9223372036854775808,9223372036854775807]
 Int128: [-170141183460469231731687303715884105728,170141183460469231731687
303715884105727]
  UInt8: [0,255]
 UInt16: [0,65535]
 UInt32: [0,4294967295]
 UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]
```

Пример взят из книги:

Белов, Г. В. Краткое описание языка программирования Julia и некоторые примеры его использования/ Г. В. Белов. – Москва: МГТУ им. Н.Э. Баумана, 2024. – 108 с.

В языке программирования Julia переменные по умолчанию имеют тип, аналогичный 64-битному целому числу со знаком (то есть тип Int64). Это значит, что когда вы пишете целое число без указания его типа, Julia автоматически интерпретирует его как Int64. Чтобы убедиться в этом, можно воспользоваться функцией **typeof**, которая возвращает тип данных для заданного значения.

In [3]:
```
typeof(17)
```

Out[3]:
```
Int64
```

Большие целые числа, которые не могут быть представлены с использованием 64 бит, но могут быть представлены в 128 битах будут иметь тип Int128.

In [4]:
```
typeof(134232345454123123213)
```

Out[4]:
```
Int128
```

Для явного указания типа, можно использовать синтаксис, который заранее определяет размер числа.

In [6]:
```
typeof(Int8(17))
```

Out[6]:
```
Int8
```

Выбор подходящего типа данных для целых чисел имеет значение с точки зрения производительности и использования памяти. Например, если вы уверены, что значения чисел будут в пределах диапазона Int8, то использование этого типа сэкономит память, поскольку он занимает всего 1 байт.

Беззнаковые целые числа вводятся и выводятся с использованием префикса 0x. Префикс 0x используется для обозначения чисел в шестнадцатеричной системе счисления. Это позволяет записывать числа, используя цифры от 0 до 9 и буквы от a до f (или от A до F).

In [7]:
```
typeof(0x0a)
```

Out[7]: UInt8

In [8]:
```
typeof(0x1233432456789abcdef)
```

Out[8]: UInt128

Для двоичных чисел используется префикс 0b.

In [9]:
```
b = 0b1010 # Двоичное представление числа 10
```

Out[9]: 0x0a

Для восьмеричных чисел используется префикс 0o.

In [10]:
```
c = 0o12  # Восьмеричное представление числа 10
```

Out[10]: 0x0a

При выводе числа оно автоматически отображается в шестнадцатеричной системе как 0x0a

Значения, слишком большие для типов Int128, UInt128, получают специальный тип **BigInt**. Размер типа BigInt зависит только от доступной оперативной памяти.

In [11]:
```
typeof(100000000000000000000000000000000000000000000000000000000000000000000000000000)
```

Out[11]: BigInt

In [12]:
```
typeof(0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff)
```

Out[12]: BigInt

1.2.2 Вещественные данные в Julia

Как и с целыми числами, существуют числа с плавающей точкой разной длины в битах. По умолчанию числа с плавающей точкой в Julia имеют тип Float64.

In [13]:
```
typeof(0.24324)
```

Out[13]: Float64

In [14]:
```
typeof(0.)
```

Out[14]: Float64

In [15]:
```
typeof(.3)
```

Out[15]: Float64

Можно использовать экспоненциальную форму представления вещественного числа:

In [16]:
```
1e10
```

Out[16]: 1.0e10

In [17]:
```
typeof(1e10)
```

Out[17]: Float64

In [20]:
```
typeof(Float16(0.32323))
```

Out[20]: Float16

Когда точности или размерности Float64 недостаточно, можно использовать специальный тип **BigFloat**.

In [21]:
```
2.0^1000
```

Out[21]: 1.0715086071862673e301

In [22]:
```
BigFloat(2.0)^1000
```

Out[22]:
```
1.071508607186267320948425049060001810561404811705533607443750388370351051124936e+301
```

BigFloat не назначается автоматически при вводе, а требует явного объявления для использования.

1.2.3 Комплексные чисела

In [23]:
```
3 + 4im
```

Out[23]: 3 + 4im

In [24]:
```
typeof(3 + 4im)
```

Out[24]: Complex{Int64}

In [25]:
```
typeof(3.1 + 4im)
```

Out[25]: ComplexF64 (alias for Complex{Float64})

1.3 Определение переменных в Julia

По умолчанию Julia автоматически определяет тип данных переменной в зависимости от присваиваемого значения. Однако в некоторых случаях, чтобы избежать ошибок или повысить производительность, можно явно указать тип данных для переменной. Система типов в Julia гибридная, то есть сочетает элементы как динамической, так и статической типизации.

1.3.1 Динамическая типизация

В Julia переменные не привязаны к определённому типу, и их типы могут изменяться на протяжении выполнения программы.

In [26]:
```
a = 2
typeof(a)
```

Out[26]: Int64

In [27]:
```
a = 9.56
typeof(a)
```

Out[27]: Float64

In [28]:
```
a = 99875434567890654356789087654356789
typeof(a)
```

Out[28]: Int128

In [29]:
```
a = -2.0
typeof(a)
```

Out[29]: Float64

In [30]:
```
sqrt(a)
```

```
DomainError with -2.0:
sqrt was called with a negative real argument but will only return a comple
x result if called with a complex argument. Try sqrt(Complex(x)).

Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math ./math.jl:33
 [2] sqrt(x::Float64)
   @ Base.Math ./math.jl:608
 [3] top-level scope
   @ In[30]:1
```

Ошибка возникает, потому что Julia по умолчанию интерпретирует числовое значение как Float64, что является типом для вещественных чисел. При попытке

работы с комплексными числами, если явно не указать тип данных, Julia может не правильно интерпретировать выражение, что приводит к ошибке.

1.3.2 Статическая типизация

Несмотря на динамическую основу, Julia предоставляет возможность явно задавать типы переменных. Это важная особенность, которая помогает компилятору оптимизировать выполнение программы и избежать ошибок.

In [31]:
```
k::Complex = -2.0
typeof(k)
```

Out[31]:
```
ComplexF64 (alias for Complex{Float64})
```

In [32]:
```
sqrt(k)
```

Out[32]:
```
0.0 + 1.4142135623730951im
```

In [35]:
```
b_new::Int64 = 10
typeof(b_new)
```

Out[35]:
```
Int64
```

Если переменной типа Int64 попытаться присвоить значение с плавающей точкой, Julia выдаст ошибку, предупреждая о несоответствии типов.

In [37]:
```
b_new = 3.14
```

```
InexactError: Int64(3.14)

Stacktrace:
 [1] Int64
   @ ./float.jl:994 [inlined]
 [2] convert(::Type{Int64}, x::Float64)
   @ Base ./number.jl:7
 [3] top-level scope
   @ In[37]:1
```

Отказ от динамической типизации в пользу статической может значительно улучшить производительность программ. Для того чтобы наглядно показать, как это влияет на быстродействие, можно провести тест с использованием функции **@time**, чтобы измерить время выполнения кода с явной типизацией и без неё.

In [38]:
```
g::UInt64 = 18446744073709551615
h::UInt64 = 18446744073709551615

@time for i in 1:1000000 # Выполняем действие 1 миллион раз
    g + h
end
```

```
 0.000001 seconds
```

In [39]:
```
k = 18446744073709551615
f = 18446744073709551615

@time for i in 1:1000000
    k + f
end
```

```
 0.055297 seconds (1.00 M allocations: 45.919 MiB, 42.98% gc time, 14.54% compilation time)
```

При сравнении кода с явной статической типизацией и без неё в Julia видно, что статическая типизация может значительно ускорить выполнение программы.

1.3.3 Присваивание и привязывание значений к переменным

В Julia оператор "=" используется для присваивания значения переменной. Однако если быть точным, то, что Julia делает, является не присваиванием, а привязыванием. В целях более глубокого понимания механизма работы рассмотрим пример кода, в котором переменная "x" сначала привязывается к значению 2. Затем она повторно привязывается к значению "x + 3":

In [42]:
```
#Определяем целочисленную переменную
x = 2
println("x = ", x)
#Вычисляем адрес переменной a с помощью функции  objectid.
println("Адрес переменной x = ", objectid(x))
```

```
x = 2
Адрес переменной x = 13228483051340567920
```

In [43]:
```
#Определяем целочисленную переменную
x = x + 3
println("x = ", x)
#Вычисляем адрес переменной a с помощью функции  objectid.
println("Адрес переменной x = ", objectid(x))
```

```
x = 5
Адрес переменной x = 14624617963239389700
```

Если бы этот пример исходного кода был написан на таком языке,как C/C++, Fortran или Pascal, то для хранения переменной "x" система выделила бы ячейку памяти. При каждом присваивании нового значения переменной x значение в соответствующей ячейке памяти будет изменяться. В случае с привязыванием все работает иначе. Каждое вычисление нужно трактовать как создание числа, которое помещается в другую ячейку памяти. Привязывание предусматривает перемещение самой метки "x" в новую ячейку памяти. Переменная перемещается в результат, а не результат перемещается в переменную. Рассмотрим еще несколько примеров:

In [44]:
```
#Определяем вещественную переменную
x = 10.3
println("x = ", x)
#Вычисляем адрес переменной a с помощью функции  objectid.
println("Адрес переменной x = ", objectid(x))
```

```
x = 10.3
Адрес переменной x = 6536302452756158362
```

In [45]:
```
#Переопределяем вещественную переменную
x = -142.354
println("x = ", x)
#Вычисляем адрес переменной a с помощью функции  objectid.
println("Адрес переменной x = ", objectid(x))
```

```
x = -142.354
Адрес переменной x = 3856324898189001222
```

In [46]:
```
#Определяем строку
x="Пример строки"
println("x = ", x)
#Вычисляем адрес переменной a с помощью функции objectid.
println("Адрес переменной x = ", objectid(x))
```

```
x = Пример строки
Адрес переменной x = 13029857696698101524
```

Julia работает так, что переменная получает значение, вычисленное справа от знака равенства. Присваивание также является выражением, что означает, что его результат можно использовать в других вычислениях. Это позволяет делать несколько присваиваний одновременно или использовать их в более сложных выражениях.

In [47]:
```
x = (y = 6 + 4) * 5
println("x = ", x)
println("y = ", y)
```

```
x = 50
y = 10
```

1.4 Ввод-вывод данных

Print,**println** - это универсальные функции, которые можно использовать для вывода текста на экран. Давайте рассмотрим несколько простых примеров, чтобы продемонстрировать механизм работы этих функций:

In [48]:
```
println("hello"); println("world")
```

```
hello
world
```

In [49]:
```
print("hello"); print("world")
```

```
helloworld
```

В языке программирования Julia символ обратного слэша \ используется для экранирования специальных символов, таких как \n и \t.

\n - символ перевода текста на новую строку.

\t - символ табуляции.

Пример использования в Julia:

In [50]:
```
print("hello\n"); print("world\n")
```

```
hello
world
```

In [51]:
```
print("hello \t world")
```

```
hello    world
```

В Julia можно переносить строки прямо в тексте.

In [5]:
```
print("Строка 1
Строка 2
    Строка 3 с отступом
Строка 4")
```

```
Строка 1
Строка 2
    Строка 3 с отступом
Строка 4
```

Приведенный выше код показывает, что **println** – это тот же самый **print** с добавленным в конце символом новой строки \n.

Для ввода значений через клавиатуру в Julia можно использовать функцию **readline()**, которая считывает строку, введенную пользователем. Затем вы можете преобразовать эту строку в нужный тип данных с помощью функции **parse()**, если это необходимо.

In [52]:
```
println("Введите целое число:")
n = readline()

println("Вы ввели число: ", n)
println(typeof(n))

n = parse(Float64, n)
println("Число преобразованное в Float64: ", n)
println(typeof(n))
```

```
Введите целое число:
```

```
Вы ввели число: 5
String
Число преобразованное в Float64: 5.0
Float64
```

Округление вещественных чисел.

В Julia можно задать количество цифр после запятой, которое нужно выводить для вещественных чисел, используя функцию **round()**.

In [53]:
```
x = 3.14159265
println("x = ", x)
round_x = round(x, digits=2)
println("Округляем до 2 знаков после запятой = ", round_x)

x = 5.46
println("x = ", x)
round_x = round(x, digits=1)
println("Округляем до 1 знака после запятой = ", round_x)
```

```
x = 3.14159265
Округляем до 2 знаков после запятой = 3.14
x = 5.46
Округляем до 1 знака после запятой = 5.5
```

Выравнивание с помощью функций lpad и rpad.

С помощью функций дополнения можно указывать, что строковый литерал всегда должен иметь заданную длину. Если введенный текст меньше, то он будет дополнен выбранным знаком. Если знак не указан, то по умолчанию используется пробел.

In [54]:
```
lpad("ABC", 6, '-') #Дополнение слева.
```

Out[54]:
```
"---ABC"
```

In [55]:
```
rpad("ABC", 6, '-')#Дополнение справа.
```

Out[55]:
```
"ABC---"
```

In [56]:
```
lpad("", 10, '*')rpad("ABC", 10, '*')rpad("ABC", 10, '*')
```

Out[56]:
```
"**********ABC*******ABC*******"
```

Использование греческих букв и символов Юникода в Julia для математических вычислений.

Язык Julia отличается тем, что в нем активно использует греческие буквы, такие как π, θ, α и Δ. Это связано с тем, что в математике и науке часто используются греческие символы для обозначения переменных и констант в уравнениях. Когда такие формулы реализуются в коде на Julia, использование греческих букв делает их более похожими на математические уравнения, что упрощает их

чтение и понимание. Это делает язык Julia удобным для работы с математическими вычислениями.

Ниже приводится сводка из нескольких популярных греческих букв и символов Юникода, которые можно использовать в своем коде.

Символ	Заполнение по Tab
π	\pi
θ	\theta
Δ	\Delta
e	\euler
√	\sqrt
φ	\varphi

In [58]:
```
println(√121)
println(π * 2)
println(ℯ * 10)
```

```
11.0
6.283185307179586
27.18281828459045
```

Как известно из программы средней школы, выражения 3 × x + 2 × y записывают как 3x + 2y. Julia позволяет писать умножение таким же образом. Экземпляры такой записи называются **литеральными коэффициентами** как своего рода аббревиатура умножения числового литерала на константу или переменную.

In [59]:
```
x = 5
x = 2x
println(x)
x = 2(10 + 15)
println(x)
```

```
10
50
```

1.5 Основные операторы языка Julia над числовыми значениями

1.5.1 Базовые арифметические операции

В языке поддерживаются стандартные математические операции:

- `+` – сложение;
- `-` – вычитание;

- `*` – умножение;
- `/` – деление;
- `÷` – целочисленное деление;
- `%` – остаток от деления;
- `^` – возведение в степень.

Для работы с массивами предусмотрены поэлементные операции:
`.+`, `.-`, `.*`, `./`, `.^` – выполняются над каждым элементом векторов или матриц отдельно.

In [63]:
```
# Пример простейших арифметических выражений
println("Введите целое число a")
a = parse(Int, readline())
println("Введите целое число b")
b = parse(Int, readline())

println("a*b=", a * b) # Умножение
println("a/b=", a / b) # Деление
println("a÷b=", a ÷ b) # Целочисленное деление
println("a%b=", a % b) # Остаток от деления
println("a-b=", a - b) # Вычитание
println("a^b=", a ^ b) # Возведение в степень
```

```
Введите целое число a
Введите целое число b
a*b=55
a/b=2.2
a÷b=2
a%b=1
a-b=6
a^b=161051
```

В Julia есть проблема **возведения отрицательного числа в дробную степень**.

In [64]:
```
a=-8
b=a^(1/3)
print("a=",a,"a^(1/3)=",b);
```

```
DomainError with -8.0:
Exponentiation yielding a complex result requires a complex argument.
Replace x^y with (x+0im)^y, Complex(x)^y, or similar.

Stacktrace:
 [1] throw_exp_domainerror(x::Float64)
   @ Base.Math ./math.jl:41
 [2] ^(x::Float64, y::Float64)
   @ Base.Math ./math.jl:1157
 [3] ^(x::Int64, y::Float64)
   @ Base ./promotion.jl:478
 [4] top-level scope
   @ In[64]:2
```

In [65]:
```
# Для корректного возведения отрицатедьного числа в степень
# надо использовать оператор if (см. 3 главу)
println("Введите число a")
a = parse(Float64, readline())
if a>0
    b=a^(1/3)
else
    b=-(abs(a)^(1/3))
end
print("a=",a,"\na^(1/3)=",b)
```

```
Введите число a
a=-5.0
a^(1/3)=-1.7099759466766968
```

В Julia существует возможность ввода строки, которая является арифметическим выражением.

Функция **Meta.parse()** в Julia преобразует строку в форму, которая может быть интерпретирована и выполнена как код. Это позволяет динамически создавать и выполнять код на лету.

Функция **eval()** в Julia используется для выполнения выражений, представленных в виде кода, переданных ей в качестве аргумента. Она позволяет вычислять и выполнять код во время выполнения программы.

In [66]:
```
stroka = "45*9+334"
println(stroka)
result = eval(Meta.parse(stroka))
println("Результат вычисления: ",result)
```

```
45*9+334
Результат вычисления: 739
```

In [67]:
```
println("Введите строку для вычисления:")
stroka = readline()
result = eval(Meta.parse(stroka))
println("Результат вычисления: ",result)
```

```
Введите строку для вычисления:
Результат вычисления: 10.0
```

Можно использовать символ двоеточия для определения выражения, вместо Meta.parse.

In [68]:
```
stroka = :(45*9+334)
println(stroka)
result = eval(stroka)
println("Результат вычисления: ",result)
```

```
45 * 9 + 334
Результат вычисления: 739
```

С помощью оператора **$** можно использовать вычисленные значения при конструировании выражений.

In [69]:
```
x = 5
y = :($x + 10)
```

Out[69]:
```
:(5 + 10)
```

In [70]:
```
eval(y)
```

Out[70]:
```
15
```

В Julia, как и в других языках программирования, существует несколько специальных значений для представления неопределенных или бесконечных величин: Inf, -Inf и NaN.

Inf – это специальное значение, которое представляет положительную бесконечность. Его можно использовать в вычислениях, где результат выходит за пределы конечных чисел.

In [72]:
```
x = Inf
println(x)

y = 1 / 0
println(y)
```

```
Inf
Inf
```

-Inf – это специальное значение, которое представляет отрицательную бесконечность.

In [73]:
```
z = -Inf
println(z)
```

```
-Inf
```

NaN (Not a Number) – это специальное значение, которое используется для представления неопределенных или недопустимых результатов вычислений, например, при делении нуля на ноль.

In [75]:
```
a = 0 / 0
println(a)
```

```
NaN
```

In [76]:
```
x = Inf
y = 1 / 0  # Положительная бесконечность
z = -Inf    # Отрицательная бесконечность

println(x + y)  # бесконечность + бесконечность
```

```
println(x * 0)  # бесконечность * 0
println(Inf / Inf)  # деление бесконечности на бесконечность
```

```
Inf
NaN
NaN
```

1.5.2 Двоичные (арифметические) операторы (битовые операторы)

В языке Julia можно различить унарные и бинарные двоичные операторы над целыми значениями.

Унарные операции включают в себя операцию инверсии (~), где целое число переводится в двоичное представление и каждый бит инвертируется.

Бинарные операции включают в себя:

- **двоичное И (&)**, где оба операнда переводятся в двоичную систему, и над ними выполняется операция побитного И;
- **двоичное ИЛИ (|)**, где оба операнда переводятся в двоичную систему, и над ними выполняется операция побитного ИЛИ;
- **двоичное исключающее ИЛИ (^)**, где оба операнда переводятся в двоичную систему, и над ними выполняется операция побитного исключающего ИЛИ;
- **сдвиг влево (<<)**, где первый операнд переводится в двоичную систему счисления, а затем смещается влево на количество позиций, определяемых вторым операндом (k), что эквивалентно умножению на 2^k;
- **сдвиг вправо (>>)**, где первый операнд переводится в двоичную систему счисления, а затем смещается вправо на количество позиций, определяемых вторым операндом (k), что эквивалентно делению нацело на 2^k.

In [77]:
```
# Унарные операции
# Инверсия (~) - целое число переводится в двоичное представление и побитн
a = 13
println(a, " ", bitstring(a), " ", bitstring(~a), " ", ~a)

# Бинарные операции
# Двоичное И (&), оба операнда переводятся в двоичную систему и над ними п
a = 13
b = 23
c = a & b
println("a=", a, " b=", b, " a&b=", c)

# Двоичное ИЛИ (|), оба операнда переводятся в двоичную систему и над ними
a = 13
b = 23
c = a | b
println("a=", a, " b=", b, " a|b=", c)
```

```
# Двоичное исключающее ИЛИ (^), оба операнда переводятся в двоичную систем
a = 13
b = 23
c = a ⊻ b
println("a=", a, " b=", b, " a^b=", c)

# Сдвиг влево <<, первый операнд переводится в двоичную систему счисления
a = 23
println(a, " ", a << 1, " ", a << 2, " ", a << 3)

# Сдвиг вправо >>, первый операнд переводится в двоичную систему счисления
a = 20
println(a, " ", a >> 1, " ", a >> 2, " ", a >> 3)
```

```
13 00000000000000000000000000000000000000000000000000000000000001101 1111111
11111111111111111111111111111111111111111111111111111110010 -14
a=13 b=23 a&b=5
a=13 b=23 a|b=31
a=13 b=23 a^b=26
23 46 92 184
20 10 5 2
```

1.5.3 Операторы отношения

В Julia определены следующие операторы отношения:

- `>` – больше;
- `<` – меньше;
- `==` – проверка эквивалентности объектов;
- `===` – проверка идентичности объектов;
- `!=` – не равно;
- `!==` – не идентично;
- `>=` – больше или равно;
- `<=` – меньше или равно.

In [1]:
```
3 == 3.0
```

Out[1]:
```
true
```

In [2]:
```
3 === 3.0
```

Out[2]:
```
false
```

In [3]:
```
3.0 === 3.0
```

Out[3]:
```
true
```

Julia позволяет проверять операции двойного неравенства, принятые в математике, например:

In [4]:
```
println("x = ")
x = parse(Float64, readline())
print(-7<x<=3)
```

```
x =
true
```

1.5.4 Логические операторы

Логические операторы языка Julia: **||** (или) и **&&** (и).

In [5]:
```
x = 3
x < 4 || x > 10
```

Out[5]: true

In [6]:
```
x > 4 && x < 10
```

Out[6]: false

1.5.5 Операторы присваивания

In [19]:
```
# Примеры операторов присваивания
x=3
println("x = ",x)
y=z=0.2^1.7
println("y = ",y," z = ",z)
x=3
a=4
println("x = ",x," a = ",a)
x+=a
println("x += a = ",x)
x=3
a=4
println("x = ",x," a = ",a)
x-=a
println("x -= a = ",x)
x=3
a=4
println("x = ",x," a = ",a)
x*=a
println("x *= a = ",x)
x=22
a=5
println("x = ",x," a = ",a)
x/=a
println("x /= a = ",x)
x=22
a=5
println("x = ",x," a = ",a)
x÷=a
println("x ÷= a = ",x)
x=22
```

```
a=5
println("x = ",x," a = ",a)
x%=a
println("x %= a = ",x)
x=22
a=5
println("x = ",x," a = ",a)
x^=a
println("x ^= a = ",x)
```

```
x = 3
0.06482626386771051 z = 0.06482626386771051
x = 3 a = 4
x += a = 7
x = 3 a = 4
x -= a = -1
x = 3 a = 4
x *= a = 12
x = 22 a = 5
x /= a = 4.4
x = 22 a = 5
x ÷= a = 4
x = 22 a = 5
x %= a = 2
x = 22 a = 5
x ^= a = 5153632
```

1.6 Некоторые встроенные функции Julia

Основные арифметические операции:

- rem(a, b) – остаток от деления (аналог a % b);
- div(a, b) – целочисленное деление (аналог a ÷ b);
- floor(a) – округление вниз;
- ceil(a) – округление вверх;
- round(a) – округление до ближайшего целого;
- abs(a) – модуль числа;
- sqrt(a) – квадратный корень;
- cbrt(a) – кубический корень.

Логарифмические функции:

- log(a) – натуральный логарифм;
- log2(a) – логарифм по основанию 2;
- log10(a) – десятичный логарифм;
- log(n, a) – логарифм a по основанию n.

Тригонометрические функции:

- радианы: sin(x), cos(x), tan(x), cot(x), asin(x), acos(x), atan(x), acot(x), sec(x);

- градусы: sind(x), cosd(x), tand(x), cotd(x), asind(x), acosd(x), atand(x), acotd(x), secd(x);
- конвертация:
  - rad2deg(a) – радианы → градусы;
  - deg2rad(a) – градусы → радианы.

Гиперболические функции:

- sinh(x), cosh(x), tanh(x), coth(x).

Специальные функции

- hypot(a, b) – гипотенуза прямоугольного треугольника с катетами a и b;
- factorial(a) – факториал числа.

In [20]:
```
a = -23
b = 3.456
c = 7875
println(floor(b))
println(ceil(b))
println(round(b))
println(abs(a))
println(sqrt(b))
println(cbrt(b))
println(log(c))
println(floor(b))
println(rad2deg(2π))
println(deg2rad(180))
```

```
3.0
4.0
3.0
23
1.85903200617956
1.5119052598738478
8.971448463693834
3.0
360.0
3.141592653589793
```

Глава 2 Структуры данных

2.1 Строки в julia

Строка в Julia – это набор символов, заключенных между двойными кавычками " ". Символы вводятся в кодировке UTF-8. Строки могут содержать специальные символы, например, символ табуляции '\t' или символ перевода на новую строку '\n'.

In [21]:
```
b = "строка 1\nстрока 2\n"
println(b)
```

```
строка 1
строка 2
```

Функция **length()** – возвращает число символов в строке.

In [22]:
```
st = "Конь"
length(st)
```

Out[22]: 4

Строку можно рассматривать как одномерный массив (вектор). Например, если строка s="abc", то s[2]=='b', а s[end]=='c'. Однако изменять элементы строки присваиванием нельзя, т. е. оператор s[3]='d' является ошибочным с точки зрения языка Julia. В этом случае используется функция **replace()**.

In [24]:
```
replace(s,'c'=>'a')
```

Out[24]: "aba"

Еще один нюанс, строка и символ – существенно разные понятия языка Julia,поэтому равенство "A" == 'A' является ложным.

Проверить наличие символа s в строке st можно с использованием конструкции **s in st**, которая возвращает true или false

In [25]:
```
'b' in "abc"
```

Out[25]: true

Проверка того, что подстрока или символ ss входит в строку st осуществляется с использованием функции **occursin(ss,st)**.

In [6]:
```
occursin("CCC","CCCA")
```

Out[6]: true

findfirst(ss,st) – найти первое вхождение подстроки или символа ss в строке st. Если ss – строка, то результатом является первый и последний индексы подстроки ss в строке st. Если ss – символ, результатом будет индекс, который соответствует номеру символа в строке. Если подстрока или символ не найдены, функция возвращает nothing.

In [36]:
```
findfirst("ia","julia")
```

Out[36]: 4:5

In [37]:
```
findfirst('u',"julia")
```

Out[37]:
```
2
```

findlast(ss,st) – найти последнее вхождение подстроки или символа ss в строке st. Если ss – строка, то результатом является первый и последний индексы подстроки ss в строке st. Если ss – символ, результатом будет индекс, который соответствует номеру символа в строке. Если подстрока или символ не найдены, функция возвращает nothing.

In [39]:
```
findlast('m',"comment")
```

Out[39]:
```
4
```

Объединение строк производится с использованием символа `*`.

In [40]:
```
c="Hello,"
d="world!"
e=c*d
```

Out[40]:
```
"Hello,world!"
```

strip(st) – удаляет пробелы вначале и в конце строки, если строка состоит из пробелов, то strip(st) возвращает пустую строку.

In [11]:
```
st = "        Кубанский государственный университет        "
strip(st)
```

Out[11]:
```
"Кубанский государственный университет"
```

isempty(st) – проверяет, есть ли символы в строке, если нет, возвращает true, иначе – false.

In [12]:
```
isempty(st)
```

Out[12]:
```
false
```

split(st,'R') – разобрать строку на элементы, если в качестве разделителя используется символ R.

In [53]:
```
s = "a,bc,d"
split(s,',')
```

Out[53]:
```
3-element Vector{SubString{String}}:
 "a"
 "bc"
 "d"
```

join(A,'R') – преобразовать элементы массива A в строку, используя в качестве разделителя символ R.

In [55]:
```
A=[1.0, 2.0, 3.0]
join(A,',')
```

Out[55]:
```
"1.0,2.0,3.0"
```

uppercase(st) – преобразовать строку в верхний регистр.

In [56]:
```
uppercase(st)
```

Out[56]:
```
"        КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ        "
```

lowercase(st) – преобразовать строку в нижний регистр.

In [57]:
```
lowercase(st)
```

Out[57]:
```
"        кубанский государственный университет        "
```

titlecase(st) – преобразовать в верхний регистр первый символ каждого слова строки.

In [58]:
```
titlecase(st)
```

Out[58]:
```
"        Кубанский Государственный Университет        "
```

string(x) – превратить число x в строку.

In [60]:
```
x = 12312424
string(x)
```

Out[60]:
```
"12312424"
```

В Julia используется такое понятие, как **интерполяция**. Его смысл заключается в следующем: если есть строка st = "123.456" или число a = 3.14, то их значения можно **«встраивать»** (интерполировать) в строку с помощью знака доллара **$**.

In [61]:
```
print("1 + 2 = $(1 + 2)")
```

```
1 + 2 = 3
```

2.2 Массивы в julia

Массив (англ. array) – это структура данных, которая хранит набор значений, идентифицируемых по индексу или набору индексов.

В Julia индексация массивов начинается с 1.

2.2.1 Способы объявления массива

Существует довольно много способов объявления массива. Рассмотрим некоторые из них.

In [26]:
```
# Пустой массив с указанием типа элементов
Int[]          # Пустой массив целых чисел
Float64[]      # Пустой массив чисел с плавающей точкой
String[]       # Пустой массив строк
Any[]          # Пустой массив для элементов любого типа
```

Out[26]:
```
Any[]
```

In [3]:
```
#Одномерный массив с тремя элементами типа Float64, элементы массива не оп
a=Array{Float64,1}(undef,3)
```

Out[3]:
```
3-element Vector{Float64}:
 6.9180327879339e-310
 6.9180632143114e-310
 6.9180315011767e-310
```

In [4]:
```
#Двумерный массив 3*5 – три строки, пять столбцов типа Float64, элементы м
a=Array{Float64,2}(undef,3,5)
```

Out[4]:
```
3×5 Matrix{Float64}:
 0.0       1.5e-323  3.0e-323  4.4e-323  4.4e-323
 5.0e-324  2.0e-323  3.5e-323  5.0e-323  4.4e-323
 1.0e-323  2.5e-323  4.0e-323  5.0e-323  7.4e-323
```

In [5]:
```
#Одномерный массив с десятью элементами типа Float64, элементы массива не
a=Vector{Float64}(undef,10)
```

Out[5]:
```
10-element Vector{Float64}:
 3.5e-323
 0.0
 4.0e-323
 0.0
 4.0e-323
 0.0
 1.0e-323
 0.0
 3.0e-323
 0.0
```

In [6]:
```
#Двумерный массив 2*5 – две строки, пять столбцов типа Int64, элементы мас
a=Matrix{Int64}(undef,2,5)
```

Out[6]:
```
2×5 Matrix{Int64}:
 140023162695376  140023126177104  140023179501576  140023126177296  0
 140023179501576  140023162689488  140023162689616                0  0
```

In [7]:
```
#При таком объявлении (элементы через пробел) Julia создает двумерный масс
a=[1 2 3 4]
```

Out[7]:
```
1×4 Matrix{Int64}:
 1  2  3  4
```

In [8]:
```
#Создание одномерного массива (вектора)
a=[1, 2, 3, 4]
```

Out[8]:
```
4-element Vector{Int64}:
 1
 2
 3
 4
```

In [9]:
```
a=[1 2; 3 4] #Двумерная матрица
```

Out[9]:
```
2×2 Matrix{Int64}:
 1  2
 3  4
```

In [12]:
```
a=zeros(3) #Одномерный массив с тремя элементами типа Float64, с нулевыми
```

Out[12]:
```
3-element Vector{Float64}:
 0.0
 0.0
 0.0
```

In [11]:
```
a=zeros(3,4) #Двумерный массив 3*4 типа Float64, с нулевыми значениями эле
```

Out[11]:
```
3×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
```

In [13]:
```
a=ones(3) #Одномерный массив с тремя элементами типа Float64, с единичными
```

Out[13]:
```
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

In [14]:
```
a=ones(3,4) #Двумерный массив 3*4 типа Float64, с единичными значениями эл
```

Out[14]:
```
3×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

In [16]:
```
a=Int64.(zeros(3)) #Одномерный массив с тремя элементами типа Int64, с нул
```

Out[16]:
```
3-element Vector{Int64}:
 0
 0
 0
```

In [17]:
```
a=zeros(3,4,5) #Двумерный массив 3*4*5 , с нулевыми значениями элементов
```

Out[17]:
```
3×4×5 Array{Float64, 3}:
[:, :, 1] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

[:, :, 3] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

[:, :, 4] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

[:, :, 5] =
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
```

In [19]:
```
a=fill(5,3,3) #Двумерный массив 3*3 типа Int64
```

Out[19]:
```
3×3 Matrix{Int64}:
 5  5  5
 5  5  5
 5  5  5
```

In [20]:
```
a=rand(5) #Одномерный массив из 5 элементов, заполненный случайными числам
```

Out[20]:
```
5-element Vector{Float64}:
 0.3084198317399174
 0.29333039483790624
 0.05875249707582253
 0.9398077001166478
 0.6792000600065631
```

In [21]:
```
a=rand(1:5,5) #Одномерный массив из 5 элементов, заполненный случайными це
```

Out[21]:
```
5-element Vector{Int64}:
 5
 1
 1
 1
 2
```

In [24]:
```
a = 4
b = 2
c = 20
x=collect(a:b:c) #Создать одномерный массив, первый элемент которого равен
```

Out[24]:
```
9-element Vector{Int64}:
  4
  6
  8
 10
 12
 14
 16
 18
 20
```

2.2.2 Работа с элементами массива

push!(A,V) – добавить элемент V в конец массива A.

В Julia функция с «!» в конце (например, push!) изменяет исходные данные, поступающие в функцию, а без «!» (например, sort) оставляет их неизменными.

In [35]:
```
a = [3, 56, 75, 3]
push!(a,23)
```

Out[35]:
```
5-element Vector{Int64}:
  3
 56
 75
  3
 23
```

pushfirst!(A,V) – добавить элемент V в начало массива A.

In [36]:
```
pushfirst!(a,3455)
```

Out[36]:
```
6-element Vector{Int64}:
 3455
    3
   56
   75
    3
   23
```

insert!(A,N,X) – для массива A типа Vector определена операция вставки элемента X в произвольную позицию N.

In [37]:
```
insert!(a,3,10000)
```

Out[37]:
```
7-element Vector{Int64}:
  3455
     3
 10000
    56
    75
     3
    23
```

pop!(a) – удаление последнего элемента массива.

In [39]:
```
pop!(a)
print(a)
```

```
[3455, 3, 10000, 56, 75]
```

popfirst!(a) – удаление первого элемента массива.

In [41]:
```
popfirst!(a)
print(a)
```

```
[3, 10000, 56, 75]
```

deleteat!(a, n) – удаление элемента массива в позиции n.

In [42]:
```
deleteat!(a, 2)
```

Out[42]:
```
3-element Vector{Int64}:
  3
 56
 75
```

a=a[end:-1:1] – поменять порядок элементов массива на обратный.

In [43]:
```
a=a[end:-1:1]
```

Out[43]:
```
3-element Vector{Int64}:
 75
 56
  3
```

length(a) – определить длину массива.

In [45]:
```
length(a)
```

Out[45]:
```
3
```

maximum(a) – найти максимальное число массива.

In [46]:
```
maximum(a)
```

Out[46]:
```
75
```

minimum(a) – найти минимальное число массива.

In [48]:
```
minimum(a)
```

Out[48]:
```
3
```

2.2.3 Выборки

A[i1:i2:i3] – выбрать элементы вектора A с i_1 по i_3 с шагом i_2.

In [49]:
```
a = [3, 4, 5, 0]
```

Out[49]:
```
4-element Vector{Int64}:
 3
 4
 5
 0
```

In [50]:
```
a[2:1:3]
```

Out[50]:
```
2-element Vector{Int64}:
 4
 5
```

In [51]:
```
b = [3 4 6; 4 9 2; 4 1 10]
```

Out[51]:
```
3×3 Matrix{Int64}:
 3  4   6
 4  9   2
 4  1  10
```

In [52]:
```
b[2:3,2:3] #Выбрать элементы матрицы из строк 2, 3 и столбцов 2, 3.
```

Out[52]:
```
2×2 Matrix{Int64}:
 9   2
 1  10
```

In [54]:
```
b[1:2,:] #Выбрать строки 1 и 2.
#Двоеточие на месте одного из индексов означает, что выбераем каждый элеме
```

Out[54]:
```
2×3 Matrix{Int64}:
 3  4  6
 4  9  2
```

Применительно к элементам массива можно использовать операции с точкой

In [55]:
```
a.^2 # вычислить квадрат всех элементов массива
```

Out[55]:
```
4-element Vector{Int64}:
  9
 16
 25
  0
```

In [56]:
```
a = [2, 4, 6]
b = [4, 10, 12]
a.+b
```

Out[56]:
```
3-element Vector{Int64}:
  6
 14
 18
```

In [57]:
```
a.*b
```

Out[57]:
```
3-element Vector{Int64}:
  8
 40
 72
```

In [58]:
```
log.(a) #Вычислить натуральный логарифм всех элементов массива
```

Out[58]:
```
3-element Vector{Float64}:
 0.6931471805599453
 1.3862943611198906
 1.791759469228055
```

In [59]:
```
2 .*a
```

Out[59]:
```
3-element Vector{Int64}:
  4
  8
 12
```

2.2.4 Объединение двух массивов

Если массивы a и b имеют одинаковое число столбцов, то объединить их (по вертикали) можно командой **vcat(a,b)** или **[a;b]**.

In [60]:
```
a = [2.34, 4.355, 6.87]
b = [4, 10, 12]
vcat(a,b)
```

Out[60]:
```
6-element Vector{Float64}:
  2.34
  4.355
  6.87
  4.0
 10.0
 12.0
```

In [61]:
```
[a;b]
```

Out[61]:
```
6-element Vector{Float64}:
  2.34
  4.355
  6.87
  4.0
 10.0
 12.0
```

Если массивы a и b имеют одинаковое число строк, то объединить их (по горизонтали) можно командой **hcat(a,b)** или **[a b]**.

In [62]:
```
hcat(a,b)
```

Out[62]:
```
3×2 Matrix{Float64}:
 2.34    4.0
 4.355  10.0
 6.87   12.0
```

In [63]:
```
[a b]
```

Out[63]:
```
3×2 Matrix{Float64}:
 2.34    4.0
 4.355  10.0
 6.87   12.0
```

2.2.5 Изменение размерности массива

При необходимости можно изменить размерность массива с использованием функции **reshape()**. Допустим, нужно преобразовать одномерный массив a= [1,2,3,4,5,6] в двумерный с двумя строками и тремя столбцами. Это можно сделать следующим образом:

In [64]:
```
a=[1,2,3,4,5,6]
a=reshape(a,2,3)
```

Out[64]:
```
2×3 Matrix{Int64}:
 1  3  5
 2  4  6
```

In [65]:
```
a=reshape(a,6)
```

Out[65]:
```
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

2.3 Словари

Словари – это неупорядоченные коллекции пар "ключ-значение". Они позволяют эффективно хранить и извлекать данные по ключу. Ключи в словарях должны быть уникальными и неизменяемыми, а значения могут быть любыми и изменяемыми.

2.3.1 Создание словарей

Словари создаются с использованием конструктора **Dict()**.

In [66]:
```
# Пустой словарь
d = Dict()
```

```
# Словарь с элементами
a = Dict("apple" => 1.2, "banana" => 0.8, "cherry" => 2.5)
```

Out[66]:
```
Dict{String, Float64} with 3 entries:
  "cherry" => 2.5
  "banana" => 0.8
  "apple"  => 1.2
```

2.3.2 Доступ к элементам словаря

Доступ к значениям словаря осуществляется с помощью ключей. Если ключ не существует в словаре, возникает ошибка KeyError.

In [67]:
```
a = Dict("milk" => 1.5, "bread" => 2.0)

# Доступ к значениям
b = a["milk"]
print(b)
# Попытка доступа к несуществующему ключу
c = a["butter"]  # KeyError
```

```
1.5
KeyError: key "butter" not found

Stacktrace:
 [1] getindex(h::Dict{String, Float64}, key::String)
   @ Base ./dict.jl:477
 [2] top-level scope
   @ In[67]:7
```

2.3.3 Добавление и изменение элементов

Для добавления новых пар "ключ-значение" и изменения существующих используется синтаксис с квадратными скобками.

In [68]:
```
# Создание словаря
a = Dict("name" => "Alice", "age" => 30)
println(a)
# Добавление новой пары
a["city"] = "New York"
println(a)
# Изменение значения существующего ключа
a["age"] = 31
println(a)
```

```
Dict{String, Any}("name" => "Alice", "age" => 30)
Dict{String, Any}("name" => "Alice", "city" => "New York", "age" => 30)
Dict{String, Any}("name" => "Alice", "city" => "New York", "age" => 31)
```

2.3.4 Удаление элементов

Элементы можно удалять с помощью функции **delete!**().

In [69]:
```
# Создание словаря
a = Dict("name" => "Alice", "age" => 30, "city" => "New York")
println(a)
# Удаление элемента
delete!(a, "city")
println(a)
```

```
Dict{String, Any}("name" => "Alice", "city" => "New York", "age" => 30)
Dict{String, Any}("name" => "Alice", "age" => 30)
```

2.3.5 Проверка наличия ключа

Для проверки наличия ключа в словаре используется функция **haskey()**.

In [70]:
```
a = Dict("milk" => 1.5, "bread" => 2.0)

# Проверка наличия ключа
println(haskey(a, "milk"))
println(haskey(a, "butter"))
```

```
true
false
```

2.3.5 Объединение словарей

Для объединения словарей можно использовать функцию **merge()**.

In [74]:
```
dict1 = Dict("a" => 1, "b" => 2)
dict2 = Dict("c" => 3, "d" => 4)
dict3 = Dict("e" => 5, "f" => 6)

# Объединение словарей
dict4 = merge(dict1, dict2, dict3)
```

Out[74]:
```
Dict{String, Int64} with 6 entries:
  "f" => 6
  "c" => 3
  "e" => 5
  "b" => 2
  "a" => 1
  "d" => 4
```

2.4 Кортежи

Кортежи (tuples) – это упорядоченные, неизменяемые коллекции элементов различных типов.

2.4.1 Создание кортежей

Кортежи создаются с использованием круглых скобок и запятых для разделения элементов.

In [75]:
```
t = (1, "hello", 3.5)
```

Out[75]:
```
(1, "hello", 3.5)
```

Кортежи с одним элементом создаются с запятой, чтобы избежать двусмысленности.

In [76]:
```
t = (1,)
```

Out[76]:
```
(1,)
```

2.4.2 Доступ к элементам кортежа

Элементы кортежа можно получить с помощью индексации.

In [77]:
```
t = (10, 20, 30, 40)

# Доступ к первому элементу
first_element = t[1]
println(first_element)
# Доступ к последнему элементу
last_element = t[end]
println(last_element)
```

```
10
40
```

2.4.3 Распаковка кортежей

Распаковка позволяет присвоить элементы кортежа переменным.

In [78]:
```
coordinates = (3, 5)

# Распаковка кортежа
x, y = coordinates

println("x = $x, y = $y")
```

```
x = 3, y = 5
```

2.4.4 Неизменяемость кортежей

Попытка изменить элемент кортежа приведет к ошибке.

In [79]:
```
t = (1, 2, 3)
t[1] = 10
```

```
MethodError: no method matching setindex!(::Tuple{Int64, Int64, Int64}, ::I
nt64, ::Int64)
The function `setindex!` exists, but no method is defined for this combinat
ion of argument types.

Stacktrace:
 [1] top-level scope
   @ In[79]:2
```

2.4.5 Вложенные кортежи

Кортежи могут содержать другие кортежи, создавая вложенные структуры.

In [80]:
```
t = (1, (2, 3), (4, (5, 6)))

# Доступ к элементам вложенного кортежа
a = t[2]
println(a)
b = t[3][2][1]
println(b)
```

```
(2, 3)
5
```

2.4.6 Преобразование кортежей

Хотя кортежи неизменяемы, можно создать новый кортеж на основе существующего, добавив или изменив элементы.

In [81]:
```
t = (1, 2, 3)

# Добавление элемента
new_t = (t..., 4)
println(new_t)
# Изменение элемента (создание нового кортежа)
modified_t = (t[1], 42, t[3])
println(modified_t)
```

```
(1, 2, 3, 4)
(1, 42, 3)
```

Глава 3 Управляющие конструкции языка Julia

3.1 Условные операторы

3.1.1 Условный (тройной) оператор

Условный оператор вида **условие ? выражение_если_истина : выражение_если_ложь** выполняет второе выражение, если условие истинно, и третье – если ложно, при этом символы **?** и **:** обязательно отделяются пробелами.

In [9]:
```
x = 0
x > 0 ? println("x > 0") : println("x <= 0")
```

```
x <= 0
```

In [8]:
```
x = -5
x > 0 ? println("x > 0") : x == 0 ? println("x = 0") : println("x < 0")
```

```
x < 0
```

3.1.2 Условный оператор if

В Julia простейшая форма условного оператора имеет вид:

```
if логическое_выражение
        операторы, выполняемые, когда логическое выражение истинно
end
```

В такой форме действия после двоеточия выполняются, если логическое выражение истинно. Если же оно ложно, программа ничего не делает и переходит к следующему оператору. Полная форма оператора **if**:

```
if логическое_выражение
        операторы, выполняемые, когда логическое выражение истинно
else
        операторы, выполняемые, когда логическое выражение ложно
end
```

Если нужно последовательно проверить несколько условий, используется расширенная форма с дополнительным оператором **elseif**:

```
if логическое_выражение
        операторы, выполняемые, когда логическое выражение истинно
elseif второе_логическое_выражение
        операторы, выполняемые, когда второе логическое выражение
истинно
elseif третье_логическое_выражение
        операторы, выполняемые, когда третье логическое выражение
истинно
...
else
        операторы, выполняемые, когда все логические выражения ложны
end
```

Дополнительных условий и связанных с ними блоков **elseif** может быть сколько угодно. Если некоторое условие оказалось истинным, соответствующий блок кода выполняется, и дальнейшие условия не проверяются.

3.1.3 Примеры задач

Задача 1. Решить квадратное уравнение общего вида:

$$a \cdot x^2 + b \cdot x + c = 0$$

Версия 1.

In [19]:
```
println("Введите коэффициент a:")
a = parse(Float64, readline())

println("Введите коэффициент b:")
b = parse(Float64, readline())

println("Введите коэффициент c:")
c = parse(Float64, readline())

d = b * b - 4 * a * c
x1 = (-b + sqrt(d)) / 2 / a
x2 = (-b - sqrt(d)) / 2 / a

println("Два действительных корня:")
print("x1 = ",x1,"\t","x2 = ",x2)
```

```
Введите коэффициент a:
Введите коэффициент b:
Введите коэффициент c:
Два действительных корня:
x1 = -0.4       x2 = 1.0
```

Версия 2.

In [18]:
```
println("Введите коэффициент a:")
a = parse(Float64, readline())

println("Введите коэффициент b:")
b = parse(Float64, readline())

println("Введите коэффициент c:")
c = parse(Float64, readline())

d = b * b - 4 * a * c

if d >= 0
    x1 = (-b + sqrt(d)) / 2 / a
    x2 = (-b - sqrt(d)) / 2 / a

    println("Два действительных корня:")
    print("x1 = ",x1,"\t","x2 = ",x2)
```

```
else
    print("Действительных корней нет")
end
```

```
Введите коэффициент a:
Введите коэффициент b:
Введите коэффициент c:
Действительных корней нет
```

Версия 3. Исправим ошибку вычисления корня из отрицательного числа.

In [22]:
```
println("Введите коэффициент a:")
a = parse(Float64, readline())

println("Введите коэффициент b:")
b = parse(Float64, readline())

println("Введите коэффициент c:")
c = parse(Float64, readline())

# Вычисление дискриминанта
D = b^2 - 4*a*c

if D > 0
    # Два действительных корня
    x1 = (-b + sqrt(D)) / (2*a)
    x2 = (-b - sqrt(D)) / (2*a)

    println("Два действительных корня:")
    println("x_1 = ", x1)
    println("x_2 = ", x2)
elseif D == 0
    # Один действительный корень
    x = -b / (2*a)

    println("Один действительный корень:")
    println("x = ", x)
else
    # Два комплексных корня
    real = -b / (2*a)
    imag = sqrt(-D) / (2*a)

    println("Два комплексных корня:")
    println("x1 = ", real, " + ", imag, "i")
    println("x2 = ", real, " - ", imag, "i")
end
```

```
Введите коэффициент a:
Введите коэффициент b:
Введите коэффициент c:
Два комплексных корня:
x1 = -0.625 + 1.0532687216470449i
x2 = -0.625 - 1.0532687216470449i
```

Задача 2. Решить кубическое уравнение общего вида:

$$a \cdot x^3 + b \cdot x^2 + c \cdot x + d = 0$$

In [25]:

```
println("a = ")
a = parse(Float64, readline())
println("b = ")
b = parse(Float64, readline())
println("c = ")
c = parse(Float64, readline())
println("d = ")
d = parse(Float64, readline())


r = b / a
s = c / a
t = d / a


p = s - r^2/3
q = (2*r^3)/27 - (r*s)/3 + t
D = (q/2)^2 + (p/3)^3

if D > 0
    # Один действительный корень
    u = cbrt(-q/2 + sqrt(D)) #cbrt - вычисление кубического корня
    v = cbrt(-q/2 - sqrt(D))
    x1 = u + v - r/3
    println("Один действительный корень:")
    println("x1 = ", x1)

    # Два комплексно-сопряженных корня
    real = -(u + v)/2 - r/3
    imag = abs(u - v)*sqrt(3)/2
    println("Два комплексных корня:")
    println("x2 = ", real, " + ", imag, "im")
    println("x3 = ", real, " - ", imag, "im")

elseif D == 0
    # Кратные корни
    x1 = 2*cbrt(-q/2) - r/3
    x2 = -cbrt(-q/2) - r/3
    println("Два действительных корня (один кратный):")
    println("x1 = ", x1)
    println("x2 = ", x2, " (кратный)")
else
    # Три действительных корня
    fi = acos(-q/2 * sqrt(-27/p^3))
    root = 2 * sqrt(-p/3)

    x1 = root * cos(fi/3) - r/3
    x2 = root * cos((fi + 2π)/3) - r/3
    x3 = root * cos((fi + 4π)/3) - r/3

    println("Три действительных корня:")
    println("x1 = ", x1)
    println("x2 = ", x2)
```

```
    println("x3 = ", x3)
end
```

```
a =
b =
c =
d =
Один действительный корень:
x1 = 0.5154597608712708
Два комплексных корня:
x2 = -0.8827298804356354 + 1.459729914865296im
x3 = -0.8827298804356354 - 1.459729914865296im
```

3.2 Операторы цикла Julia

3.2.1 Цикл while

Общая структура цикла **while**

```
while условие
        оператор 1
        …
        оператор n
end
```

Задача 1. Напишите программу, которая ищет и выводит наибольшее отрицательное число из всех введенных пользователем значений, ввод прекращается, когда вводится 0.

In [27]:
```
# Запрашиваем у пользователя первое число
println("N = ")
N = parse(Float64, readline())

# Инициализируем счетчик отрицательных чисел и переменную для хранения мак
kp = 0    # счетчик отрицательных чисел
mx = 0    # будет хранить максимальное отрицательное число

# Основной цикл, который работает пока пользователь не введет 0
while N != 0
    # Проверяем, является ли введенное число отрицательным
    if N < 0
        # Увеличиваем счетчик отрицательных чисел
        kp = kp + 1

        # Если это первое отрицательное число, сразу запоминаем его как ма
        if kp == 1
            mx = N
        # Иначе сравниваем с текущим максимальным и обновляем при необходи
        elseif N > mx
            mx = N
        end
    end

    # Запрашиваем следующее число у пользователя
```

```
    println("N = ")
    N = parse(Float64, readline())
end

# После завершения цикла выводим максимальное отрицательное число
println("Количество отрицательных чисел: ", kp)
println("Максимальное отрицательное число: ", mx)
```

```
N =
N =
N =
N =
N =
N =
N =
N =
N =
Количество отрицательных чисел: 5
Максимальное отрицательное число: -1.0
```

Операторы управления циклом. Оператор **continue** начинает следующий проход цикла, минуя оставшееся тело цикла (**for** или **while**). Оператор **break** досрочно прерывает цикл.

In [28]:
```
i = 0
while i < 10
    i += 1  # увеличиваем i на 1

    if i % 2 == 0
        continue  # пропускаем чётные числа
    end

    println("Нечётное i = ", i)

    if i >= 7
        break  # выходим из цикла, если i >= 7
    end
end
```

```
Нечётное i = 1
Нечётное i = 3
Нечётное i = 5
Нечётное i = 7
```

3.2.2 Оператор for

Цикл for можно организовать так: `i1` – начальное значение, `i2` – конечное, а `i3` (по умолчанию равный 1) – шаг, который можно не указывать, если он равен единице.

```
for i in i1:i2
        оператор 1
        …
```

```
        оператор n
end
for i = i1:i2
        оператор 1
        …
        оператор n
end
for i in i1:i3:i2
        оператор 1
        …
        оператор n
end
for i = i1:i3:i2
        оператор 1
        …
        оператор n
end
```

Задача 1. Переменная x меняется от xn до xk с шагом dx. Значение y вычисляется по формуле

$$e^{sin(x)}cos(x)$$

Найти сумму и произведение значений y, минимальное и максимальное значение y.

In [15]:
```
println("Введите начальное значение xn:")
xn = parse(Float64, readline())

println("Введите конечное значение xk:")
xk = parse(Float64, readline())

println("Введите шаг dx:")
dx = parse(Float64, readline())

# Инициализация переменных
s = 0.0    # Сумма значений y
p = 1.0    # Произведение значений y
min = Inf  # Минимальное значение y
max = -Inf # Максимальное значение y

for x in xn:dx:xk
    y = exp(sin(x)) * cos(x)

    println("x=$x \t y=$y") # Вывод текущих значений

    s += y
    p *= y

    if y < min
        min = y
    end
    if y > max
        max = y
```

```
    end
end
println("Сумма=$s\nПроизведение=$p\nМинимум=$min\nМаксимум=$max")
```

```
Введите начальное значение xn:
Введите конечное значение xk:
Введите шаг dx:
x=4.0    y=-0.3066661772342105
x=4.1    y=-0.25360722271836467
x=4.2    y=-0.20507212889574455
x=4.3    y=-0.16033967719350098
x=4.4    y=-0.11866796146370862
x=4.5    y=-0.07930964690547916
x=4.6    y=-0.0415197321954094
x=4.7    y=-0.004557884361316417
x=4.8    y=0.03231277189004458
x=4.9    y=0.06982868939983652
x=5.0    y=0.10872913262953789
Сумма=-0.9588698370483153
Произведение=1.1173944655306317e-12
Минимум=-0.3066661772342105
Максимум=0.10872913262953789
```

3.3 Исключения

try/catch – это блок для обработки исключений.

Пример 1. Ошибка выхода за пределы массива

In [30]:
```
# Массив с числами
arr = [1, 2, 3]

try
    # Попытка обратиться к несуществующему индексу массива
    println(arr[5])
catch e
    println("Ошибка: выход за пределы массива.")
end
```

```
Ошибка: выход за пределы массива.
```

Пример 2. Обработка ошибки преобразования типов.

In [31]:
```
try
    # Попытка преобразовать строку в число
    num = parse(Int, "abc")  # Эта строка вызовет ошибку
    println(num)
catch e
    # Этот блок выполнится при возникновении ошибки
    println("Ошибка: не удалось преобразовать строку в число.")
    println("Тип ошибки: ", typeof(e))
    println("Сообщение: ", e.msg)
end
```

```
Ошибка: не удалось преобразовать строку в число.
Тип ошибки: ArgumentError
Сообщение: invalid base 10 digit 'a' in "abc"
```

Пример 3. Работа с файлами (например, файл не существует).

In [32]:
```
try
    # Попытка открыть несуществующий файл
    file = open("file.txt", "r")
    println(readline(file))
catch e
    println("Ошибка: файл не найден.")
end
```

```
Ошибка: файл не найден.
```

Подробнее о работе с файлами в Julia, включая создание, чтение и запись вы можете найти в **Главе 11**.

Глава 4 Функции в Julia

4.1 Базовый синтаксис определения функции.

Структура функции в языке Julia:

```
function name(список параметров)
тело функции
return результат  # не обязателен, если return не указан, функция возвращает результат последнего вычисленного выражения.
end
```

Ключевое слово return обеспечивает выход из функции и может встречаться в функции несколько раз.

```
function test(n)
    if n < 0
        return "n < 0"
    elseif n==0
        return "n == 0"
    else
        return "n > 0"
    end
end
```

Вызов функции:

```
name(список параметров)
```

Рассмотрим на примере функции нахождения корней квадратного уравнения.

In [34]:
```
function quadratic_roots(a, b, c)
    d = b^2 - 4*a*c
    if d < 0
```

```
        return "Нет действительных корней"
    else
        x1 = (-b + sqrt(d)) / (2a)
        x2 = (-b - sqrt(d)) / (2a)
        return x1, x2
    end
end

roots = quadratic_roots(1, -3, 2)
```

Out[34]: (2.0, 1.0)

4.2 Однострочные функции.

В языке программирования Julia можно определять функции не только с использованием традиционного синтаксиса **function ... end**, но и с помощью более компактного синтаксиса, который позволяет определить функцию в одну строку. Такие функции называются **однострочными функциями**.

`function_name(список параметров) = выражение`

In [35]:
```
add(x, y) = x + y
println(add(5, 6))
```

```
11
```

4.3 Анонимные функции.

Анонимные функции в Julia – это функции, которые не имеют имени и определяются прямо в месте их использования.

Анонимные функции в Julia обычно создаются с использованием стрелочного синтаксиса `->`, который указывает на определение функции. Стрелка разделяет параметры функции от тела функции.

In [36]:
```
r = x -> x^2
println(r(5))
```

```
25
```

Анонимные функции могут принимать несколько аргументов, и это делается аналогично обычным функциям.

In [37]:
```
r = (x, y) -> x + y
println(r(3, 4))
```

```
7
```

Анонимные функции могут возвращать несколько значений, если они заключены в кортеж (или массив). Это полезно, если нужно вернуть сразу

несколько результатов.

In [40]:
```
r = (x, y) -> (x + y, x * y)
println(r(5, 6))
```

```
(11, 30)
```

По умолчанию анонимные функции в Julia предполагают, что тело функции состоит из одного выражения. Однако можно использовать более сложные блоки кода, если они обернуты в **begin ... end**. В таком случае анонимная функция может содержать несколько выражений.

In [41]:
```
r = x -> begin
    z = x^2
    z + 1
end
println(r(3))
```

```
10
```

In [42]:
```
arr = [1, 2, 3, 4, 5]
r = map(x -> x^2, arr) # map применяет анонимную функцию ко всем элементам
println(r)
```

```
[1, 4, 9, 16, 25]
```

4.4 Блоки do

Блоки **do** создает анонимную функцию и передает ее в качестве первого аргумента внешней функции при вызове.

In [43]:
```
map([1, 2, 3]) do x
2*x
end
```

Out[43]:
```
3-element Vector{Int64}:
 2
 4
 6
```

In [44]:
```
map([1, 2, 3], [1, 2, 3]) do x, y
x + y
end
```

Out[44]:
```
3-element Vector{Int64}:
 2
 4
 6
```

Глава 5 Примеры программ на языке Julia

ЗАДАЧА 1. Найти первые **M** чисел Армстронга. Число Армстронга – натуральное число, равно сумме своих цифр, возведённых в степень, равную количеству его цифр.

Мы решили проверить скорость работы двух программ, написанных на Python и Julia, которые находят и выводят первые M чисел Армстронга.

Код на Python:

```
import time

def armstrong(x):
    p = x
    x_str = str(x)
    l = len(x_str)
    sum_x = 0
    while x != 0:
        sum_x += (x % 10) ** l
        x = x // 10
    if sum_x == p:
        return True
    else:
        return False

def find_armstrong_numbers(M):
    m = 0
    x = 0
    while m < M:
        x += 1
        if armstrong(x):
            m += 1
            print(f"{m}) x={x}")
    print("Цикл завершился!")

print("Введите количество чисел Армстронга, которые хотите найти: ")
M = int(input())
start_time = time.time()
find_armstrong_numbers(M)
end_time = time.time()
print(f"Время выполнения: {end_time - start_time:.6f} секунд")
```

Результат работы программы

```
Введите количество чисел Армстронга, которые хотите найти: 20
1) x=1
2) x=2
3) x=3
4) x=4
5) x=5
6) x=6
7) x=7
8) x=8
```

```
9) x=9
10) x=153
11) x=370
12) x=371
13) x=407
14) x=1634
15) x=8208
16) x=9474
17) x=54748
18) x=92727
19) x=93084
20) x=548834
Цикл завершился!
Время выполнения: 0.823005 секунд
```

In [45]:

```
function armstrong(x)
    p = x
    x_str = string(x)
    l = length(x_str)
    sum_x = 0
    while x != 0
        sum_x += (x % 10)^l
        x = x ÷ 10
    end
    if sum_x == p
        return true
    else
        return false
    end
end

function find_armstrong_numbers(M)
    m = 0
    x = 0
    while m < M
        x += 1
        if armstrong(x)
            m += 1
            println("$m) x=$x")
        end
    end
    println("Цикл завершился!")
end

println("Введите количество чисел Армстронга, которые хотите найти: ")
M = parse(Int, readline())
start_time = time()
find_armstrong_numbers(M)
end_time = time()
println("Время выполнения: $(end_time - start_time) секунд")
```

```
Введите количество чисел Армстронга, которые хотите найти:
```

```
1) x=1
2) x=2
3) x=3
4) x=4
5) x=5
6) x=6
7) x=7
8) x=8
9) x=9
10) x=153
11) x=370
12) x=371
13) x=407
14) x=1634
15) x=8208
16) x=9474
17) x=54748
18) x=92727
19) x=93084
20) x=548834
Цикл завершился!
Время выполнения: 0.11522698402404785 секунд
```

Julia продемонстрировала значительно более высокую производительность по сравнению с Python. Время выполнения программы на Julia для нахождения и вывода первых M чисел Армстронга было заметно меньше, что свидетельствует о высокой эффективности и оптимизации языка. Это делает Julia предпочтительным выбором для задач, требующих высокой скорости работы.

ЗАДАЧА 2. Решить уравнения методом Ньютона-Рафсона.

Метод Ньютона-Рафсона – итерационный алгоритм поиска корней уравнений вида f(x) = 0.

In [46]:
```julia
function newton_raphson(f, df, x0, tol=1e-7, max_iter=1000)
    # Инициализация начального значения
    x = x0

    # Главный итерационный цикл
    for i in 1:max_iter
        # Вычисляем значение функции в текущей точке
        fx = f(x)

        # Критерий остановки: достигнута требуемая точность
        if abs(fx) < tol
            return x  # Возвращаем найденный корень
        end

        # Вычисляем значение производной
        dfx = df(x)

        # Проверка на нулевую производную (метод не применим)
        if dfx == 0
            error("Производная равна нулю, метод не применим.")
```

```
        end

        # Основная формула метода Ньютона
        x = x - fx / dfx  # Получаем следующее приближение
    end

    # Если превышено максимальное число итераций
    error("Максимальное количество итераций достигнуто, решение не найдено
end

# Определяем тестовую функцию
f(x) = x^2 - 2

# Аналитическая производная функции
df(x) = 2x

# Начальное приближение
x0 = 1.0

# Вызываем метод Ньютона-Рафсона
root = newton_raphson(f, df, x0)


println("Корень уравнения x^2 - 2 = 0: ", root)
```

```
Корень уравнения x^2 - 2 = 0: 1.4142135623746899
```

ЗАДАЧА 3. Вывести на экран первую тысячу чисел Хэмминга. Число Хэмминга – это положительное целое число вида $2^i \cdot 3^j \cdot 5^k$ для некоторых неотрицательных целых чисел i, j, и k. Первое число Хэмминга равно $1 = 2^0 \cdot 3^0 \cdot 5^0$, второе число Хэмминга равно $2 = 2^1 \cdot 3^0 \cdot 5^0$, третье число Хэмминга равно $3 = 2^0 \cdot 3^1 \cdot 5^0$, четвертое число Хэмминга равно $4 = 2^2 \cdot 3^0 \cdot 5^0$, пятое число Хэмминга равно $5 = 2^0 \cdot 3^0 \cdot 5^1$.

In [53]:
```
# Функция проверки, является ли число числом Хемминга
function heming(n)
    # Делим число на 5, пока это возможно
    while n % 5 == 0
        n = n / 5
    end

    # Затем делим на 3, пока это возможно
    while n % 3 == 0
        n = n / 3
    end

    # Затем делим на 2, пока это возможно
    while n % 2 == 0
        n = n / 2
    end

    # Если после всех делений получили 1 - это число Хемминга
    return n == 1
end
```

```
# Основная программа для поиска первых k чисел Хемминга

# Ввод количества чисел Хемминга, которые нужно найти
k = parse(Int64, readline())

count = 0  # Счетчик найденных чисел Хемминга
j = 1      # Текущее проверяемое число

# Ищем числа Хемминга, пока не найдем нужное количество
while count < k
    if heming(j)
        # Если число Хемминга - увеличиваем счетчик и выводим
        count += 1
        println("№=", count, ", ", j)
    end
    j += 1  # Переходим к следующему числу
end
```

```
№=1, 1
№=2, 2
№=3, 3
№=4, 4
№=5, 5
№=6, 6
№=7, 8
№=8, 9
№=9, 10
№=10, 12
№=11, 15
№=12, 16
№=13, 18
№=14, 20
№=15, 24
№=16, 25
№=17, 27
№=18, 30
№=19, 32
№=20, 36
```

ЗАДАЧА 4. Найти сумму квадратов первых n (100 ≤ n ≤ 1000) чисел, кратных 7.

In [52]:
```
function sum_squares(n)
    first = 7
    sum = 0
    for i in 0:(n-1)
        num = first + i * 7
        sum += num ^ 2
    end
    return sum
end

println("Введите количество чисел, для которых нужно найти сумму квадратов
n = parse(Int, readline())
if 100 <= n <= 1000
    result = sum_squares(n)
```

```
    println("Сумма квадратов первых $n чисел, кратных 7: $result")
else
    println("Ошибка: n должно быть в диапазоне от 100 до 1000.")
end
```

```
Введите количество чисел, для которых нужно найти сумму квадратов (100 ≤ n
≤ 1000):
Сумма квадратов первых 250 чисел, кратных 7: 256741625
```

Глава 6 Модули

Модули в Julia позволяют организовывать код в отдельные, независимые блоки, что упрощает управление большими проектами, улучшает структуру кода и способствует его повторному использованию.

6.1 Создание модулей

Модуль создаётся с помощью ключевого слова **module**, и завершается ключевым словом **end**.

In [54]:
```
module NewModule
function hello(name)
    println("Привет, $name !")
end
end
```

Out[54]:
```
Main.NewModule
```

В этом примере создаётся модуль с именем MyModule, в котором определена функция hello.

Чтобы использовать модуль, его нужно импортировать с помощью ключевого слова **using** или **import**.

In [55]:
```
# если модуль в том же файле
using .NewModule
NewModule.hello("Юля")
```

```
Привет, Юля !
```

В случае, если нужный нам файл находится не там же, где и исполняемый файл Julia, необходимо будет создать файл с модулем, с расширением **.jl**, а так же указать путь к файлу следующим образом: **include("<путь к файлу><имя файла>.jl")**

In [1]:
```
include("/home/jusya/Hello_Julia.jl")  # Подключаем модуль
using .Hello  # Используем модуль
Hello.hello() # Вызов функции внутри модуля
```

```
Привет, мир !
```

Если вы хотите, чтобы какие-то функции или переменные из модуля были доступны напрямую, их нужно экспортировать с помощью ключевого слова **export**.

In [2]:
```
module Add
export add
function add(a, b)
    return a + b
end
end

using .Add
add(2, 3)
```

Out[2]: 5

Переменные, объявленные внутри модуля, не доступны напрямую извне, если они не экспортируются.

In [3]:
```
module Test1
export x1
x1 = 10
end

using .Test1
println(x1)
```

```
10
```

Если бы переменная не была экспортирована, доступ к ней был бы возможен только через полное имя модуля.

In [4]:
```
module Test2
x2 = 10
end

using .Test2
println(Test2.x2)
```

```
10
```

6.2 Модули внутри других модулей

В Julia можно создавать модули внутри других модулей.

In [5]:
```
module Test3
module Test4
    export hello
    function hello(name)
        println("Привет из Test4, ", name)
    end
```

```
end
end

using .Test3.Test4

hello("Алекс")
```

```
Привет из Test4, Алекс
```

In [6]:
```
module Test5

module Test6
    function hello(name)
        println("Привет из Test6, ", name)
    end
end

function hello(name)
    println("Привет из Test5, ", name)
end

end

using .Test5

Test5.hello("Алиса")
Test5.Test6.hello("Саша")
```

```
Привет из Test5, Алиса
Привет из Test6, Саша
```

6.3 Разрешение конфликтов имени

Если два модуля экспортируют одну и ту же функцию или переменную, можно использовать полное имя модуля для явного указания, какую именно функцию вы хотите использовать.

In [7]:
```
module A
    export f
    function f()
        println("Функция из модуля A")
    end
end

module B
    export f
    function f()
        println("Функция из модуля B")
    end
end

using .A
using .B
```

```
# Обращение через полное имя
A.f()
B.f()
```

```
Функция из модуля A
Функция из модуля B
```

Так же можно переименовать функцию при импорте с помощью ключевого слова **as**. Это позволяет избежать путаницы и легко различать функции с одинаковыми именами.

In [8]:
```
using .A
using .B

# Переименование функции из модуля A
using .A: f as fA

# Переименование функции из модуля B
using .B: f as fB

# Теперь можно использовать переименованные функции
fA()
fB()
```

```
Функция из модуля A
Функция из модуля B
```

6.4 Стандартные и пустые модули

В Julia есть **стандартные модули**, которые автоматически доступны, и **пустые модули**, которые можно создавать самостоятельно.

Стандартные модули, которые доступны всегда и не требуют загрузки:

- **Core** – содержит все функциональные возможности, встроенные в язык;
- **Base** – содержит базовый функционал;
- **Main** – это модуль, в котором выполняется код, когда вы запускаете программу или работаете в REPL.

В Julia модули по умолчанию включают **using Core** и **using Base**, но если эти стандартные определения не нужны, можно использовать baremodule – тогда модуль будет содержать только using Core. Такие модули имеют следующую структуру:

```
baremodule ModuleName

end
```

Если вы не хотите использовать даже Core, то структура такого модуля будет выглядеть следующим образом:

```
ModuleName =  Module(:ModuleName, false, false)
```

Пример. Создадим модуль, в котором с помощью макроса **@eval** (подробное описание макросов можно найти в главе 7) определим функции для выполнения базовых арифметических операций: сложения, вычитания, умножения и деления.

In [9]:
```
# Создаём модуль без использования Core и Base
NoCoreArithmetic = Module(:NoCoreArithmetic, false, false)

@eval NoCoreArithmetic begin
    # Функция для сложения двух чисел
    add(x, y) = $(+)(x, y)

    # Функция для вычитания
    subtract(x, y) = $(-)(x, y)

    # Функция для умножения
    multiply(x, y) = $(*)(x, y)

    # Функция для деления
    divide(x, y) = $(/)(x, y)
end
```

Out[9]:
```
divide (generic function with 1 method)
```

In [11]:
```
# Подключаем модуль
using .NoCoreArithmetic

# Используем функции, определённые в модуле
println("Сложение: ", NoCoreArithmetic.add(10, 5))
println("Вычитание: ", NoCoreArithmetic.subtract(10, 5))
println("Умножение: ", NoCoreArithmetic.multiply(10, 5))
println("Деление: ", NoCoreArithmetic.divide(10, 5))
```

```
Сложение: 15
Вычитание: 5
Умножение: 50
Деление: 2.0
```

Глава 7 Макросы

Макросы в Julia похожи на функции, но есть несколько отличий. Они определяются с помощью ключевого слова **macro** (вместо **function**) и вызываются с символом **@** перед именем макроса. В отличие от функций, для вызова макроса не обязательно использовать скобки – достаточно указать параметры через пробел.

```
@some_macro arg1 arg2
```

Макросы выполняются на этапе компиляции **до того, как код начнёт выполняться**. То есть, они изменяют сам код, а не его результаты. Когда вы

вызываете макрос, он преобразует код в абстрактное синтаксическое дерево (AST), и это позволяет манипулировать выражениями и структурой кода.

Важно, что макросы работают не с конкретными значениями переменных, а с **выражениями** (то есть, с самими кусками кода). Это позволяет создавать программы, которые могут изменять или генерировать код во время работы. Это называется **метапрограммированием**.

7.1 Встроенные макросы

Макрос **@time** измеряет время выполнения кода и сообщает о затраченном времени, а также о расходе памяти.

In [12]:
```
@time begin
    x = 0
    for i in 1:1000000
        x += i
    end
end
```

```
 0.071738 seconds (2.00 M allocations: 30.509 MiB, 32.57% gc time)
```

Макрос **@elapsed** – это упрощённая версия **@time**. Возвращает время выполнения выражения, но не выводит его в консоль.

In [14]:
```
t = @elapsed begin
    x = 0
    for i in 1:1000000
        x += i
    end
end
println("Время выполнения: ", t)
```

```
Время выполнения: 0.04051345
```

Макрос **@which** используется для того, чтобы узнать, какой метод будет вызван для конкретного выражения.

In [15]:
```
@which println("Hello, world!")
```

Out[15]: println(xs...) in Base at coreio.jl:4

Макрос **@show** используется для вывода значений выражений на экран. Это полезно для отладки, чтобы видеть результаты вычислений прямо в процессе выполнения.

In [20]:
```
x = 42
@show x
```

```
x = 42
```

Out[20]: 42

Макрос **@assert** используется для проверки условий. Если условие ложно, макрос генерирует ошибку и выводит проблемное выражение.

In [21]:
```
x = 5
@assert x > 0   # Условие истинно, ошибок не будет

@assert x < 0   # Ошибка
```

```
AssertionError: x < 0

Stacktrace:
 [1] top-level scope
   @ In[21]:4
```

7.2 Создание макросов

Теперь, когда мы рассмотрели основные встроенные макросы, давайте перейдем к созданию собственных макросов.

Синтаксис для создания макроса выглядит так:

```
macro имя_макроса(параметры)
    тело_макроса
end
```

In [24]:
```
macro hello(name)
    return :("Привет, " * $name)
end
```

Out[24]: @hello (macro with 1 method)

In [25]:
```
@hello("Мир")
```

Out[25]: "Привет, Мир"

Макросы возвращают код в виде выражений, используя символ `:`. Для вставки значений переменных в код макроса используется символ `$`.

In [26]:
```
macro add(a, b)
    println("Вычисляем сумму: ", a, " + ", b)
    return :( $a + $b )
end
```

Out[26]: @add (macro with 1 method)

In [27]:
```
@add 3 5
```

```
Вычисляем сумму: 3 + 5
```

Out[27]: 8

Глава 8 Инструменты графического представления результатов

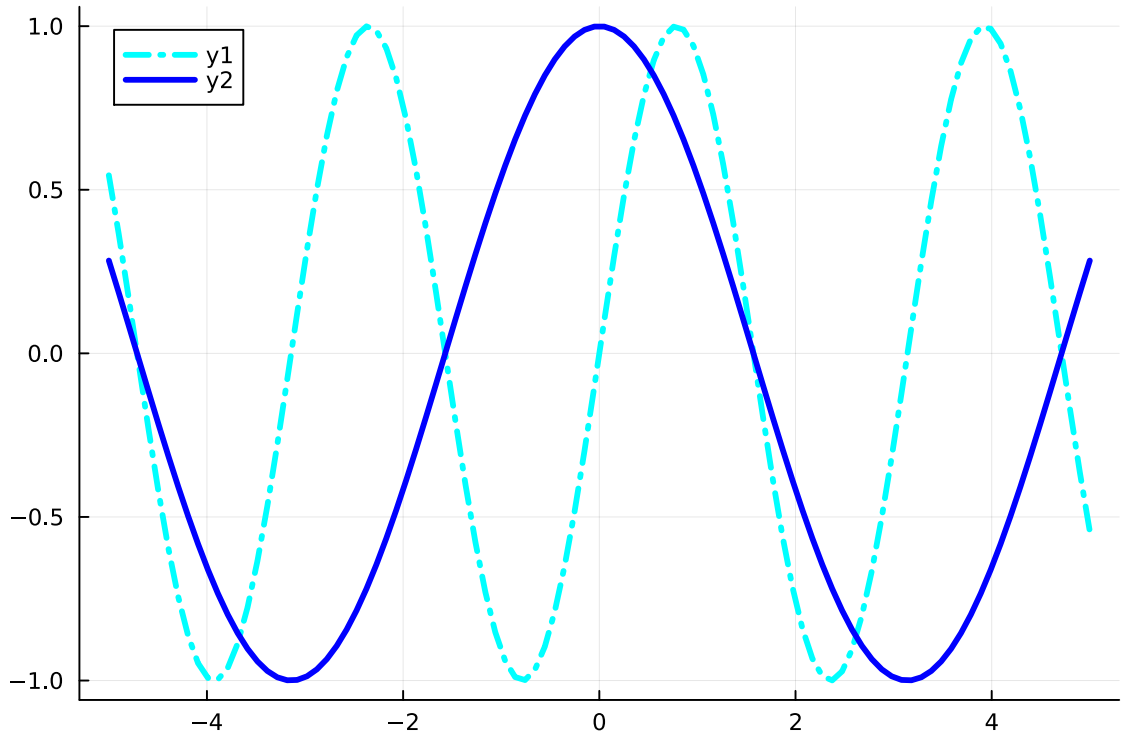
В Julia для визуализации данных часто используют пакет Plots. Этот пакет предоставляет единый интерфейс для множества различных бэкендов визуализации, таких как PyPlot (для работы с Matplotlib), GR (для работы с Gnuplot), PGFPlotsX и других.

Для получения дополнительной информации о пакете см. документацию.

Вот несколько простых примеров использования пакета Plots для создания различных типов графиков.

In [1]:
```
using Plots
x = range(-5, stop = 5, length = 100)
y = sin.(2 .* x)
z = cos.(x)
plot(x, y, color=:cyan, width=3, linestyle = :dashdot)
plot!(x, z, color=:blue, linestyle =:solid, width=3)
```

Out[1]:



Для построения графика функция plot имеет вид **plot(x,y,[type])** x – массив абсцисс, y – массив ординат, параметр type является строкой, в которой определяется цвет графика и тип линии, в строку type можно включать следующие параметры:

- **стиль линии** – строка, определяющая стиль линии;

Атрибут	Стиль линии
dashdot	Штрихпунктирная линия
dash	Штриховая линия
dot	Пунктирная линия
solid	Жирная линия

- **цвет** – строка, определяющая цвет линии;

Атрибут	Цвет
red	Красный
green	Зелёный
blue	Синий
cyan	Голубой
purple	Пурпурный
yellow	Жёлтый
black	Чёрный
white	Белый

- **width** – строка, определяющая толщину линии.

Кроме просмотра графиков на экране можно сохранить их в файл. В Plots это делается очень просто с помощью команд.

Сохранить график в файл в формате .png

```
savefig("myplot.png")
```
Или так

```
png("myplot.png")
```
Сохранить график в файл в формате .pdf

```
savefig("myplot.pdf")
```
Или так

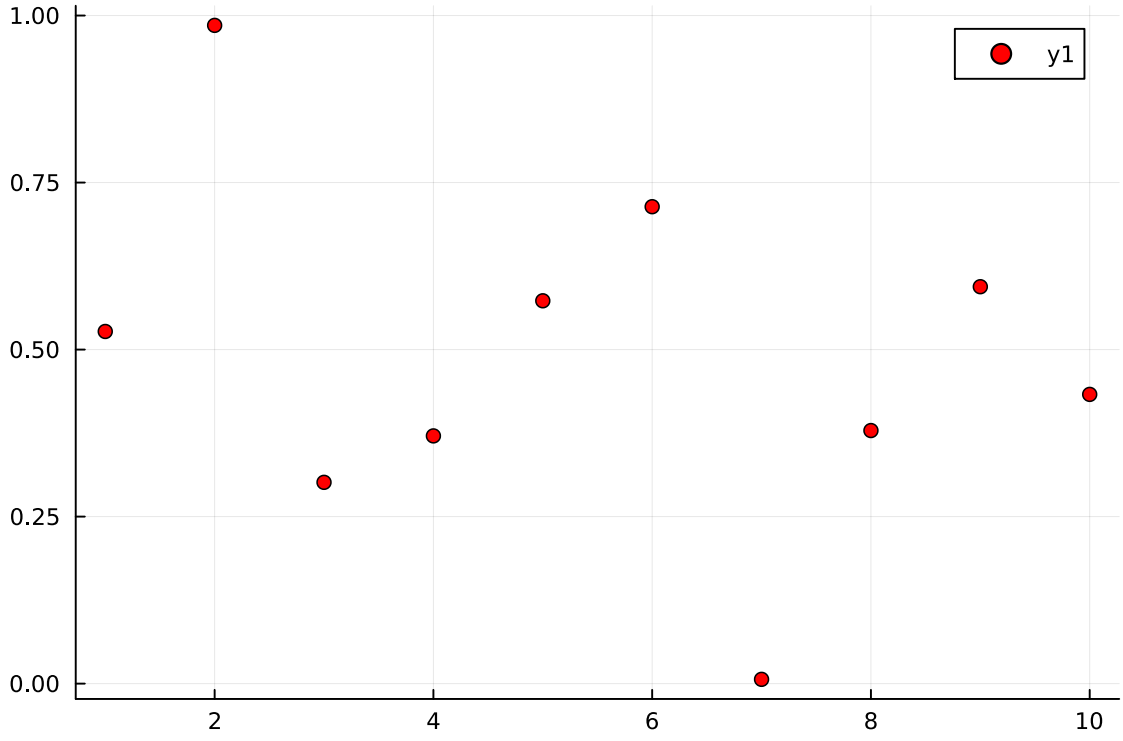
```
pdf("myplot.png")
```
Если нужно построить на одном графике несколько функций, это можно сделать с использованием вызова функций с восклицательным знаком.

Функция **scatter**, используемая в библиотеке Plots, позволяет отображаеть данные в виде точек на плоскости.

`scatter(x, y; [type])`
где x – массив абсцисс, y – массив ординат, type – дополнительные параметры (например, цвет точек, размер, маркеры и т. д.).

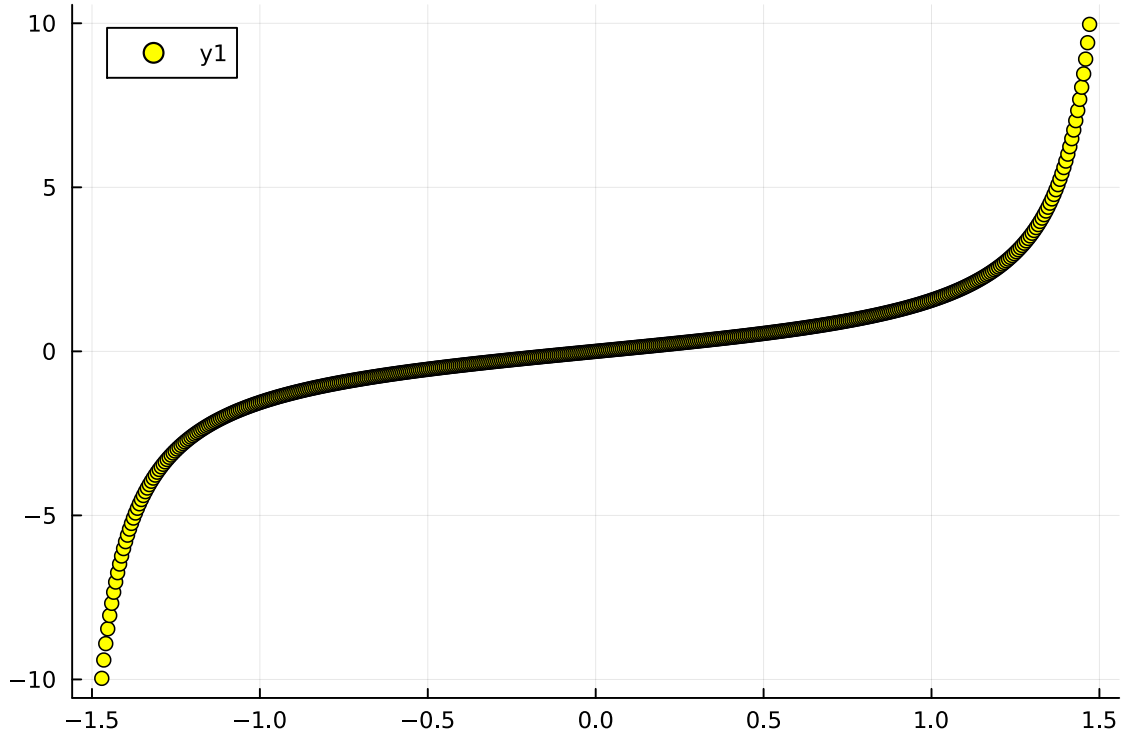
In [2]:
```
using Plots

x = 1:10
y = rand(10)

scatter(x, y, color=:red)
```

Out[2]:



In [3]:
```
using Plots

# Определяем диапазон значений для x
x = LinRange(-π/2 + 0.1, π/2 - 0.1, 500)


y = tan.(x)

scatter(x, y, color=:yellow)
```

Out[3]:

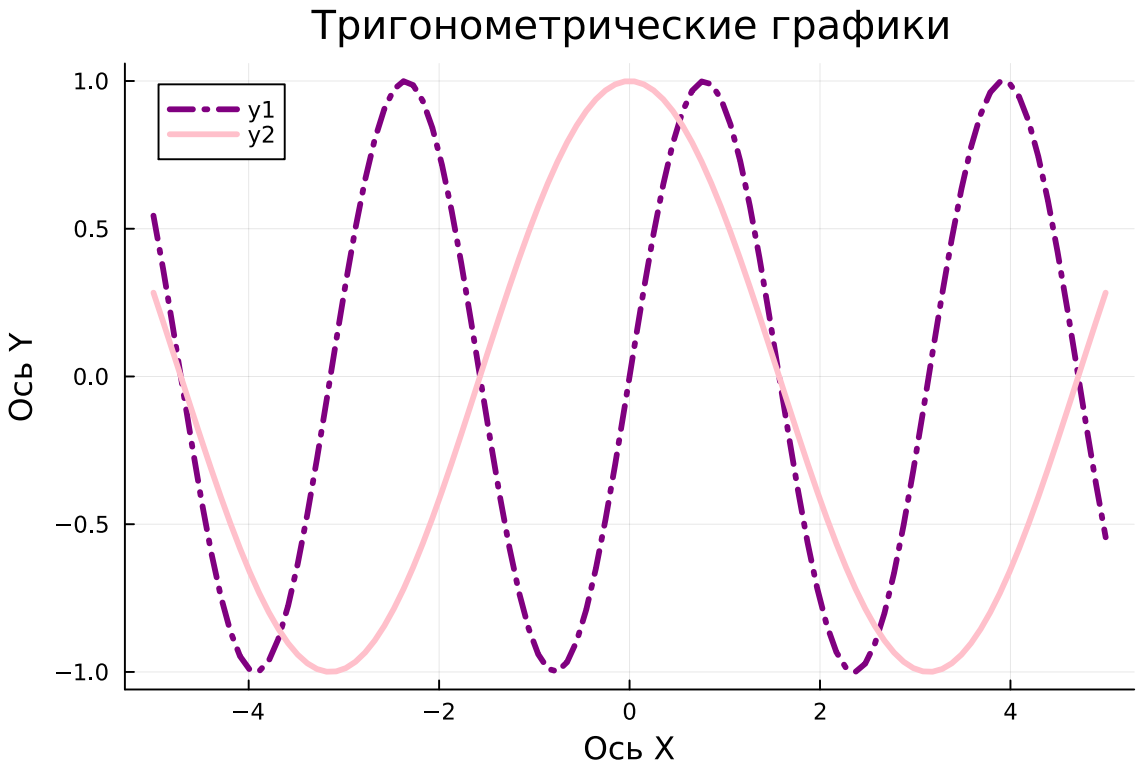


8.1 Заголовок, подписи, сетка, легенда

Заголовок и подписи к осям ставятся командами **title**, **xlabel** и **ylabel**, принимающими единственный аргумент – строку.

In [4]:
```
x = range(-5, stop = 5, length = 100)
y = sin.(2 .* x)
z = cos.(x)
plot(x, y, color=:purple, width=3, linestyle = :dashdot, title = "Тригоном
plot!(x, z, color=:pink, linestyle =:solid, width=3)
```

Out[4]:



Часто необходимо, чтобы в подписях на осях или легенде содержались верхние или нижние индексы, греческие буквы, различные значки и прочие математические символы. Для этого Plots имеет режим совместимости с командами LaTeX.В строке внутри пары символов **\$** можно вводить обычные формулы LaTeX ($ – начало и конец формулы)

Вывод легенды осуществить командой **legend()**. Чтобы переместить легенду или изменить её расположение, можно использовать:

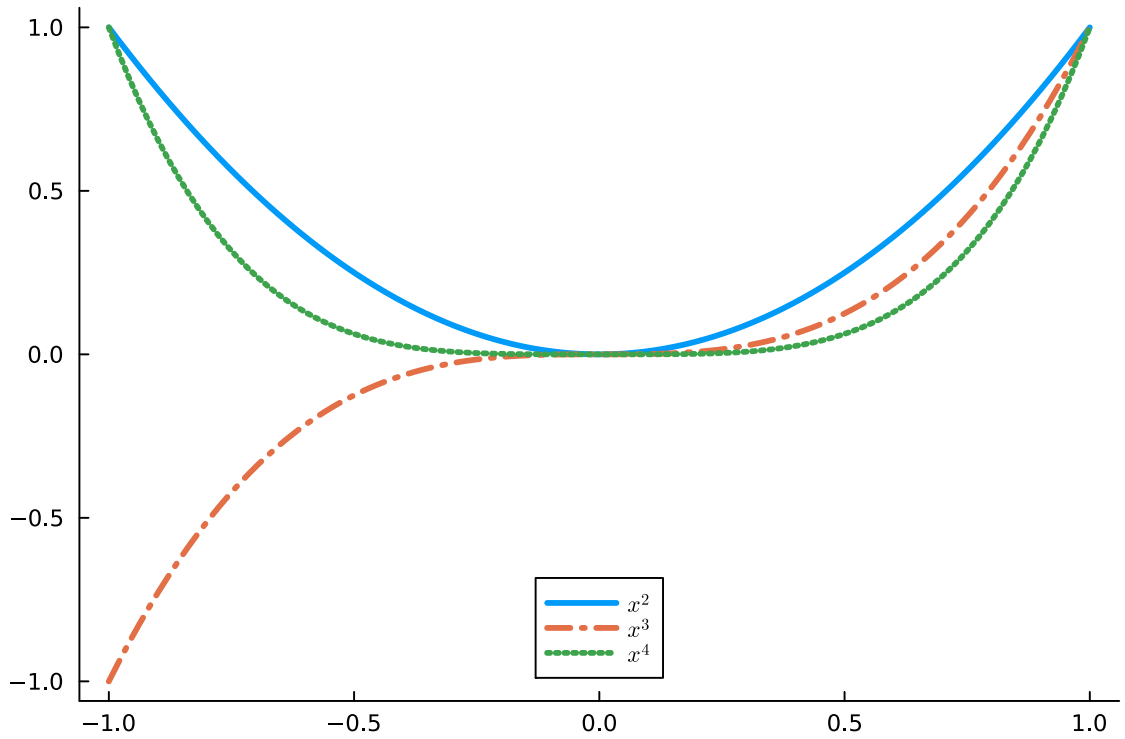
Параметры	Строка
best	наилучший
topright	сверху справа
topleft	сверху слева
bottomleft	снизу слева
bottomright	снизу справа
left	по центру слева
right	по центру справа
bottom	снизу по центру
top	сверху по центру
inside	по центру

Чтобы отключить легенду, можно использовать **legend=false**.

Ещё один часто встречающийся элемент графиков – сетка. Для получения сетки служит функция **grid** , у которой единственный логический аргумент (True – сетка будет, False – нет).

In [5]:
```
t = range(-1, stop = 1, length = 100)
x = t .^2
y = t .^3
z = t .^4
plot(t , x , label = "\$x^2\$", legend=:bottom, grid=:false, width=3)
plot!(t , y , label = "\$x ^3\$",  linestyle = :dashdot, width=3)
plot!(t , z , label = "\$x ^4\$",  linestyle = :dot, width=3)
```

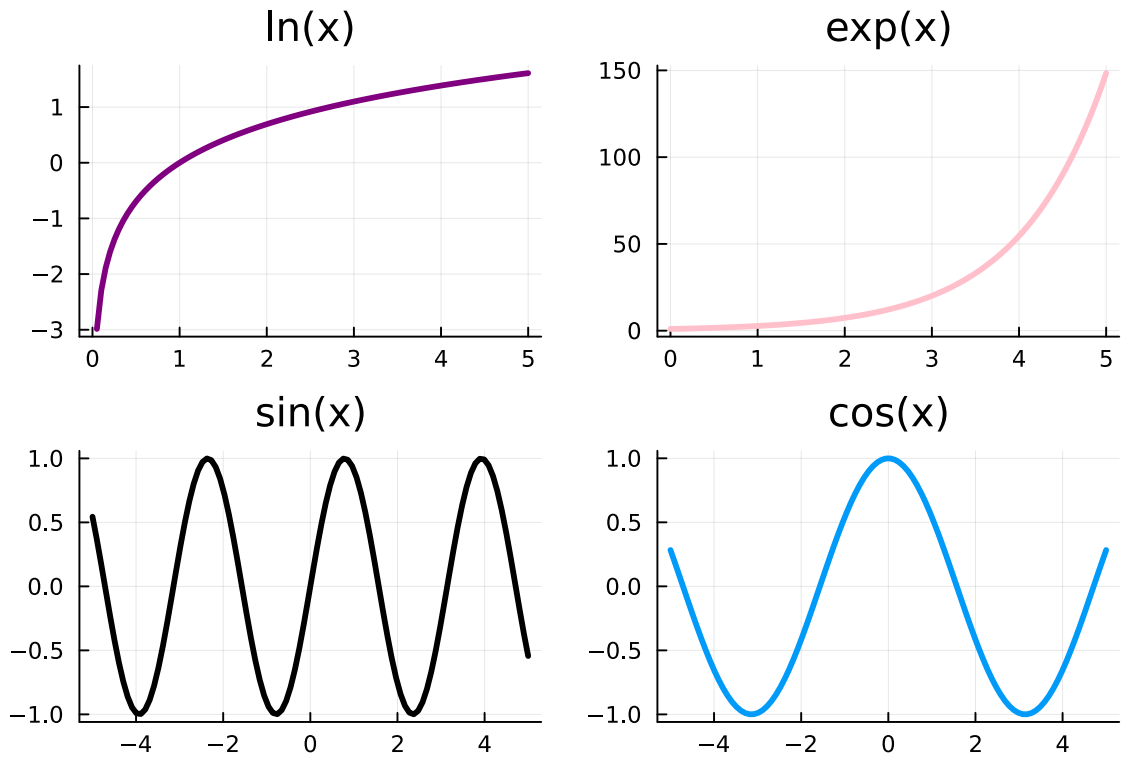
Out[5]:



8.2 Несколько графиков на одном полотне

In [6]:
```
x = range(0, stop = 5, length = 100)
y=log.(x)
p1 = plot(x,y, title="ln(x)", color=:purple, width=3, legend=false)
x = range(0, stop = 5, length = 100)
y=exp.(x)
p2 = plot(x,y, title="exp(x)", color=:pink, width=3, legend=false)
x = range(-5, stop = 5, length = 100)
y = sin.(2 .* x)
p3 = plot(x,y, title="sin(x)",  color=:black, width=3, legend=false)
x = range(-5, stop = 5, length = 100)
y = cos.(x)
p4 = plot(x,y, title="cos(x)", width=3, legend=false)
plot(p1, p2, p3, p4)
```

Out[6]:



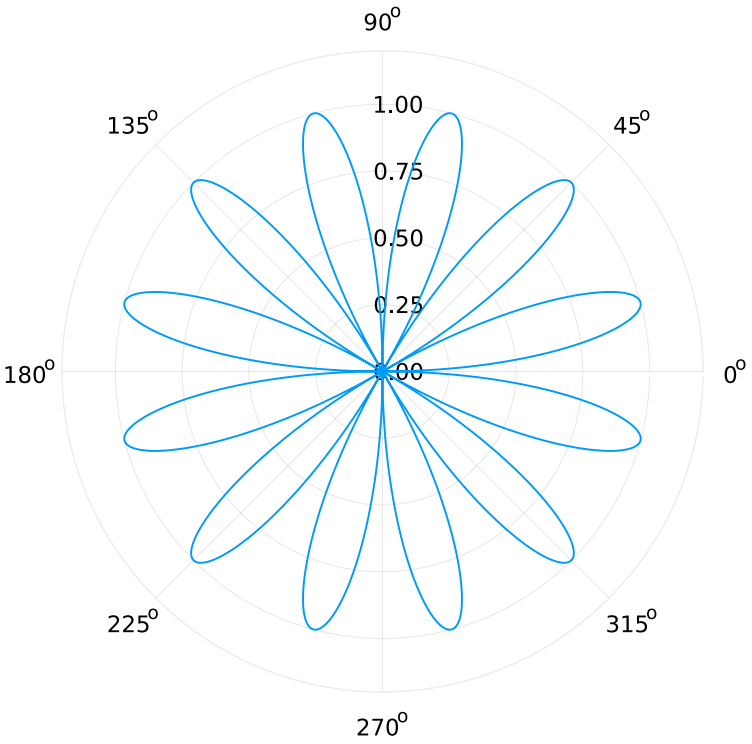
8.3. Графики в полярных координатах

Для построения графиков в полярных координатах нужно при вызове функции plot() указать proj=:polar.

Пример. Построить график следующей зависимости: r(t)=sin(6t), t∈[0;2π] в полярных координатах.

In [7]:
```
t=range(0,2pi, 1000)
rt=sin.(6.0.*t)
plot(t,rt,proj=:polar, legend=false)
```

Out[7]:



8.4 Трехмерные графики в Plots

Для построения трехмерных графиков в Julia с использованием библиотеки Plots можно использовать функцию **surface**, которую необходимо использовать внутри функции **plot**.

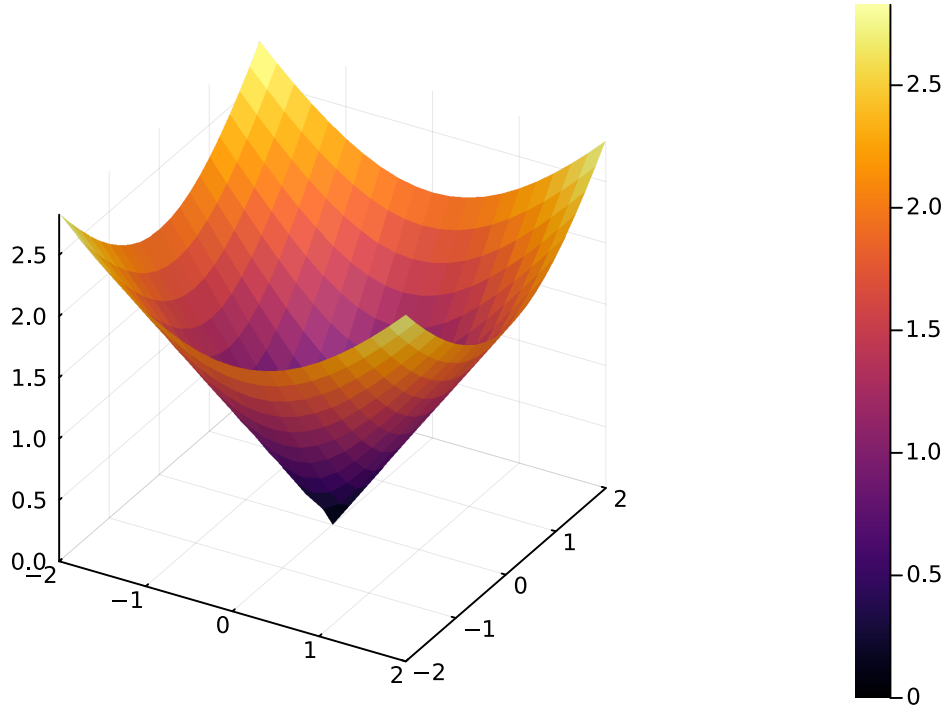
```
plot(surface(x, y, z))
```

Пример 1. Построить график функции:

$$z(x, y) = \sqrt{x^2 + y^2}$$

In [8]:
```julia
using Plots

# Создаем сетку значений x и y
x = -2:0.2:2
y = -2:0.2:2

# Создаем матрицы X, Y для сетки
X = [xi for xi in x, yi in y]
Y = [yi for xi in x, yi in y]

# Вычисляем значения Z = sqrt(x^2 + y^2)
Z = [sqrt(xi^2 + yi^2) for (xi, yi) in zip(X, Y)]#функция zip позволяет па
# Строим 3D график
plot(surface(X, Y, Z))
```

Out[8]:



Пример 2. Построить поверхность однополостного гиперболоида, уравнение которого задано в параметрическом виде:

$$x(u,v) = ch(u)cos(v),$$

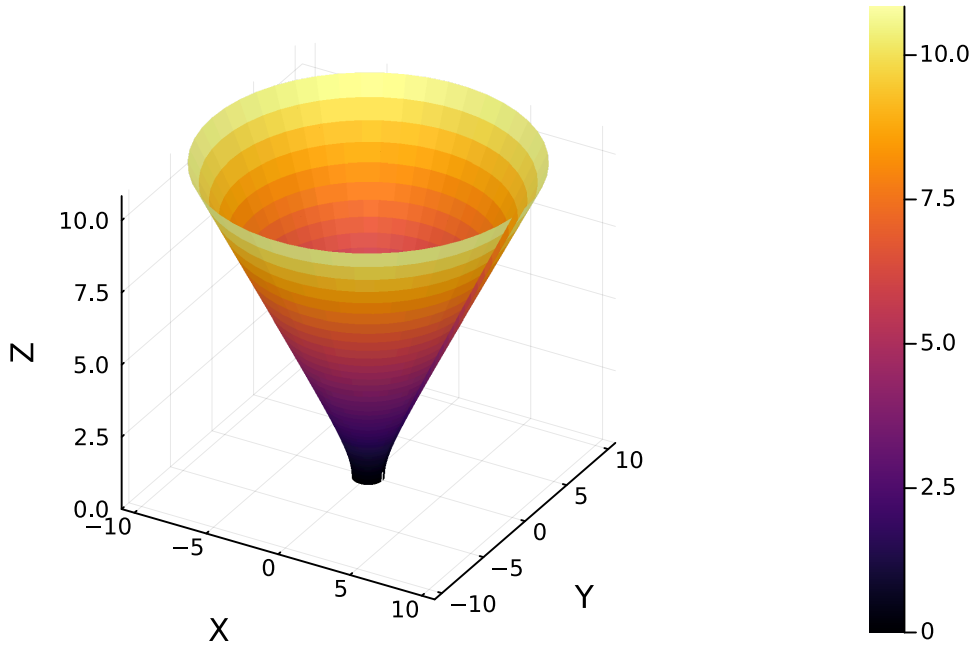
$$y(u,v) = ch(u)sin(v),$$

$$z(u,v) = sh(u).$$

In [9]:

```
using Plots

# Шаг для параметров
h = π / 50

# Формируем вектор u и v
u = 0:h:π
v = 0:2h:2π


x = [cosh(ui) * cos(vi) for ui in u, vi in v]
y = [cosh(ui) * sin(vi) for ui in u, vi in v]
z = [sinh(ui) for ui in u, vi in v]

# Строим 3D-график поверхности
plot(surface(x, y, z), xlabel="X", ylabel="Y", zlabel="Z", title="ГРАФИК О
```

Out[9]:



Пример 3. Построить поверхность сферы, уравнение которой задано в параметрическом виде:

$$x(u,v) = sin(u)cos(v),$$

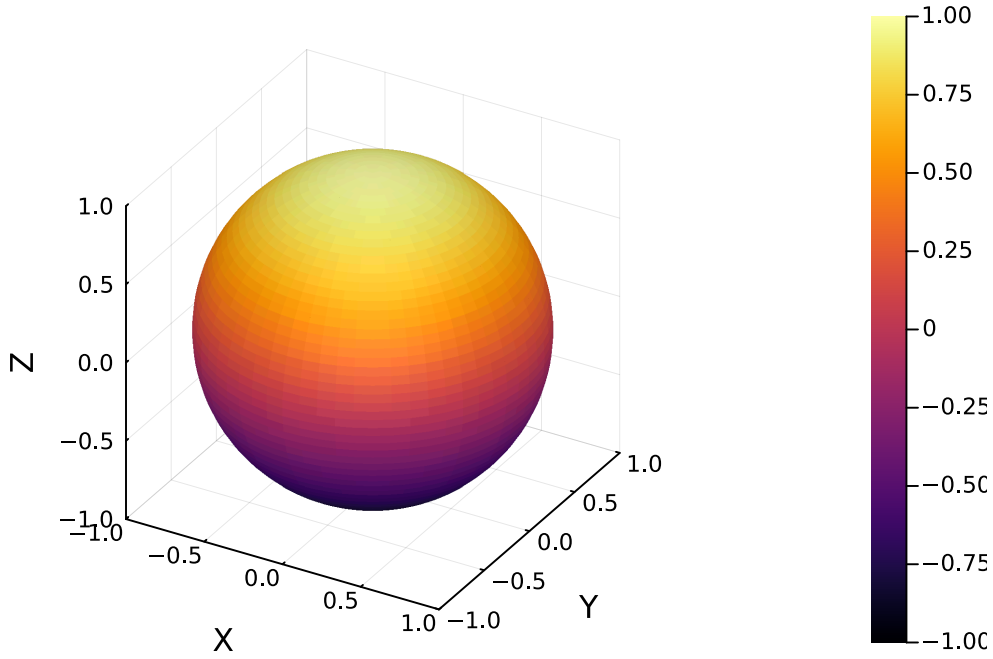
$$y(u,v) = sin(u)sin(v),$$

$$z(u,v) = cos(u).$$

In [10]:

```julia
using Plots

# Шаг для параметров
h = π / 60

# Формируем вектор u и v
u = 0:h:π
v = 0:2h:2π

x = [sin(ui) * cos(vi) for ui in u, vi in v]
y = [sin(ui) * sin(vi) for ui in u, vi in v]
z = [cos(ui) for ui in u, vi in v]

# Строим 3D-график поверхности
plot(surface(x, y, z), xlabel="X", ylabel="Y", zlabel="Z", title="СФЕРА")
```

Out[10]:

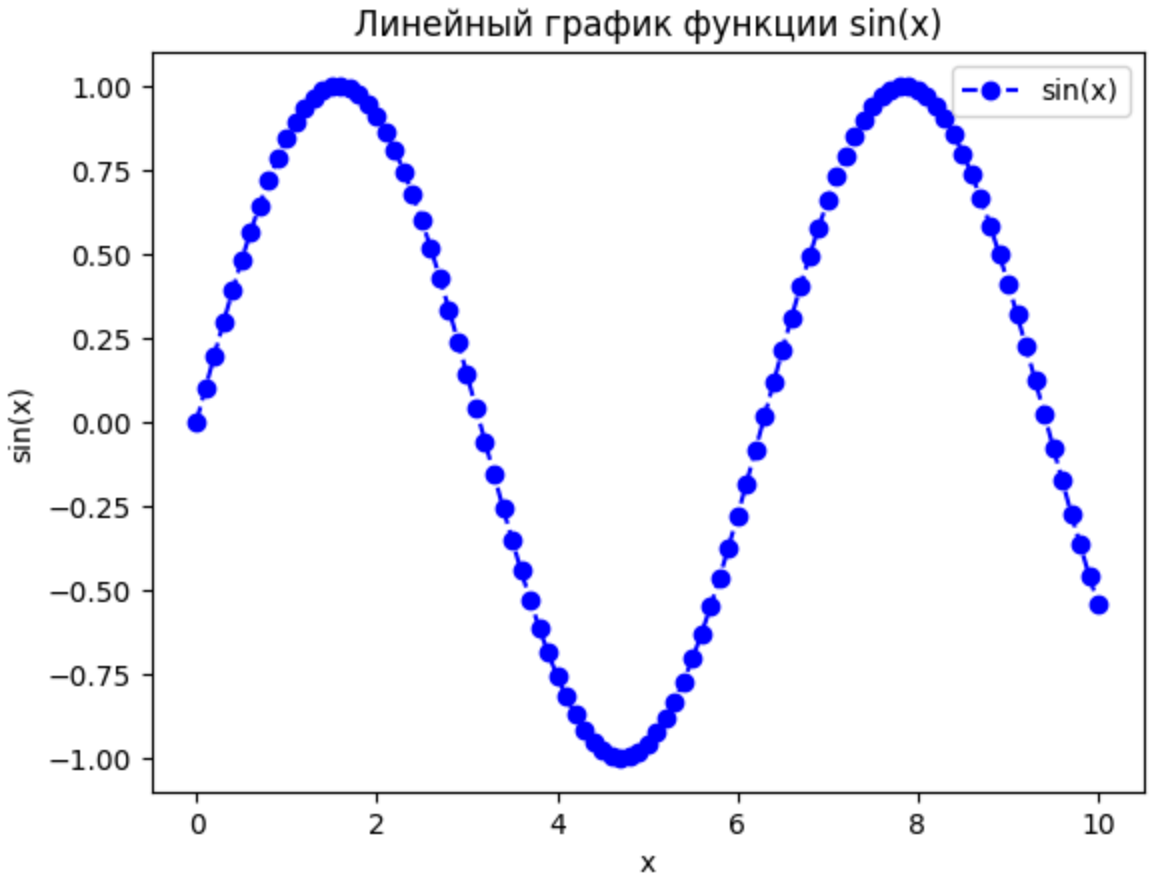


8.5 Библиотека PyPlot и ее использование с Plots

Для визуализации данных в Julia также существует библиотека PyPlot, которая предоставляет интерфейс для работы с популярной Python-библиотекой matplotlib.

In [2]:
```
using PyPlot

#Гграфик с одной линией
x = collect(0:0.1:10)
y = sin.(x)  # Применяем sin к каждому элементу массива x

figure()  # Создаем новый график
plot(x, y, label="sin(x)", color="blue", linestyle="--", marker="o")
xlabel("x")  # Подписываем ось X
ylabel("sin(x)")  # Подписываем ось Y
title("Линейный график функции sin(x)")  # Заголовок графика
legend()  # Легенда
show()  # Показываем график

#График с несколькими линиями
y2 = cos.(x)  # Применяем cos к каждому элементу x

figure()  # Создаем новый график
plot(x, y, label="sin(x)", color="blue")
plot(x, y2, label="cos(x)", color="red")
xlabel("x")
ylabel("y")
```

```
title("График sin(x) и cos(x)")
legend()
show()

#Гистограмма
data = randn(1000)  # Генерация случайных данных

figure()  # Создаем новый график
hist(data, bins=30, edgecolor="black")
xlabel("Значения")
ylabel("Частота")
title("Гистограмма случайных данных")
show()
```



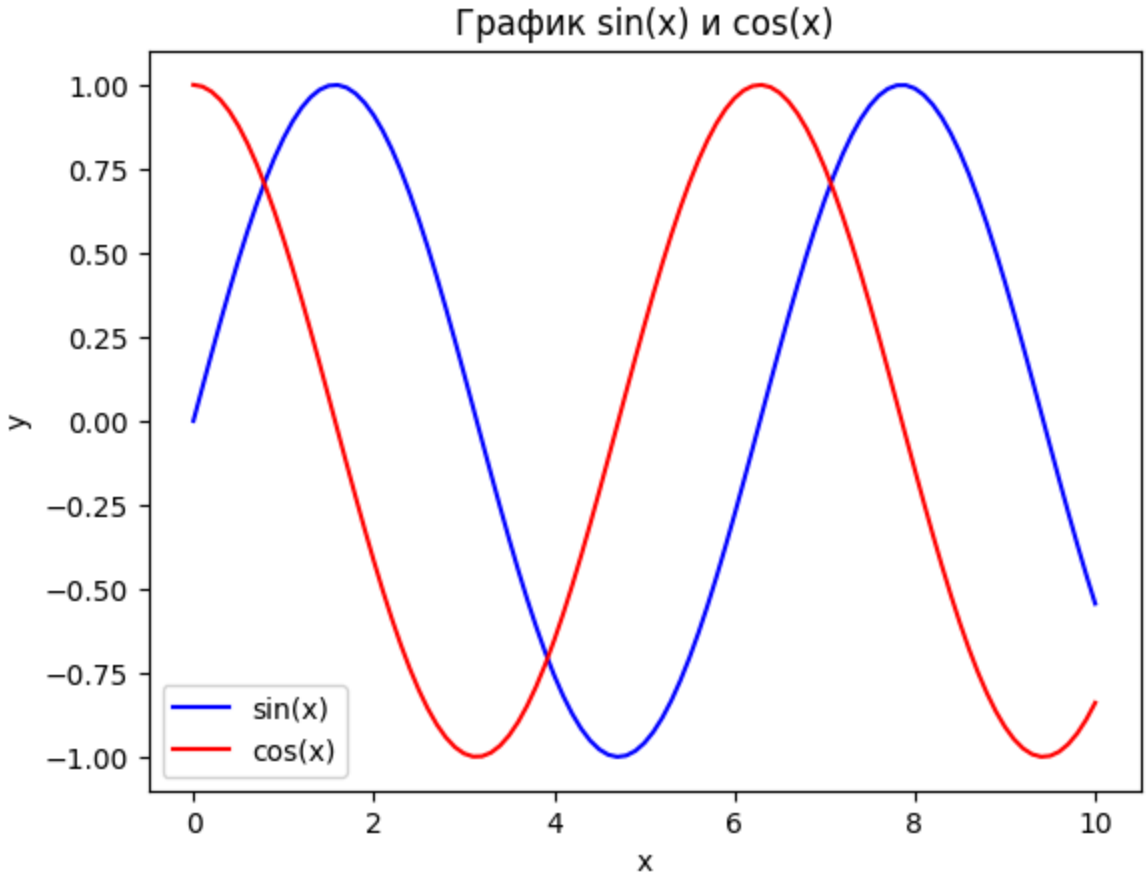
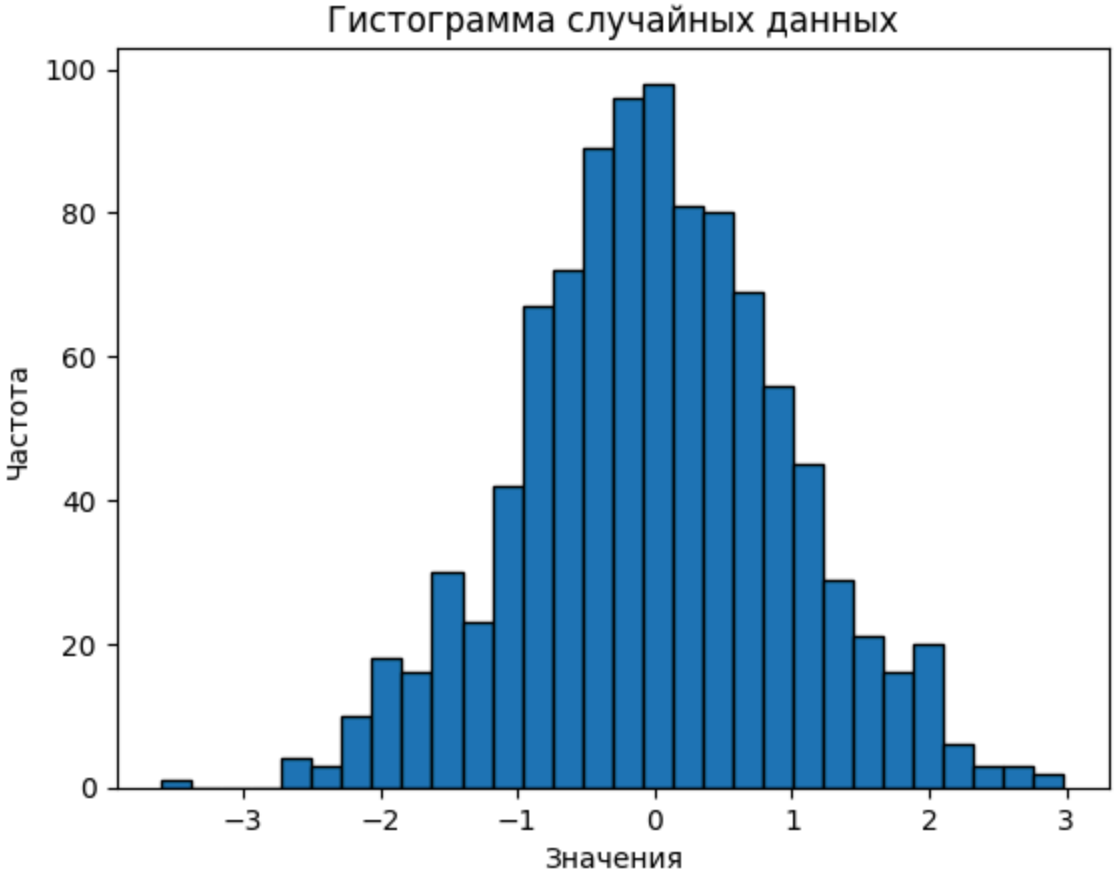


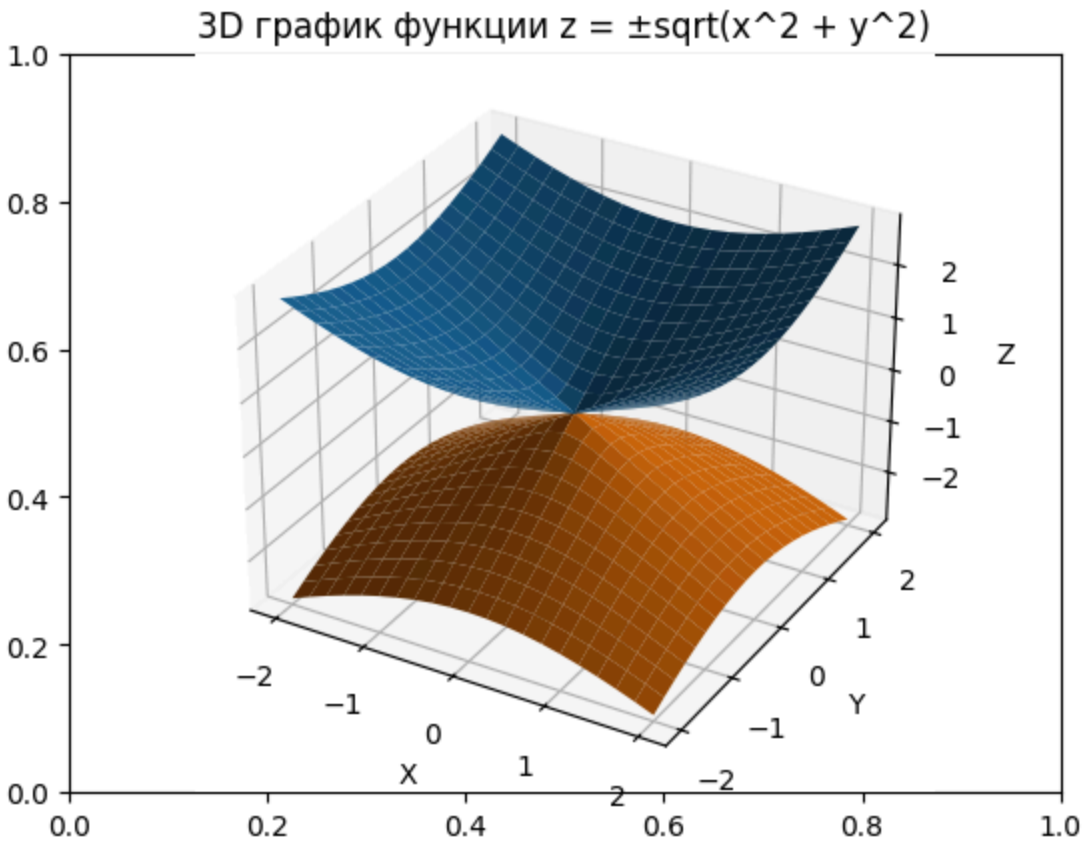
График sin(x) и cos(x)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
0
2
4
6
8
10
x
sin(x)
cos(x)


Гистограмма случайных данных
100
80
60
40
20
0
Частота
−3
−2
−1
0
1
2
3
Значения

В Julia для построения 3D-графиков с использованием библиотеки PyPlot используется функция **surf(x, y, z)**

Пример 4. Построить график функции $z(x, y) = \pm\sqrt{x^2 + y^2}$

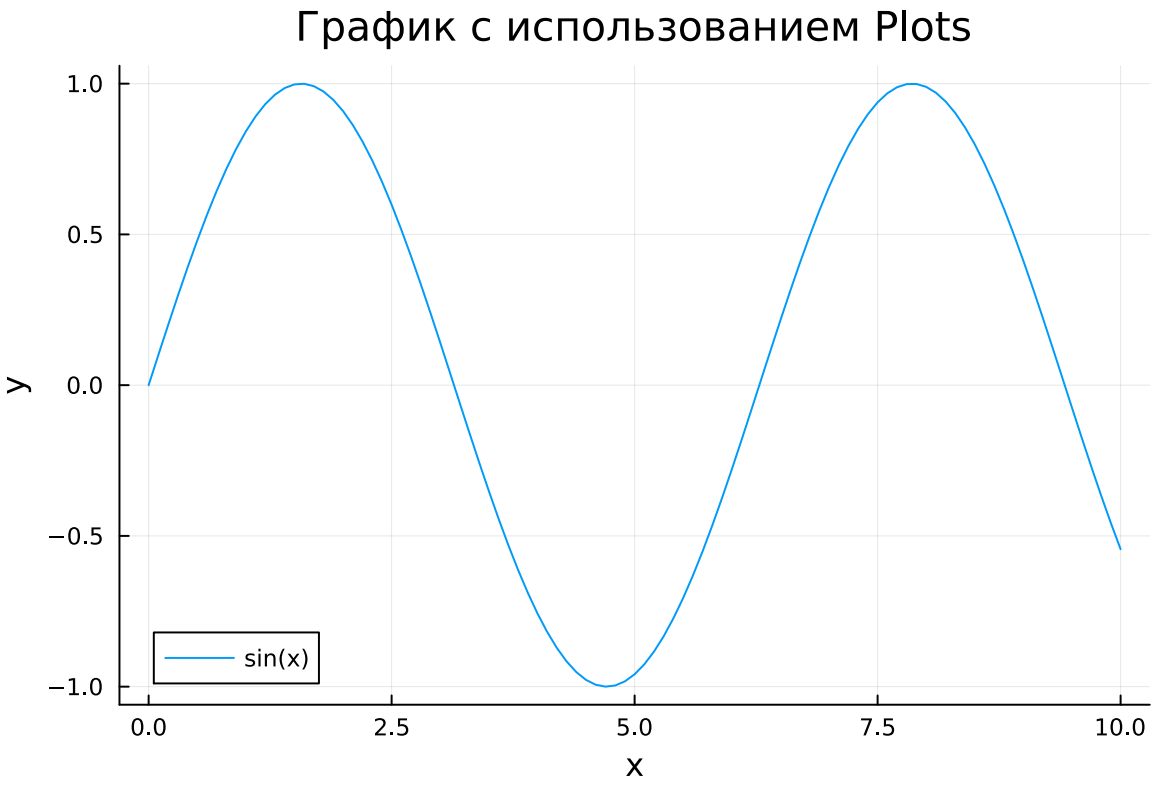
In [3]:
```
using PyPlot

# Генерация данных для x и y
x = -2:0.2:2
y = -2:0.2:2

# Создаем матрицы X, Y для сетки
X = [xi for xi in x, yi in y]
Y = [yi for xi in x, yi in y]

# Вычисляем значения Z и Z1 для функции z(x, y) = ±(sqrt(x^2 + y^2))
Z = [sqrt(xi^2 + yi^2)  for (xi, yi) in zip(X, Y)]  # Положительная часть
Z1 = [-sqrt(xi^2 + yi^2)  for (xi, yi) in zip(X, Y)]  # Отрицательная част

# Строим 3D поверхности для положительной и отрицательной части
surf(X, Y, Z)  # Положительная часть
surf(X, Y, Z1)  # Отрицательная часть

# Подписываем оси и добавляем заголовок
xlabel("X")
ylabel("Y")
zlabel("Z")
title("3D график функции z = ±sqrt(x^2 + y^2)")
```

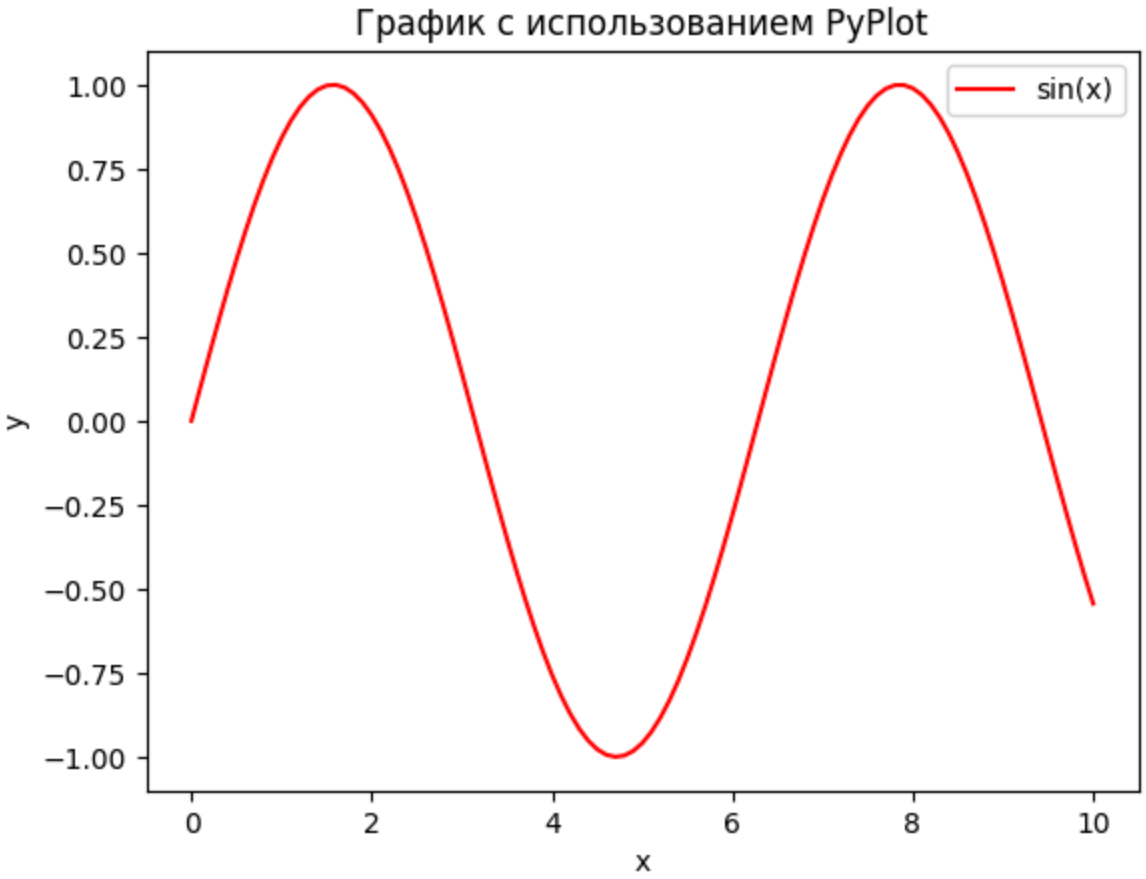


Out[3]: PyObject Text(0.5, 1.0, '3D график функции z = ±sqrt(x^2 + y^2)')

Чтобы использовать **Plots** и **PyPlot** в одном коде, важно помнить, что обе эти библиотеки предоставляют функции с одинаковыми именами. Чтобы избежать конфликта имен, Julia предоставляет возможность явно указывать, из какой библиотеки или модуля должна быть вызвана конкретная функция. Для этого достаточно написать имя библиотеки перед функцией, разделив их точкой. Например, если вы хотите использовать функцию plot из библиотеки Plots, пишите **Plots.plot()**, а для **PyPlot** – **PyPlot.plot()**.

In [8]:
```
using Plots   # Подключаем библиотеку Plots
using PyPlot  # Подключаем библиотеку PyPlot

x = 0:0.1:10
y = sin.(x)

# Строим график с использованием Plots.plot
p1 = Plots.plot(x, y, label="sin(x)", title="График с использованием Plots

# Строим график с использованием PyPlot.plot
p2 = PyPlot.plot(x, y, label="sin(x)", color="red")  # Явно указываем испо
PyPlot.xlabel("x")
PyPlot.ylabel("y")
PyPlot.title("График с использованием PyPlot")
PyPlot.legend()

# Отображаем оба графика
display(p1)  # Показываем график, построенный с использованием Plots
PyPlot.show()  # Показываем график, построенный с использованием PyPlot
```





Глава 9. Визуальное программирование

Визуальное программирование – это подход, при котором разработка программ происходит с использованием графических интерфейсов и визуальных элементов. Одним из популярных инструментов для создания таких интерфейсов является **GTK**, который поддерживает различные языки программирования, включая C, Python и Julia. В Julia для работы с GTK используется пакет **Gtk.jl**. Далее мы познакомимся с этим пакетом и его возможностями для создания оконных приложений, добавления кнопок, текстовых полей, панелей и других элементов управления, что позволяет легко создавать интерактивные интерфейсы прямо в Julia.

Для получения дополнительной информации о пакете см. документацию.

9.1 Создание окна (GtkWindow) и кнопки (GtkButton)

Начнём с очень простого примера, в котором создадим пустое окно размером 400x200 пикселей и добавим к нему кнопку. Для этого нам понадобятся несколько

ключевых функций из библиотеки **Gtk.jl: GtkWindow, GtkButton, push! и showall**. Давайте подробнее рассмотрим их использование и синтаксис:

1. **GtkWindow** – функция для создания окна с заданными размерами и заголовком;
2. **GtkButton** – функция для создания кнопки;
3. **push!** – функция для добавления виджетов в окно;
4. **showall** – функция для отображения окна и всех его компонентов.

In [5]:
```
using Gtk

# Создаём окно с заголовком "Моя первая программа на Gtk.jl" и размерами 4
win = GtkWindow("Моя первая программа на Gtk.jl", 400, 200)

# Создаём кнопку с текстом "Нажми на меня"
b = GtkButton("Нажми на меня")

# Добавляем кнопку в окно
push!(win,b)

# Показываем все элементы окна
showall(win)
```

Out[5]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Моя пе
рвая программа на Gtk.jl", role=NULL, resizable=TRUE, modal=FALSE, window
-position=GTK_WIN_POS_NONE, default-width=400, default-height=200, destro
y-with-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=
NULL, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FA
LSE, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-
on-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WE
ST, transient-for, attached-to, has-resize-grip, resize-grip-visible, app
lication, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonic
s-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

Теперь мы расширим пример, чтобы кнопка действительно могла что-то делать.Чтобы сделать кнопку интерактивной (например, вызвать функцию при нажатии), можно добавить обработчик событий с помощью функции **signal_connect**. Синтаксис: **signal_connect(handler, widget, signal_name)**

Параметры:

1. **handler** – функция (или метод), которая будет вызвана при наступлении события. Эта функция должна принимать как минимум один аргумент – сам

виджет, к которому привязан сигнал (**виджет** – это элемент интерфейса для взаимодействия с пользователем, например кнопка, текстовое поле);
2. **widget** – виджет, к которому привязывается обработчик событий;
3. **signal_name** – строка, указывающая на имя события, на которое мы хотим реагировать. Например, "clicked" для кнопки, "changed" для текстового поля, "destroy" для окна.

Напишем программу, которая при нажатии на кнопку будет выводить сообщение в консоль.

In [6]:
```
using Gtk

win = GtkWindow("Пример подключения обработчика событий", 400, 200)

b = GtkButton("Нажми на меня")
push!(win, b)

# Определяем функцию-обработчик, которая будет вызвана при нажатии на кноп
function on_button_clicked(b)
    # Эта строка выводит сообщение в консоль, когда кнопка нажата
    println("Кнопка была нажата")
end

# Связываем сигнал "clicked" (событие нажатия на кнопку) с функцией-обрабо
# signal_connect устанавливает, что при нажатии на кнопку b будет вызвана
signal_connect(on_button_clicked, b, "clicked")

showall(win)
```

Out[6]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
подключения обработчика событий", role=NULL, resizable=TRUE, modal=FALSE,
window-position=GTK_WIN_POS_NONE, default-width=400, default-height=200,
destroy-with-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon
-name=NULL, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-h
int=FALSE, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE,
focus-on-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NO
RTH_WEST, transient-for, attached-to, has-resize-grip, resize-grip-visibl
e, application, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mn
emonics-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
Кнопка была нажата
```

Функция **set_gtk_property!** в Gtk.jl используется для динамического изменения внешнего вида и поведения различных Gtk-виджетов во время работы приложения. Все виджеты в Gtk имеют набор свойств, которые можно изменять,

например, текст, цвет, размер и поведение. Синтаксис функции **set_gtk_property!**: set_gtk_property!(widget, property, value)

Параметры:

1. **widget** – это Gtk-виджет, для которого вы хотите установить свойство;
2. **property** – это имя свойства, которое вы хотите изменить (:label, :text, :visible и т. д.);
3. **value** – значение, которое вы хотите установить для этого свойства. Тип значения зависит от типа свойства.

Пример 1. Изменение текста в кнопке.

Каждому виджету можно установить различные свойства. Например, чтобы изменить текст кнопки, используем свойство **:label**.

In [7]:
```
using Gtk


win = GtkWindow("Изменение текста на кнопке", 400, 200)
b = GtkButton("Нажми на меня")

push!(win, b)

function on_button_clicked(b)
    # Изменяем текст на кнопке с помощью set_gtk_property!
    set_gtk_property!(b, :label, "Нажато")
end

signal_connect(on_button_clicked, b, "clicked")


showall(win)
```

Out[7]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Измене
ние текста на кнопке", role=NULL, resizable=TRUE, modal=FALSE, window-pos
ition=GTK_WIN_POS_NONE, default-width=400, default-height=200, destroy-wi
th-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NUL
L, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALS
E, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on
-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WES
T, transient-for, attached-to, has-resize-grip, resize-grip-visible, appl
ication, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics
-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

Пример 2. Изменение видимости виджета.

Некоторые свойства управляют видимостью или состоянием виджетов. Например, чтобы скрыть или показать виджет, можно использовать свойство **:visible**.

In [8]:
```
using Gtk

win = GtkWindow("Пример наглядности", 400, 200)


b = GtkButton("Нажми на меня")
push!(win, b)

function on_button_clicked(b)
    set_gtk_property!(b, :visible, false)  # Скрываем кнопку
end

signal_connect(on_button_clicked, b, "clicked")

showall(win)
```

Out[8]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
наглядности", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK
_WIN_POS_NONE, default-width=400, default-height=200, destroy-with-parent
=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen,
type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-page
r-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, d
ecorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-
for, attached-to, has-resize-grip, resize-grip-visible, application, is-a
ctive=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALS
E, focus-visible=FALSE, is-maximized=FALSE)
```

Пример 3. Изменение размера окна.

Размер окна можно установить с помощью свойств **:width_request** и **:height_request**.

In [10]:
```
using Gtk

win = GtkWindow("Пример размера")


b = GtkButton("Нажми на меня")
push!(win, b)
```

```
function on_button_clicked(b)
    set_gtk_property!(win, :width_request, 500)  # Устанавливаем ширину ок
    set_gtk_property!(win, :height_request, 500) # Устанавливаем высоту ок
end

signal_connect(on_button_clicked, b, "clicked")


showall(win)
```

Out[10]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
размера", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN
_POS_NONE, default-width=-1, default-height=-1, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

Пример 4. Изменение состояния кнопки (активна/неактивна)

Можно управлять состоянием кнопки, например, сделать её неактивной с помощью свойства **:sensitive**.

In [11]:
```
using Gtk

win = GtkWindow("Неактивная кнопка")


b = GtkButton("Нажми на меня")
push!(win, b)

function on_button_clicked(b)
    set_gtk_property!(b, :sensitive, false)# Делаем кнопку неактивной (нев
end

signal_connect(on_button_clicked, b, "clicked")


showall(win)
```

Out[11]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Неакти
вная кнопка", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK
_WIN_POS_NONE, default-width=-1, default-height=-1, destroy-with-parent=F
ALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, t
ype-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager
-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, de
corated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-f
or, attached-to, has-resize-grip, resize-grip-visible, application, is-ac
tive=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALS
E, focus-visible=FALSE, is-maximized=FALSE)
```

9.2 Макеты

Обычно в приложение требуется добавить более одного виджета. Для этого GTK предоставляет несколько виджетов для организации макета. Вместо использования точного позиционирования, виджеты макета в GTK используют подход, при котором виджеты выравниваются в контейнерах, таких как коробки и таблицы.

9.2.1 GtkBox

GtkBox – это один из самых простых и популярных виджетов для создания макета в GTK. Он позволяет организовать другие виджеты в порядке, либо по горизонтали, либо по вертикали. GtkBox упрощает расположение элементов в интерфейсе, избавляя от необходимости вручную управлять позиционированием.

Можно указать, как именно вы хотите выравнивать виджеты внутри GtkBox – по горизонтали или по вертикали:

- **GtkBox(:h)** – горизонтальное расположение;
- **GtkBox(:v)** – вертикальное расположение.

Вы можете управлять тем, как виджеты растягиваются внутри GtkBox с помощью параметров expand, fill и padding:

- **expand** – указывает, будет ли виджет расширяться, чтобы заполнить доступное пространство;

- **fill** – указывает, будет ли виджет растягиваться по размеру в том направлении, в котором он выровнен (горизонтально или вертикально);
- **padding** – добавляет отступы вокруг виджета.

Виджеты добавляются в GtkBox в определённом порядке, и порядок их добавления влияет на то, в какой последовательности они будут отображаться (слева направо или сверху вниз).

In [14]:
```
using Gtk

# Создаем окно
win = GtkWindow("GtkBox пример", 400, 200)

# Создаем горизонтальную коробку (выравнивание по горизонтали)
box = GtkBox(:h)  # :h означает горизонтальное выравнивание

# Создаем несколько кнопок
button1 = GtkButton("Кнопка 1")
button2 = GtkButton("Кнопка 2")
button3 = GtkButton("Кнопка 3")

# Добавляем кнопки в коробку
push!(box, button1, button2, button3)

# Добавляем коробку в окно
push!(win, box)

# Показываем окно
showall(win)
```

Out[14]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="GtkBox
пример", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_
POS_NONE, default-width=400, default-height=200, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

После того как виджеты были добавлены в GtkBox, мы можем обратиться к каждому из них по индексу и получить их свойства.

In [15]:
```
length(box)# Получаем количество виджетов в GtkBox
```

Out[15]:
```
3
```

In [16]:
```
get_gtk_property(box[1], :label, String)  # Получаем текст первой кнопки
```

Out[16]:
```
"Кнопка 1"
```

In [17]:
```
get_gtk_property(box[2], :label, String)  # Получаем текст второй кнопки
```

Out[17]:
```
"Кнопка 2"
```

Предположим, что вы хотите, чтобы кнопка "Кнопка 3" заполнила доступное пространство, а между кнопками был отступ. Мы можем настроить свойства **expand** и **spacing**.

In [18]:
```
using Gtk


win = GtkWindow("GtkBox пример", 400, 200)

# Создаем горизонтальную коробку (выравнивание по горизонтали)
box = GtkBox(:h)  # :h означает горизонтальное выравнивание

# Создаем несколько кнопок
button1 = GtkButton("Кнопка 1")
button2 = GtkButton("Кнопка 2")
button3 = GtkButton("Кнопка 3")

# Добавляем кнопки в коробку
push!(box, button1, button2, button3)

# Добавляем коробку в окно
push!(win, box)

# Делаем кнопку расширяемой
set_gtk_property!(box, :expand, button3, true)

# Добавляем отступы между виджетами в GtkBox
set_gtk_property!(box, :spacing, 10)

# Показываем окно
showall(win)
```

Out[18]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="GtkBox
пример", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_
POS_NONE, default-width=400, default-height=200, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

9.2.2 GtkButtonBox

Если вы хотите, чтобы кнопки в GtkBox имели равный размер и правильное выравнивание, лучше использовать **GtkButtonBox**. Этот виджет предназначен специально для создания горизонтальных или вертикальных групп кнопок.

In [19]:
```
using Gtk

win = GtkWindow("Пример GtkButtonBox", 400, 200)

# Создаем GtkButtonBox с горизонтальным расположением
hbox = GtkButtonBox(:h)  # :h - это горизонтальное расположение кнопок

# Создаем кнопки
button1 = GtkButton("Кнопка 1")
button2 = GtkButton("Кнопка 2")
button3 = GtkButton("Кнопка 3")

# Добавляем кнопки в GtkButtonBox
push!(win, hbox)
push!(hbox, button1)
push!(hbox, button2)
push!(hbox, button3)

showall(win)
```

Out[19]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
GtkButtonBox", role=NULL, resizable=TRUE, modal=FALSE, window-position=GT
K_WIN_POS_NONE, default-width=400, default-height=200, destroy-with-paren
t=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, scree
n, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-p
ager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRU
E, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transi
ent-for, attached-to, has-resize-grip, resize-grip-visible, application,
is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=
FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

9.2.3 GtkGrid

GtkGrid – это макет, который позволяет размещать другие виджеты в виде сетки с строками и столбцами.

В GtkGrid можно задавать как обычное размещение виджетов, так и задавать их размеры и отступы между ними. Этот макет используется для построения интерфейсов с более сложным расположением элементов, чем, например, вертикальные или горизонтальные контейнеры.

Для добавления виджетов в сетку мы используем индексы строк и столбцов.

In [20]:
```
using Gtk

win = GtkWindow("Пример GtkGrid")
# Создаем сетку
g = GtkGrid()

# Создаем несколько кнопок
button1 = GtkButton("Кнопка 1")
button2 = GtkButton("Кнопка 2")
button3 = GtkButton("Кнопка 3")

# Размещение кнопок в сетке
g[1, 1] = button1  # Кнопка 1 в первом столбце и в первой строке
g[2, 1] = button2  # Кнопка 2 во втором столбце и в первой строке
g[1, 2] = button3  # Кнопка 3 в первом столбце и во второй строке


# Отображаем окно с кнопками
push!(win, g)
showall(win)
```

Out[20]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
GtkGrid", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN
_POS_NONE, default-width=-1, default-height=-1, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

Свойства GtkGrid:

- **row_spacing** – промежуток между строками;
- **column_spacing** – промежуток между столбцами;
- **column_homogeneous** – если установлено в true, все столбцы будут одинаковой ширины;
- **row_homogeneous** – если установлено в true, все строки будут одинаковой высоты.

In [21]:
```
using Gtk

win = GtkWindow("Пример GtkGrid")
# Создаем сетку
g = GtkGrid()

# Создаем несколько кнопок
button1 = GtkButton("Кнопка 1")
button2 = GtkButton("Кнопка 2")
button3 = GtkButton("Кнопка 3")
button4 = GtkButton("Кнопка 4")

# Размещение виджетов в сетке
g[1, 1] = button1  # button1 в ячейке (1, 1)
g[2, 1] = button2  # button2 в ячейке (2, 1)
g[1, 2] = button3  # button3 в ячейке (1, 2)
g[2, 2] = button4  # button4 в ячейке (2, 2)

# Настройка свойств GtkGrid
set_gtk_property!(g, :row_spacing, 10)         # Промежуток между строками
set_gtk_property!(g, :column_spacing, 15)      # Промежуток между столбцам
set_gtk_property!(g, :column_homogeneous, true) # Все столбцы одинаковой ш
set_gtk_property!(g, :row_homogeneous, true)    # Все строки одинаковой вы

# Добавляем сетку в окно
```

```
push!(win, g)


showall(win)
```

Out[21]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
GtkGrid", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN
_POS_NONE, default-width=-1, default-height=-1, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

Можно задать несколько ячеек для одного виджета.

In [22]:
```
using Gtk

win = GtkWindow("Пример GtkGrid")
# Создаем сетку
g = GtkGrid()

# Создаем несколько кнопок
button1 = GtkButton("Кнопка 1")
button2 = GtkButton("Кнопка 2")
button3 = GtkButton("Кнопка 3")

# Размещение кнопок в сетке
g[1:2, 1] = button1  # button1 займет 2 столбца в 1 строке
g[3, 1:3] = button2  # button2 займет 3 строки в 3 столбце
g[1:3, 4] = button3  # button3 займет 3 столбца в 4 строке

# Настроим расстояние между столбцами и строками
set_gtk_property!(g, :column_spacing, 10)
set_gtk_property!(g, :row_spacing, 10)

# Отображаем окно с кнопками
push!(win, g)
showall(win)
```

Out[22]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
GtkGrid", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN
_POS_NONE, default-width=-1, default-height=-1, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

9.3 Текстовые поля

В GTK на языке Julia два наиболее распространённых виджета для работы с текстом – это **GtkLabel** и **GtkEntry**. Оба виджета используются для работы с текстом, но их функциональность различна.

9.3.1 GtkLabel

GtkLabel – метка (текст, который не редактируется)

GtkLabel представляет собой виджет для отображения текста, который не может быть отредактирован пользователем. Это полезно, например, для вывода статического текста, инструкций, заголовков и других элементов, которые должны быть видны, но не изменяемы.

In [23]:
```julia
using Gtk

# Создаём окно
win = GtkWindow("Пример GtkLabel", 400, 200)

# Создаём метку с текстом
label = GtkLabel("Это метка с текстом!")

push!(win, label)
showall(win)
```

Out[23]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
GtkLabel", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WI
N_POS_NONE, default-width=400, default-height=200, destroy-with-parent=FA
LSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, ty
pe-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-
hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, dec
orated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-fo
r, attached-to, has-resize-grip, resize-grip-visible, application, is-act
ive=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE,
focus-visible=FALSE, is-maximized=FALSE)
```

Текст надписи можно изменить с помощью **GAccessor.text**.

In [24]:
```
using Gtk


win = GtkWindow("Пример GtkLabel", 400, 200)

label = GtkLabel("Это метка с текстом!")
GAccessor.text(label,"Мой другой текст")

push!(win, label)
showall(win)
```

Out[24]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
GtkLabel", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WI
N_POS_NONE, default-width=400, default-height=200, destroy-with-parent=FA
LSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, ty
pe-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-
hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, dec
orated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-fo
r, attached-to, has-resize-grip, resize-grip-visible, application, is-act
ive=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE,
focus-visible=FALSE, is-maximized=FALSE)
```

Метод **GAccessor.markup** используется для того, чтобы задать разметку (markup) для текста в виджете GtkLabel. Это позволяет не просто отображать текст, но и

форматировать его, например, сделать части текста жирными, курсивными, добавить гиперссылки или изменить цвет.

Давайте рассмотрим примеры, где будет использована разметка для стилизации текста.

Пример 1. Изменение размера текста (size).

In [25]:
```
using Gtk

# Создаем окно
win = GtkWindow("Изменение размера шрифта", 400, 300)

label = GtkLabel("")
GAccessor.markup(label,"""
<span size="small">Маленький размер шрифта</span>\n
<span size="medium">Средний размер шрифта</span>\n
<span size="large">Большой размер шрифта</span>\n
<span size="x-large">Очень большой размер шрифта</span>\n
<span size="xx-large">Очень-очень большой размер шрифта</span>
""")

# Добавляем метку в окно
push!(win, label)

# Показываем окно
showall(win)
```

Out[25]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Измене
ние размера шрифта", role=NULL, resizable=TRUE, modal=FALSE, window-posit
ion=GTK_WIN_POS_NONE, default-width=400, default-height=300, destroy-with
-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL,
screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, s
kip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map
=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, tr
ansient-for, attached-to, has-resize-grip, resize-grip-visible, applicati
on, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visi
ble=FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

Пример 2. Установка цвета текста (foreground).

In [26]:
```
using Gtk

# Создаем окно
win = GtkWindow("Цвет текста", 400, 300)
```

```
label = GtkLabel("")
GAccessor.markup(label,"""
<span foreground="red">Красный цвет текста</span>\n
<span foreground="green">Зеленый цвет текста</span>\n
<span foreground="blue">Синий цвет текста</span>\n
<span foreground="purple">Пурпурный цвет текста</span>\n
<span foreground="#FF5733">Пользовательский цвет (#FF5733)</span>
""")

push!(win, label)
showall(win)
```

Out[26]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Цвет т
екста", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_P
OS_NONE, default-width=400, default-height=300, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

Пример 3. Установка фона текста (background).

In [27]:
```
using Gtk

# Создаем окно
win = GtkWindow("Цвет фона текста", 400, 300)

label = GtkLabel("")
GAccessor.markup(label,"""
<span background="yellow" foreground="black">Желтый фон и черный текст </s
<span background="lightgray" foreground="black">Светло-серый фон и черный
<span background="cyan" foreground="black">Голубой фон и черный текст </sp
<span background="black" foreground="white">Черный фон и белый текст</span
""")

# Добавляем метку в окно
push!(win, label)

# Показываем окно
showall(win)
```

Out[27]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Цвет ф
она текста", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_
WIN_POS_NONE, default-width=400, default-height=300, destroy-with-parent=
FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen,
type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-page
r-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, d
ecorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-
for, attached-to, has-resize-grip, resize-grip-visible, application, is-a
ctive=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALS
E, focus-visible=FALSE, is-maximized=FALSE)
```

Пример 4. Использование конкретного шрифта (font).

In [28]:
```
using Gtk

# Создаем окно
win = GtkWindow("Шрифт текста", 400, 300)

label = GtkLabel("")
GAccessor.markup(label,"""
<span font="Arial 14">Шрифт Arial, размер 14</span>\n
<span font="Courier New 18">Шрифт Courier New, размер 18</span>\n
<span font="Times New Roman 20">Шрифт Times New Roman, размер 20</span>\n
<span font="Helvetica 16">Шрифт Helvetica, размер 16</span>
""")

# Добавляем метку в окно
push!(win, label)

# Показываем окно
showall(win)
```

Out[28]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Шрифт
текста", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_
POS_NONE, default-width=400, default-height=300, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

Пример 5. Изменение жирности шрифта (weight).

In [29]:
```
using Gtk

# Создаем окно
win = GtkWindow("Жирность текста", 400, 300)

label = GtkLabel("")
GAccessor.markup(label,"""
<span weight="normal">Обычный шрифт</span>\n
<span weight="bold"> Жирный шрифт</span>\n
<span weight="light">Легкий шрифт</span>\n
""")

# Добавляем метку в окно
push!(win, label)

# Показываем окно
showall(win)
```

Out[29]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Жирнос
ть текста", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_W
IN_POS_NONE, default-width=400, default-height=300, destroy-with-parent=F
ALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, t
ype-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager
-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, de
corated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-f
or, attached-to, has-resize-grip, resize-grip-visible, application, is-ac
tive=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALS
E, focus-visible=FALSE, is-maximized=FALSE)
```

Пример 6. Изменение стиля шрифта (style).

In [30]:
```
using Gtk

# Создаем окно
win = GtkWindow("Стиль текста", 400, 300)

label = GtkLabel("")
GAccessor.markup(label,"""
<span style="normal">Нормальный стиль</span>\n
<span style="italic">Курсивный стиль</span>\n
<span style="oblique">Наклонный стиль</span>\n
""")

# Добавляем метку в окно
push!(win, label)

# Показываем окно
showall(win)
```

Out[30]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Стиль
текста", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_
POS_NONE, default-width=400, default-height=300, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

Пример 7. Подчеркивание текста (underline).

In [31]:
```
using Gtk

# Создаем окно
win = GtkWindow("Подчеркивание текста", 400, 300)

label = GtkLabel("")
GAccessor.markup(label,"""
<span underline="none">Без подчеркивания</span>\n
<span underline="single">Одиночное подчеркивание</span>\n
<span underline="double">Двойное подчеркивание</span>\n
<span underline="error">Подчеркивание ошибочного текста</span>
""")

# Добавляем метку в окно
push!(win, label)

# Показываем окно
showall(win)
```

Out[31]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Подчер
кивание текста", role=NULL, resizable=TRUE, modal=FALSE, window-position=
GTK_WIN_POS_NONE, default-width=400, default-height=300, destroy-with-par
ent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, scre
en, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-
pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRU
E, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transi
ent-for, attached-to, has-resize-grip, resize-grip-visible, application,
is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=
FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

Пример 8. Перечеркивание текста (strikethrough).

In [33]:
```
using Gtk

# Создаем окно
win = GtkWindow("Перечеркивание текста", 400, 300)

label = GtkLabel("")
GAccessor.markup(label,"""
<span strikethrough="false">Без перечеркивания</span>\n
<span strikethrough="true">Перечеркнутый текст</span>
""")

# Добавляем метку в окно
push!(win, label)

# Показываем окно
showall(win)
```

Out[33]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Перече
ркивание текста", role=NULL, resizable=TRUE, modal=FALSE, window-position
=GTK_WIN_POS_NONE, default-width=400, default-height=300, destroy-with-pa
rent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, scr
een, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip
-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TR
UE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, trans
ient-for, attached-to, has-resize-grip, resize-grip-visible, application,
is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=
FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

9.3.2 GtkEntry

GtkEntry – это виджет для ввода текста, который представляет собой однострочное текстовое поле.

Основные свойства GtkEntry:

- **:text** – текущее содержимое текстового поля (строка);
- **:visibility** – отображение текста (используется, например, для скрытия текста в поле ввода пароля);
- **:editable** – разрешает или запрещает редактирование текста;
- **:placeholder_text** – текст-подсказка, который отображается в поле, если оно пустое (например, для указания пользователю, что нужно ввести).

Пример 1. Создание простого поля для ввода текста.

In [34]:
```
using Gtk

# Создаем окно
win = GtkWindow("GtkEntry")

# Создаем поле для ввода текста
entry = GtkEntry()

# Устанавливаем начальный текст
set_gtk_property!(entry, :text, "Введите текст сюда")

# Добавляем поле в окно
push!(win, entry)
showall(win)
```

Out[34]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="GtkEnt
ry", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS_
NONE, default-width=-1, default-height=-1, destroy-with-parent=FALSE, hid
e-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint=
GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FAL
SE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=T
RUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, attac
hed-to, has-resize-grip, resize-grip-visible, application, is-active=FALS
E, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-v
isible=FALSE, is-maximized=FALSE)
```

Пример 2. Получение текста из поля ввода.

In [36]:
```
using Gtk

# Создаем окно
win = GtkWindow("GtkEntry",400,300)

# Создаем поле для ввода текста
entry = GtkEntry()

# Устанавливаем текст по умолчанию
set_gtk_property!(entry, :placeholder_text, "Введите ваш email")

# Функция для получения введенного текста
function get_input_text(widget)
    str = get_gtk_property(entry,:text,String)
    println("Введенный текст: ", str)
end

# Добавляем обработчик события для нажатия клавиши Enter
signal_connect(get_input_text, entry, "activate")

# Добавляем поле в окно
push!(win, entry)
showall(win)
```

Out[36]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="GtkEnt
ry", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS_
NONE, default-width=400, default-height=300, destroy-with-parent=FALSE, h
ide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hin
t=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=F
ALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated
=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, att
ached-to, has-resize-grip, resize-grip-visible, application, is-active=FA
LSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus
-visible=FALSE, is-maximized=FALSE)
Введенный текст: julia
```

Пример 3. Работа с паролем (скрытие текста).

In [37]:
```julia
using Gtk

# Создаем окно
win = GtkWindow("Password", 500, 400)

# Создаем поле для ввода пароля
entry = GtkEntry()

# Устанавливаем текст-подсказку
set_gtk_property!(entry, :placeholder_text, "Введите ваш пароль")

# Скрываем введенный текст (для поля пароля)
set_gtk_property!(entry, :visibility, false)

# Добавляем поле в окно
push!(win, entry)
showall(win)
```

Out[37]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Passwo
rd", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS_
NONE, default-width=500, default-height=400, destroy-with-parent=FALSE, h
ide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hin
t=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=F
ALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated
=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, att
ached-to, has-resize-grip, resize-grip-visible, application, is-active=FA
LSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus
-visible=FALSE, is-maximized=FALSE)
```

Пример 4. Отключение редактирования.

In [38]:
```
using Gtk

# Создаем окно
win = GtkWindow("Пример записи, не подлежащей редактированию", 500, 400)

# Создаем поле для ввода текста
entry = GtkEntry()

# Устанавливаем начальный текст
set_gtk_property!(entry, :text, "Этот текст нельзя редактировать")

# Отключаем редактирование
set_gtk_property!(entry, :editable, false)

# Добавляем поле в окно
push!(win, entry)
showall(win)
```

Out[38]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
записи, не подлежащей редактированию", role=NULL, resizable=TRUE, modal=F
ALSE, window-position=GTK_WIN_POS_NONE, default-width=500, default-height
=400, destroy-with-parent=FALSE, hide-titlebar-when-maximized=FALSE, ico
n, icon-name=NULL, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-ta
skbar-hint=FALSE, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus
=TRUE, focus-on-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRA
VITY_NORTH_WEST, transient-for, attached-to, has-resize-grip, resize-grip
-visible, application, is-active=FALSE, has-toplevel-focus=FALSE, startup
-id, mnemonics-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

9.4 Виджеты списка и дерева

GtkTreeView – это виджет для отображения табличных или иерархических данных. Несмотря на название, GtkTreeView используется не только для отображения деревьев, но и для работы с обычными списками. Главной особенностью этих виджетов является то, что GtkTreeView не хранит данные напрямую. Вместо этого данные хранятся в контейнерах, таких как **GtkListStore** для списков и **GtkTreeStore** для деревьев.

Списки (или таблицы) представляют собой упорядоченные коллекции данных, где каждый элемент можно воспринимать как строку. Например, можно создать таблицу, в которой будут храниться данные о людях: имя, возраст и пол. Для таких данных удобно использовать **GtkListStore**, который представляет собой таблицу, где каждая строка может содержать несколько значений разных типов. Если же необходимо отобразить данные в виде дерева, где элементы могут иметь дочерние элементы, то лучше использовать **GtkTreeStore**. Дерево позволяет организовать элементы в иерархическую структуру, где один элемент может быть родителем других, создавая, например, структуру каталогов в файловой системе. В обоих случаях для отображения данных используется виджет **GtkTreeView**, который привязывается к контейнеру данных и отображает их на экране. Отличие между списком и деревом заключается в том, что в списке все элементы независимы друг от друга, а в дереве каждый элемент может иметь потомков, создавая иерархию.

9.4.1 GtkListStore

Создание GtkListStore: store = GtkListStore(T1, T2, ..., Tn),

где **T1, T2, ..., Tn** – типы данных для каждого столбца. Например, String, Int, Bool, и т.д.

Добавление строки: push!(store, (value1, value2, ..., value_n))

- **store** – объект GtkListStore;
- **value1, value2, ..., value_n** – значения для добавления в строку.

Вставка данных: insert!(store, row_index, (value1, value2, ..., value_n))

- **store** – это объект GtkListStore;
- **row_index** – индекс строки, в которую нужно вставить данные;
- **(value1, value2, ..., value_n)** – кортеж значений, которые будут вставлены в соответствующие столбцы.

Получение данных с использованием индексации:

- ls[row_index, column_index] – возвращает строку по индексу;
- ls[row_index, column_index] = value – устанавливает значение в строку и столбец по индексу.

Получение количества строк: length(store)

GtkTreeView – это виджет, который отображает данные из модели в виде таблицы или дерева. Важный момент: для отображения данных в GtkTreeView нам нужно передать модель данных через интерфейс GtkTreeModel. GtkTreeView получает данные из модели и использует их для отображения на экране.

```
GtkTreeView(GtkTreeModel(store))
```

Рендереры – это компоненты, которые отвечают за визуализацию данных, предоставленных моделью, в виде видимых элементов на экране. Они могут отображать текст, изображения, прогресс-бары, чекбоксы и другие элементы интерфейса.

GtkCellRendererText – это рендерер, который используется для отображения текста в ячейках таблицы или дерева. Когда вы хотите отображать строковые данные (например, имя или возраст), вам нужно использовать этот рендерер.

GtkCellRendererToggle – это рендерер, который используется для отображения булевых значений в виде чекбоксов. Когда значение в модели данных представляет собой true или false, этот рендерер позволяет пользователю взаимодействовать с этим значением через чекбокс.

GtkCellRendererProgress – это рендерер, который используется для отображения прогресса выполнения в ячейках таблицы или дерева. Он принимает числовое значение и отображает его как прогресс (например, от 0% до 100%).

GtkTreeViewColumn – это объект, который определяет, как отображать данные в каждой колонке, ассоциируя их с рендерерами, которые отвечают за визуальное представление данных (например, текст, чекбоксы, прогресс-бары и т.д.).

Пример 1. Список людей с именем, возрастом и полом.

In [39]:
```
using Gtk

# Создаем контейнер для данных: 3 колонки (имя, возраст, пол)
ls = GtkListStore(String, Int, Bool)

# Заполняем список данными
push!(ls, ("Саша", 20, false))  # Мужчина
push!(ls, ("Рома", 30, false))   # Мужчина
push!(ls, ("Маша", 25, true))    # Женщина

# Создаем виджет TreeView для отображения данных из контейнера
tv = GtkTreeView(GtkTreeModel(ls))

# Создаем рендереры для отображения данных в колонках. Рендереры — это ком
rTxt = GtkCellRendererText()  # Рендерер для текста
rTog = GtkCellRendererToggle()  # Рендерер для чекбокса

# Создаем колонки для отображения данных
c1 = GtkTreeViewColumn("Имя", rTxt, Dict([("text", 0)]))  # Колонка для им
c2 = GtkTreeViewColumn("Возраст", rTxt, Dict([("text", 1)]))   # Колонка д
c3 = GtkTreeViewColumn("Пол", rTog, Dict([("active", 2)]))  # Колонка для
# Добавляем колонки в TreeView

push!(tv, c1, c2, c3)


# Создаем окно для отображения данных
win = GtkWindow(tv, "List View")

# Показываем окно
showall(win)
```

Out[39]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="List V
iew", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS
_NONE, default-width=-1, default-height=-1, destroy-with-parent=FALSE, hi
de-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint
=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FA
LSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=
TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, atta
ched-to, has-resize-grip, resize-grip-visible, application, is-active=FAL
SE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-
visible=FALSE, is-maximized=FALSE)
```

Пример 2. Список людей с именем, возрастом и полом, с возможностью изменять ширину колонок.

In [45]:
```
using Gtk

# Создаем контейнер для данных: 3 колонки (имя, возраст, пол)
ls = GtkListStore(String, Int, Bool)

# Заполняем список данными
push!(ls, ("Саша", 20, false))  # Мужчина
push!(ls, ("Рома", 30, false))   # Мужчина
push!(ls, ("Маша", 25, true))    # Женщина

# Создаем виджет TreeView для отображения данных из контейнера
tv = GtkTreeView(GtkTreeModel(ls))

# Создаем рендереры для отображения данных в колонках. Рендереры — это ком
rTxt = GtkCellRendererText()  # Рендерер для текста
rTog = GtkCellRendererToggle()  # Рендерер для чекбокса

# Создаем колонки для отображения данных
c1 = GtkTreeViewColumn("Имя", rTxt, Dict([("text", 0)]))  # Колонка для им
c2 = GtkTreeViewColumn("Возраст", rTxt, Dict([("text", 1)]))   # Колонка д
c3 = GtkTreeViewColumn("Пол", rTog, Dict([("active", 2)]))  # Колонка для
# Добавляем колонки в TreeView

push!(tv, c1, c2, c3)

for c in [c1, c2, c3]
    GAccessor.resizable(c, true)
#функция задает, может ли колонка быть изменена по ширине пользователем.
#Параметр true говорит о том, что колонка будет изменяемой по ширине.
#Если бы был передан параметр false, колонка не могла бы изменять свою шир

end
```

```
# Создаем окно для отображения данных
win = GtkWindow(tv, "List View")

# Показываем окно
showall(win)
```

Out[45]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="List V
iew", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS
_NONE, default-width=-1, default-height=-1, destroy-with-parent=FALSE, hi
de-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint
=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FA
LSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=
TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, atta
ched-to, has-resize-grip, resize-grip-visible, application, is-active=FAL
SE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-
visible=FALSE, is-maximized=FALSE)
```

Пример 3. Отображение прогресса задач.

In [42]:
```
using Gtk

# Создаём модель данных для имени и прогресса
ls = GtkListStore(String, Int)

# Добавляем данные в модель (имя и прогресс)
push!(ls, ("Задача 1", 50))  # Прогресс 50%
push!(ls, ("Задача 2", 80))  # Прогресс 80%
push!(ls, ("Задача 3", 30))  # Прогресс 30%

# Создаём виджет GtkTreeView с моделью данных
tv = GtkTreeView(GtkTreeModel(ls))

# Создаём рендерер для отображения прогресса
rProg = GtkCellRendererProgress()

# Создаём колонку для отображения прогресса
c1 = GtkTreeViewColumn("Задача", GtkCellRendererText(), Dict([("text", 0)]
c2 = GtkTreeViewColumn("Прогресс", rProg, Dict([("value", 1)]))  # "value"

# Добавляем колонки в GtkTreeView
push!(tv, c1, c2)

# Создаём окно и отображаем виджет
win = GtkWindow(tv, "Таблица с прогрессом")
showall(win)
```

Out[42]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Таблиц
а с прогрессом", role=NULL, resizable=TRUE, modal=FALSE, window-position=
GTK_WIN_POS_NONE, default-width=-1, default-height=-1, destroy-with-paren
t=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, scree
n, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-p
ager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRU
E, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transi
ent-for, attached-to, has-resize-grip, resize-grip-visible, application,
is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=
FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

Когда мы работаем с таблицами или списками данных в GTK, часто возникает необходимость взаимодействовать с выбранным элементом. Для этого используется объект **GtkTreeSelection**, который позволяет отслеживать и управлять выбором элементов в списке. Чтобы получить этот объект, необходимо использовать функцию:

```
selection = GAccessor.selection(tv)
```

Здесь **tv** – это объект **GtkTreeView**, в котором отображаются данные. С помощью **GtkTreeSelection** мы можем задать режим выбора: один элемент или несколько. В данном примере мы будем использовать выбор только одного элемента (одиночный выбор). Для включения мульти-выбора можно вызвать:

```
selection = GAccessor.mode(selection,
Gtk.GConstants.GtkSelectionMode.MULTIPLE)
```

Для текущего примера мы будем использовать одиночный выбор, и нам нужно получить индекс выбранного элемента. Это можно сделать следующим образом:

```
selected_item = selected(selection)
println("Выбранный элемент: ", ls[selected_item, 1])
```

Здесь **selected(selection)** возвращает индекс выбранного элемента, и мы можем получить данные из соответствующей строки с помощью индексации.

В случае, если пользователь выбрал элемент и вы хотите выполнить какое-то действие, например, вывести информацию о выбранной строке, можно использовать сигнал **"changed"**, который срабатывает каждый раз при изменении выбора:

```
signal_connect(selection, "changed") do widget
  if hasselection(selection)
    currentIt = selected(selection)
    println("Имя: ", ls[currentIt, 1], " Возраст: ", ls[currentIt, 2])
```

```
  end
end
```

Этот код подключает обработчик события, который будет вызываться каждый раз, когда пользователь выбирает новый элемент. В нем проверяется, был ли выбран элемент (с помощью **hasselection(selection)**), и если да – выводится информация о выбранной строке.

In [46]:

```
using Gtk

# Создаем контейнер для данных: 3 колонки (имя, возраст, пол)
ls = GtkListStore(String, Int, Bool)

# Заполняем список данными
push!(ls, ("Саша", 20, false))  # Мужчина
push!(ls, ("Рома", 30, false))   # Мужчина
push!(ls, ("Маша", 25, true))    # Женщина

# Создаем виджет TreeView для отображения данных из контейнера
tv = GtkTreeView(GtkTreeModel(ls))

# Создаем рендереры для отображения данных в колонках.
rTxt = GtkCellRendererText()  # Рендерер для текста
rTog = GtkCellRendererToggle()  # Рендерер для чекбокса

# Создаем колонки для отображения данных
c1 = GtkTreeViewColumn("Имя", rTxt, Dict([("text", 0)]))  # Колонка для им
c2 = GtkTreeViewColumn("Возраст", rTxt, Dict([("text", 1)]))   # Колонка д
c3 = GtkTreeViewColumn("Пол", rTog, Dict([("active", 2)]))  # Колонка для
# Добавляем колонки в TreeView

push!(tv, c1, c2, c3)

# Создаем окно для отображения данных
win = GtkWindow(tv, "List View")


selection = GAccessor.selection(tv)
signal_connect(selection, "changed") do widget
  if hasselection(selection)
    currentIt = selected(selection)
    println("Выбранный элемент: ", ls[currentIt, 1], " Возраст: ", ls[curr
  end
end

# Показываем окно
showall(win)
```

Out[46]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="List V
iew", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS
_NONE, default-width=-1, default-height=-1, destroy-with-parent=FALSE, hi
de-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint
=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FA
LSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=
TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, atta
ched-to, has-resize-grip, resize-grip-visible, application, is-active=FAL
SE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-
visible=FALSE, is-maximized=FALSE)
Выбранный элемент: Рома Возраст: 30
Выбранный элемент: Саша Возраст: 20
```

9.4.2 GtkTreeStore

Пример 1. Создание дерева с использованием GtkTreeStore.

In [48]:
```
using Gtk

# Создание модели дерева, в которой будут храниться строки
ts = GtkTreeStore(String)

# Добавляем элементы в дерево
iter1 = push!(ts, ("1",))
iter2 = push!(ts, ("2",), iter1)
iter3 = push!(ts, ("3",), iter2)

# Создаем виджет для отображения дерева
tv = GtkTreeView(GtkTreeModel(ts))

# Создаем рендерер текста для отображения текста в столбце
r1 = GtkCellRendererText()

# Создаем колонку для отображения данных
c1 = GtkTreeViewColumn("A", r1, Dict([("text", 0)]))

# Добавляем колонку в TreeView
push!(tv, c1)

# Создаем окно с TreeView
win = GtkWindow(tv, "Tree View")

# Показываем окно
showall(win)

# Изменение текста в первом элементе дерева
```

```
iter = Gtk.iter_from_index(ts, [1])
ts[iter, 1] = "один"
```

Out[48]: "один"

9.5 События клавиш

Чтобы обрабатывать события нажатия клавиш, необходимо использовать событие **key-press-event** для активного окна. Это событие срабатывает каждый раз, когда пользователь нажимает клавишу на клавиатуре в пределах этого окна. Ниже приведен пример кода, который демонстрирует, как это сделать.

In [50]:
```
using Gtk
win = GtkWindow("Пример нажатия клавиш")

# Подключаем обработчик события нажатия клавиши
signal_connect(win, "key-press-event") do widget, event
  # Извлекаем значение нажатой клавиши
  k = event.keyval

  # Выводим на экран код клавиши и сам символ
  println("Вы нажали клавишу с кодом ", k, ", что соответствует символу '"
end

showall(win)
```

Out[50]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Пример
нажатия клавиш", role=NULL, resizable=TRUE, modal=FALSE, window-position=
GTK_WIN_POS_NONE, default-width=-1, default-height=-1, destroy-with-paren
t=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, scree
n, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-p
ager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRU
E, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transi
ent-for, attached-to, has-resize-grip, resize-grip-visible, application,
is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=
FALSE, focus-visible=FALSE, is-maximized=FALSE)
Вы нажали клавишу с кодом 101, что соответствует символу 'e'.
Вы нажали клавишу с кодом 99, что соответствует символу 'c'.
Вы нажали клавишу с кодом 103, что соответствует символу 'g'.
```

9.6. Рисование в GTK

С помощью библиотеки Graphics.jl в сочетании с GTK.jl можно рисовать на **канвасе** (Canvas). Канвас в контексте графических интерфейсов – это специальный элемент (виджет), на котором можно рисовать различные графические объекты, такие как линии, прямоугольники, окружности, текст, изображения и так далее.

@GtkCanvas() – это макрос, создающий канвас для рисования.

```
canvas = @GtkCanvas()
```
@guarded draw() – этот макрос используется для создания обработчика рисования, который будет вызываться каждый раз, когда нужно перерисовать канвас (например, при изменении размеров окна или обновлении содержимого).

```
@guarded draw(widget) do widget
    # Код рисования
end
```
Где widget – это объект, на котором происходит рисование (в данном случае канвас).

getgc() – функция, которая возвращает контекст рисования для канваса. Контекст рисования – это объект, который управляет такими параметрами, как цвет, шрифт и стиль линии.

```
ctx = getgc(canvas)
```
Где canvas – объект канваса, с которого нужно получить контекст, а getgc(canvas) возвращает объект контекста рисования ctx, который используется для выполнения операций рисования, таких как рисование линий, фигур и настройка цвета.

height() и width() – эти функции возвращают размеры канваса, которые полезны для адаптивного рисования в зависимости от размеров окна.

```
h = height(canvas)
w = width(canvas)
```
Где height(canvas) – возвращает высоту канваса canvas, а width(canvas) – возвращает ширину канваса canvas.

rectangle() – рисует прямоугольник в контексте рисования. Этот вызов не заполняет прямоугольник, а только рисует его контур.

```
rectangle(ctx, x, y, w, h)
```
Где x, y – координаты верхнего левого угла прямоугольника, а w, h – ширина и высота прямоугольника.

set_source_rgb() – задает цвет рисования с использованием модели RGB (красный, зеленый, синий).

```
set_source_rgb(ctx, r, g, b)
```

Где r, g, b – компоненты красного, зеленого и синего цвета в диапазоне от 0 до 1.

fill() – заполняет текущую форму (например, прямоугольник) текущим цветом.

`fill(ctx)`
Эта функция применяет текущий цвет и заполняет все фигуры, нарисованные до этого (например, прямоугольники, круги).

In [52]:
```
using Gtk, Graphics

# Создаем канвас для рисования
c = @GtkCanvas()

# Создаем окно с канвасом
win = GtkWindow(c, "Canvas")

# Гарантируем, что рисунок будет перерисован
@guarded draw(c) do widget
    # Получаем контекст рисования (gc) для канваса
    ctx = getgc(c)

    # Получаем размеры канваса
    h = height(c)
    w = width(c)

    # Рисуем первый красный прямоугольник (верхнюю половину окна)
    rectangle(ctx, 0, 0, w, h / 2)
    set_source_rgb(ctx, 1, 0, 0)  # Красный цвет
    fill(ctx)  # Заполняем прямоугольник красным цветом

    # Рисуем второй синий прямоугольник (нижнюю четверть окна)
    rectangle(ctx, 0, 3 * h / 4, w, h / 4)
    set_source_rgb(ctx, 0, 0, 1)  # Синий цвет
    fill(ctx)  # Заполняем прямоугольник синим цветом
end

# Отображаем окно с канвасом
show(c)
```

Out[52]:
```
GtkCanvas(name="", parent, width-request=-1, height-request=-1, visible=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-default=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK | GDK_SCROLL_MASK, no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=NULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, valign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-end=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALSE, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1)
```

Рисование с обработкой событий мыши

Теперь давайте добавим возможность рисовать на канвасе с помощью мыши. Каждый раз, когда пользователь нажимает кнопку мыши (например, левую кнопку), на канвасе будет рисоваться зеленый круг в том месте, где произошел клик.

In [3]:
```
using Gtk, Graphics

# Создаем канвас
c = @GtkCanvas()

# Создаем окно с канвасом
win = GtkWindow(c, "Canvas")

# Подключаем обработчик события нажатия левой кнопки мыши.(button2press –
c.mouse.button1press = @guarded (widget, event) -> begin
    # Получаем контекст рисования
    ctx = getgc(widget)

    # Устанавливаем зеленый цвет для рисования
    set_source_rgb(ctx, 0, 1, 0)

    # Рисуем круг в точке, где был клик мышью
    arc(ctx, event.x, event.y, 5, 0, 2pi) # arc(ctx, xc, yc, radius, angle
    #ctx: графический контекст полученный с помощью getgc(widget). Это объ
    #xc, yc: координаты центра дуги или круга.
    #radius: радиус дуги или круга.
    #angle1, angle2: углы, определяющие начало и конец дуги.
    #Углы измеряются в радианах, и если angle1 и angle2 охватывают полный
    #Если же углы не охватывают весь круг, будет нарисована дуга.
    stroke(ctx)  # Обводим круг

    # Обновляем канвас, чтобы отобразить изменения
    reveal(widget)
end

# Показываем окно
show(c)
```

Out[3]:
```
GtkCanvas(name="", parent, width-request=-1, height-request=-1, visible=T
RUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=FALS
E, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-default=FA
LSE, receives-default=FALSE, composite-child=FALSE, style, events=GDK_POI
NTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTTON_PRESS_MASK |
GDK_BUTTON_RELEASE_MASK | GDK_SCROLL_MASK, no-show-all=FALSE, has-tooltip
=FALSE, tooltip-markup=NULL, tooltip-text=NULL, window, opacity=1,000000,
double-buffered, halign=GTK_ALIGN_FILL, valign=GTK_ALIGN_FILL, margin-lef
t, margin-right, margin-start=0, margin-end=0, margin-top=0, margin-botto
m=0, margin=0, hexpand=FALSE, vexpand=FALSE, hexpand-set=FALSE, vexpand-s
et=FALSE, expand=FALSE, scale-factor=1)
```

Пример с добавлением прямоугольника и линии.

In [4]:
```
using Gtk
```

```
# Создаем канвас
c = @GtkCanvas()

# Создаем окно с канвасом
win = GtkWindow(c, "Canvas")

# Подключаем обработчик события нажатия левой кнопки мыши
c.mouse.button1press = @guarded (widget, event) -> begin
    # Получаем контекст рисования
    ctx = getgc(widget)

    # Рисуем круг с радиусом 20
    set_source_rgb(ctx, 0, 1, 0)  # Зеленый цвет
    arc(ctx, event.x, event.y, 20, 0, 2pi)
    stroke(ctx)  # Обводим круг

    # Рисуем прямоугольник
    set_source_rgb(ctx, 1, 0, 0)  # Красный цвет
    rectangle(ctx, event.x + 40, event.y + 40, 60, 30) # Команда rectangle
    #ctx: графический контекст, полученный через getgc(widget). Это объект
    #x, y: координаты верхнего левого угла прямоугольника. Эти значения оп
    #width, height: размеры прямоугольника: ширина и высота соответственно
    stroke(ctx)  # Обводим прямоугольник

    # Рисуем линию
    set_source_rgb(ctx, 0, 0, 1)  # Синий цвет
    move_to(ctx, event.x, event.y)# move_to используется для перемещения к
    line_to(ctx, event.x + 100, event.y + 100) #line_to(ctx, x, y) использ
    #ctx: графический контекст.
    #x, y: координаты конечной точки линии.

    # Обновляем канвас, чтобы отобразить изменения
    reveal(widget)
end

# Показываем окно
show(c)
```

Out[4]:
```
GtkCanvas(name="", parent, width-request=-1, height-request=-1, visible=T
RUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=FALS
E, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-default=FA
LSE, receives-default=FALSE, composite-child=FALSE, style, events=GDK_POI
NTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTTON_PRESS_MASK |
GDK_BUTTON_RELEASE_MASK | GDK_SCROLL_MASK, no-show-all=FALSE, has-tooltip
=FALSE, tooltip-markup=NULL, tooltip-text=NULL, window, opacity=1,000000,
double-buffered, halign=GTK_ALIGN_FILL, valign=GTK_ALIGN_FILL, margin-lef
t, margin-right, margin-start=0, margin-end=0, margin-top=0, margin-botto
m=0, margin=0, hexpand=FALSE, vexpand=FALSE, hexpand-set=FALSE, vexpand-s
et=FALSE, expand=FALSE, scale-factor=1)
```

9.7 Пример программы

In [5]:
```
using Gtk

# Функция для обновления текста в поле ввода
```

```
function update_display(entry, text)
    GAccessor.text(entry, text)
end

# Функция для обработки нажатий кнопок
function on_button_click(entry, label)
    str = get_gtk_property(entry,:text,String)
    new_text = str * label
    update_display(entry, new_text)
end

# Функция для вычисления результата
function calculate_result(entry)
    str = get_gtk_property(entry,:text,String)
    result = eval(Meta.parse(str))
    str = "$result"
    update_display(entry, str)
end

# Функция для очистки поля ввода
function clear_entry(entry)
    update_display(entry, "")
end

# Основная функция для создания окна калькулятора
function create_calculator_window()
    win = GtkWindow("Калькулятор", 150, 150)

    # Создаем поле для ввода текста (где отображаются числа и операторы)
    entry = GtkEntry()
    set_gtk_property!(entry, :editable, false)
    GAccessor.text(entry, "")
    box = GtkBox(:v)
    push!(box, entry)

    # Создаем сетку для размещения кнопок
    grid = GtkGrid()

    # Создаем кнопки и привязываем обработчики
    button7 = GtkButton("7")
    signal_connect(button7, :clicked) do widget
        on_button_click(entry, "7")
    end

    button8 = GtkButton("8")
    signal_connect(button8, :clicked) do widget
        on_button_click(entry, "8")
    end

    button9 = GtkButton("9")
    signal_connect(button9, :clicked) do widget
        on_button_click(entry, "9")
    end

    button_div = GtkButton("/")
    signal_connect(button_div, :clicked) do widget
```

```
    on_button_click(entry, "/")
end

button4 = GtkButton("4")
signal_connect(button4, :clicked) do widget
    on_button_click(entry, "4")
end

button5 = GtkButton("5")
signal_connect(button5, :clicked) do widget
    on_button_click(entry, "5")
end

button6 = GtkButton("6")
signal_connect(button6, :clicked) do widget
    on_button_click(entry, "6")
end

button_mul = GtkButton("*")
signal_connect(button_mul, :clicked) do widget
    on_button_click(entry, "*")
end

button1 = GtkButton("1")
signal_connect(button1, :clicked) do widget
    on_button_click(entry, "1")
end

button2 = GtkButton("2")
signal_connect(button2, :clicked) do widget
    on_button_click(entry, "2")
end

button3 = GtkButton("3")
signal_connect(button3, :clicked) do widget
    on_button_click(entry, "3")
end

button_sub = GtkButton("-")
signal_connect(button_sub, :clicked) do widget
    on_button_click(entry, "-")
end

button0 = GtkButton("0")
signal_connect(button0, :clicked) do widget
    on_button_click(entry, "0")
end

button_dot = GtkButton(".")
signal_connect(button_dot, :clicked) do widget
    on_button_click(entry, ".")
end

button_eq = GtkButton("=")
signal_connect(button_eq, :clicked) do widget
    calculate_result(entry)
```

```
    end

    button_add = GtkButton("+")
    signal_connect(button_add, :clicked) do widget
        on_button_click(entry, "+")
    end

    # Создание кнопки для очистки поля ввода
    button_clear = GtkButton("C")
    signal_connect(button_clear, :clicked) do widget
        clear_entry(entry)
    end

    # Размещение кнопок в сетке
    grid[1, 1] = button7
    grid[2, 1] = button8
    grid[3, 1] = button9
    grid[4, 1] = button_div

    grid[1, 2] = button4
    grid[2, 2] = button5
    grid[3, 2] = button6
    grid[4, 2] = button_mul

    grid[1, 3] = button1
    grid[2, 3] = button2
    grid[3, 3] = button3
    grid[4, 3] = button_sub

    grid[1, 4] = button0
    grid[2, 4] = button_dot
    grid[3, 4] = button_eq
    grid[4, 4] = button_add

    # Размещение кнопки очистки
    grid[1:4, 5] = button_clear


    # Автоматическое подстраивание размеров ячеек
    set_gtk_property!(grid, :column_homogeneous, true)  # Автоматическое п
    set_gtk_property!(grid, :row_homogeneous, true)     # Автоматическое п

    # Добавление кнопок в окно
    push!(box, grid)
    set_gtk_property!(win, :child, box)

    # Показать окно
    showall(win)
end

# Запуск калькулятора
create_calculator_window()
```

Out[5]:
```
GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visib
le=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=
FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-defaul
t=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0,
no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=N
ULL, window, opacity=1,000000, double-buffered, halign=GTK_ALIGN_FILL, va
lign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-en
d=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALS
E, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, bo
rder-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Кальку
лятор", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_P
OS_NONE, default-width=150, default-height=150, destroy-with-parent=FALS
E, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type
-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hi
nt=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decor
ated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for,
attached-to, has-resize-grip, resize-grip-visible, application, is-active
=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, fo
cus-visible=FALSE, is-maximized=FALSE)
```

Глава 10 Форматированный вывод

Функции **@printf** и **@sprintf** из пакета Printf.jl в языке программирования Julia используются аналогично функциям **printf** и **sprintf** из языка Си. Она принимает строку формата, а затем значения, которые нужно вставить в этот формат, и возвращает отформатированную строку.

Для получения дополнительной информации о пакете см. документацию.

In [7]:
```
using Printf
x = 3.14159
@printf("%.2f", x)
```

```
3.14
```

In [8]:
```
using Printf
h=-123.678549
k=1234567889
@printf("h=%6.4f\n" ,h)
@printf("k=%15d\n" ,k)
@printf("h=%16.9f\tk=%d\n" , h, k)
```

```
h=-123.6785
k=     1234567889
h=  -123.678549000      k=1234567889
```

In [9]:
```
n = 4
m = 7
A = rand(n, m)

println("Матрица A\n")
for i in 1:n
    for j in 1:m
```

```
        @printf("%6.2f", A[i, j])
    end
    println()
end
```

```
Матрица A

 0.39  0.15  1.00  0.67  0.82  0.55  0.39
 0.33  0.97  0.21  0.07  0.94  1.00  0.73
 0.04  0.19  0.02  0.95  0.83  0.59  0.76
 0.00  0.65  0.51  0.25  0.03  0.53  0.82
```

Глава 11 Файлы в Julia

В Julia можно легко работать с текстовыми файлами, выполняя операции как записи, так и чтения данных. Давайте рассмотрим примеры создания, записи и считывания данных из текстового файла в Julia

Работа с файлом начинается с того, что указывается его имя **fname="test.dat"**. Далее файл нужно открыть **f1=open(fname, mode)**, mode характеризует возможности работы с файлом

mode	Описание
r	Чтение
w	Запись
a	Добавление
r+	Чтение и Запись
w+	Запись и Чтение
a+	Добавление и Чтение

Можно сразу выполнить команду **f1=open("test.dat","r+")**.

Наиболее простой способ работы с данными из файла с именем fname выглядит так

```
open(fname, mode) do f1
... действия с файлом f1
end
```

В этом случае файл закрывать не нужно, он закрывается автоматически послевыхода из блока.

11.1 Создание и запись данных в файл

Пример текстового файла.

In [13]:
```
# Открываем файл для записи (если файл уже существует, он будет перезаписа
open("file.txt", "w") do file
    write(file, "Привет, мир!\n")
    write(file, "Это текстовый файл, созданный в Julia.")
end
```

Out[13]: 64

In [12]:
```
data = ["Первая строка данных", "Вторая строка данных", "Третья строка дан

open("file.txt", "a") do file  # Открываем файл для добавления данных
    for line in data # Переменная line представляет каждую строку данных и
        write(file, line * "\n")
    end
end
```

Пример двоичного файла.

In [14]:
```
data = rand(10)  # генерируем случайные данные
file = open("binary_data.bin", "w")  # открываем файл для записи в двоично
write(file, data)  # записываем данные в файл
close(file)  # закрываем файл
```

11.2 Чтение данных из файла

Пример текстового файла.

Для чтения данных из файла используем функцию **eachline**, которая читает файл построчно.

In [15]:
```
open("file.txt","r") do file
    for line in eachline(file)
        println(line)
    end
end
```

```
Привет, мир!
Это текстовый файл, созданный в Julia.
```

Пример двоичного файла

In [16]:
```
open("binary_data.bin", "r") do file
    while !eof(file)
        data = read(file, Float64)
        println(data)
    end
end
```

```
0.8519436754691689
0.309260771531353
0.5845147188674208
0.14680417622030473
0.13933755683559834
0.5469168051720864
0.7797652243356914
0.5859923566152532
0.7266987456717335
0.9112098004021612
```

Задача Записать в файл с расширением .bin и считать из него матрицу

$$A_{i,j} = \begin{cases} & 2.7 \cdot n, & i = j \\ & 1, & i \neq j \end{cases}$$

.

In [17]:
```
print("N = ")
N = parse(Int64, readline())
A = ones(Float64, N, N)
for i in 1:N
    for j in 1:N
        if i == j
            A[i, j] = 2.7 * N
        else
            A[i, j] = 1
        end
    end
end
file = open("matr.bin", "w")
write(file, N)
write(file, A)
close(file)
```

```
N =
```

In [18]:
```
open("matr.bin", "r") do file
    N = read(file, Int)
    while !eof(file)
        for j in 1:N
        for i in 1:N
        data = read(file, Float64)
        print(data," ")
        end
        print("\n")
        end
    end
end
```

```
13.5 1.0 1.0 1.0 1.0
1.0 13.5 1.0 1.0 1.0
1.0 1.0 13.5 1.0 1.0
1.0 1.0 1.0 13.5 1.0
1.0 1.0 1.0 1.0 13.5
```