

C GUI Calculator: A Simple Scientific Calculator Using C and GTK+ 3

Abstract

We built a calculator program using C programming language and GTK+ 3 for the graphical interface. Our calculator can do basic math (add, subtract, multiply, divide) and scientific functions (sin, cos, square root, etc.). The program uses smart algorithms to understand math expressions and shows clear error messages when something goes wrong.

Keywords: C Programming, Calculator, GTK+ 3, Math Functions

I. Introduction

What We Made

We created a calculator application that works like a real calculator but runs on a computer. Instead of just doing simple math, our calculator can also do advanced math functions that students and engineers need.

Why We Made It

This project helped us learn:

- How to program in C language
- How to create windows and buttons (GUI programming)
- How to handle complex math calculations
- How to work as a team on a programming project

Main Features

Our calculator can:

- Do basic math: $2+3$, $5*4$, $10/2$, 2^3
- Handle complex expressions: $(5+3)*2$, $\sin(30)+\cos(60)$
- Calculate scientific functions: square root, sine, cosine, logarithms
- Show helpful error messages when you make mistakes

- Work on different computers (Windows, Mac, Linux)

II. How We Built It

A. Program Structure

We organized our code into four main parts:

1. **Math Brain:** Understands and calculates math expressions
2. **Calculator Display:** Shows numbers and results on screen
3. **Button Handler:** Responds when you click buttons
4. **Memory Manager:** Keeps track of data without wasting memory

B. Important Data Types

We created special data structures to store information:

- **Calculator State:** Remembers what's on the display and what you typed
- **Math Tokens:** Breaks down "2+3" into pieces: "2", "+", "3"
- **Number Stack:** Stores numbers while calculating complex expressions
- **Input Parser:** Reads what you type and understands it

C. Smart Algorithm

We used a famous algorithm called "Shunting Yard" that:

- Reads math expressions like "2+3*4"
- Knows that multiplication comes before addition
- Converts it to a form the computer can easily calculate
- Gives the right answer: 14 (not 20)

III. Programming Details

A. Understanding Math Expressions

Step 1 - Breaking It Down: The program reads "sin(30)+2" and identifies:

- "sin" = function
- "(" = start of function input
- "30" = number
- ")" = end of function input
- "+" = addition operator
- "2" = another number

Step 2 - Following Math Rules: The program knows:

- Multiplication and division come before addition and subtraction
- Functions are calculated first
- Parentheses can change the order: $(2+3)*4 = 20$

Step 3 - Calculating: Uses a stack (like a pile of plates) to calculate step by step.

B. Safe Math Functions

We made sure our calculator doesn't crash by checking for problems:

Division by Zero:

- Problem: $5 \div 0$ is impossible
- Our Solution: Show "Division by zero" error message

Square Root of Negative Numbers:

- Problem: $\sqrt{-4}$ doesn't work with real numbers
- Our Solution: Show "Cannot take square root of negative number"

Trigonometry Problems:

- Problem: $\tan(90^\circ)$ is undefined
- Our Solution: Show "Tangent undefined at 90° "

C. User Interface

We created the calculator window using GTK+ 3:


Display Screen: Shows your input and results, just like a real calculator

Number Buttons: Big, easy-to-click buttons for 0-9

Operation Buttons: +, -, *, /, ^ for basic math

Function Buttons: sin, cos, tan, sqrt, log for advanced math

Special Buttons:

- C (Clear) - starts over
- = (Equals) - calculates the answer
-  (Backspace) - deletes last character

IV. What Our Calculator Can Do

A. Basic Math Examples

Simple calculations:

```
5 + 3 = 8
10 - 4 = 6
7 * 8 = 56
15 / 3 = 5
2^4 = 16
```

Complex expressions:

```
(5 + 3) * 2 = 16
10 + 2 * 3 = 16 (multiplication first)
2^3^2 = 512      (right to left for powers)
```

B. Scientific Functions

```
Square root: sqrt(16) = 4
Sine: sin(30) = 0.5 (in degrees)
Cosine: cos(60) = 0.5
Tangent: tan(45) = 1
Logarithm: log(100) = 2
Natural log: ln(2.718) = 1
```

C. Error Handling

When you make mistakes, our calculator helps you:

```
sqrt(-4) → "Cannot take square root of negative number"
5/0 → "Division by zero"
tan(90) → "Tangent undefined at 90°"
((2+3) → "Mismatched parentheses"
```

V. Testing and Results

A. What We Tested

Basic Functions: We tried hundreds of calculations to make sure addition, subtraction, multiplication, and division work correctly.

Scientific Functions: We tested all trigonometry and logarithm functions with different inputs.

Error Cases: We deliberately tried to break the calculator to make sure it handles errors gracefully.

User Interface: We clicked every button and tried keyboard shortcuts to ensure everything responds properly.

B. Performance Results

- **Speed:** Calculations happen instantly
- **Memory:** Uses very little computer memory

- **Reliability:** No crashes during extensive testing
- **Compatibility:** Works on Windows, Mac, and Linux

C. User Experience

- **Easy to Use:** Familiar calculator layout
- **Clear Display:** Large, readable numbers and text
- **Helpful Errors:** Clear messages when something goes wrong
- **Keyboard Support:** Can use Enter key for calculations

VI. Challenges We Faced

A. Math Expression Problems

Challenge: Making the calculator understand " $2+3*4$ " should equal 14, not 20.

Solution: We learned and implemented the Shunting Yard algorithm, which is like teaching the computer the order of operations from math class.

B. Memory Management

Challenge: Preventing the program from using too much memory or crashing.

Solution: We carefully managed memory by cleaning up after calculations and using dynamic arrays that grow and shrink as needed.

C. User Interface Design

Challenge: Making buttons and display look good and work properly.

Solution: We studied GTK+ documentation and created a clean, simple layout similar to real calculators.

D. Working as a Team

Challenge: Five people working on the same code without conflicts.

Solution: We used Git version control to track changes and coordinate our work effectively.

VII. What We Learned

A. Programming Skills

- **C Language:** We became much better at writing C code, managing memory, and debugging programs

- **Algorithms:** We learned how to implement complex algorithms like Shunting Yard
- **Problem Solving:** We developed skills to break big problems into smaller, manageable pieces

B. Software Development

- **Planning:** We learned to design our program before writing code
- **Testing:** We discovered the importance of thorough testing
- **Documentation:** We practiced writing clear explanations of our code
- **Teamwork:** We improved our collaboration and communication skills

C. Math and Computer Science

- **Mathematical Programming:** We learned how computers handle mathematical calculations
- **Data Structures:** We used stacks, arrays, and custom structures effectively
- **User Interface Design:** We gained experience in creating user-friendly applications

VIII. Future Improvements

If we had more time, we could add:

A. More Math Functions

- Factorial calculations ($5! = 120$)
- More trigonometric functions (arcsin, arccos)
- Statistical functions (average, standard deviation)
- Number base conversions (binary, hexadecimal)

B. Better User Experience

- History of previous calculations
- Different color themes
- Larger display for longer expressions
- Sound effects for button presses

C. Advanced Features

- Ability to save and recall numbers
- Simple graphing of functions
- Unit conversions (feet to meters, etc.)
- Programming mode for computer science calculations

IX. Conclusion

What We Accomplished

We successfully created a working scientific calculator that:

- Performs accurate mathematical calculations
- Has an intuitive, easy-to-use interface
- Handles errors gracefully
- Works reliably across different operating systems

Technical Achievements

- Implemented a complex parsing algorithm (Shunting Yard)
- Created robust error handling for mathematical edge cases
- Built a responsive graphical user interface
- Managed memory efficiently to prevent crashes

Educational Value

This project gave us hands-on experience with:

- Real-world C programming
- Algorithm implementation
- User interface design
- Team collaboration
- Software testing and debugging

Personal Growth

Each team member improved their:

- Programming skills
- Problem-solving abilities
- Communication and teamwork
- Understanding of computer science concepts

The calculator project bridges the gap between theoretical computer science concepts and practical application development. We're proud of creating a tool that's both functional and educational, demonstrating our growth as programmers and our ability to work effectively as a team.

This project prepared us for more advanced computer science courses and gave us confidence in tackling complex programming challenges.