Debugging and advanced memory

Raul P. Pelaez

October 31, 2024

Contents

1 Debugging your code

The best way to debug is to not have bugs in the first place. But that is of course impossible.
Still, with time and experience, as you learn more design techniques and you fight with bugs, you will get better at writing code that is less prone to bugs. For instance, you will quickly find that there is a high correlation between complexity and bugs.
Write code that is as high level as possible (use libraries, abstractions and the standard library) and as simple as possible. I know complexity is a vague term, but its something you will learn to feel with time. Unit testing also helps with this. But with the, perhaps unfortunate, consequence of these practices is that we will typically often end up with only the nastiest and elusive bugs left to debug.

Advice

Let me drop some quotes to inspire you:

> "The best code is no code at all"
> Jeff Atwood

> "The code easiest to maintain is the code that was never written."
> Robert Galanakis

> "Premature optimization is the root of all evil"
> Donald Knuth

Debugging code is a craft that takes a whole career to master, but there are some basic techniques that can help you get started and that cover 99% of the cases. It requires equal parts of tooling as it does of sheer intuition.
Lets explore these basic techniques.

1.1 Using prints

The simplest way to debug your code is to print the values of the variables at different points in your code. Sometimes it is hard to tell where the code is failing/crashing. We can do a kind of binary search in the code by adding print statements in the middle of the code and see if they are printed. If they are, then the bug is after that point, if they are not, then the bug is before that point.
You can also use these prints to check if the values of the variables are what you expect them to be.
It sounds too "bare" to be useful, but you will be surprised how often this is the only thing you need to find the bug.

Advice

The effects of printing can also be telling on their own. Sometimes just adding the print statement will make the bug disappear.
Adding the print instruction causes the code to be recompiled and reorganized in memory, which can change the behavior of the code. This is typically telling of something like an uninitialized variable, or in general something related to undefined behavior.
In concurrent code, printing can also change the behavior of the code, as it can change the timing of the threads. If printing suddenly fixes the code, you probably have a race condition.

Libraries like spdlog can help you print in a more organized way, and you can easily turn off the prints in release mode. Spdlog plays along well with CMake. Its really simple to use, check it out.

1.2 Using gdb

The GNU debugger (gdb) is a powerful tool that allows you to run your code step by step, inspect the values of the variables, set breakpoints, and even modify the values of the variables during runtime. Its available in conda. gdb is an immensely deep tool, but you can get a lot of mileage out of it with just a few commands, which we are going to cover here.
Debugging with gdb effectively requires the executable to be compiled with debugging information. You can do this by adding the -g flag to the compiler. You can also add the -O0 flag to avoid optimizations that can make the code harder to debug.
Debugging information includes inside the binary things like the names of the variables, the names of the functions, and the line numbers of the code. This is what allows gdb to show you the line of code where the program is currently stopped.

Advice

gdb and CMake
You can add debugging information to all targets just by calling cmake like this:

Source code

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

This will add the -g flag to the compiler. You can also add some other useful debugging flags by adding this to your top level CMakeLists.txt:

Source code

```
if(CMAKE_BUILD_TYPE STREQUAL "Debug")
  if (MSVC) # Check if we are using the Visual Studio compiler
    add_compile_options(/W4 /Od)
  else() # We are using g++, clang++, or another compiler
    add_compile_options(-O0 -Wall -Wextra -Wpedantic)
  endif()
endif()
```

Advanced

Heisenbugs: Bugs that disappear when you look at them
Some times you will find that compiling with debug information, without optimizations, or when running it through gdb, the bug not be triggered.
This is 99.9% of the time due to the program incurring in undefined behavior: Assuming an uninitialized variable stores 0, a class not initializing its members correctly, things like that.
You might also be facing an actual bug in the compiler or the OS, but that is extremely rare.

Info

Setting breakpoints
You can set breakpoints in your code by typing "break filename:linenumber" (or just "b"). For instance, "break main.cpp:10" will set a breakpoint in line 10 of the file main.cpp. You can also set breakpoints in functions by typing "break functionname". For instance, "break main" will set a breakpoint in the main function.
gdb will print the line where the breakpoint is set and will pause the execution of the program when it reaches that line.

Info

Running the program
You can run the program by typing "run" (or just "r"). This will start the program and run it until it reaches a breakpoint or crashes.

Info

Stepping through the code
You can step through the code by typing "step" (or just "s"). This will execute the next line of code and pause the execution of the program.

Info

Continuing the execution
You can continue the execution of the program by typing "continue" (or just "c"). This will run the program until it reaches a breakpoint or crashes.

Info

Printing the value of a variable
You can print the value of a variable by typing "print variablename" (or just "p"). This will print the value of the variable to the console.

Info

Setting the value of a variable
You can set the value of a variable by typing "set variablename = value". This will set the value of the variable to the specified value.

Example
Lets debug this code

Source code

```
#include <iostream>
int main(){
  int x = 0;
  x = 10;
  std::cout<<x<<std::endl;
  return 0;
}
```

The following session will set a breakpoint in the main function, run the program, and step through the code. Then it will print the value of x and set it to 123123, and continue the execution of the program.

Source code

```
g++ -g -O0 myprog.cpp # Compile with debug info
$ gdb -q ./a.out # Run gdb and be quiet
(gdb) break main # Set a breakpoint in the main function
Breakpoint 1 at 0x1139: file myprog.cpp, line 3.
(gdb) run # Run the program
Starting program: ./a.out
Breakpoint 1, main () at myprog.cpp:3
3           int x = 0;
(gdb) step # Step to the next instruction
4           x = 10;
(gdb) step # Step to the next instruction
5           std::cout<<x<<std::endl;
(gdb) print x # Print the value of x now
$1 = 10
(gdb) set x = 123123 # Set the value of x to 123123
(gdb) continue # Continue the execution of the program
Continuing.
123123
[Inferior 1 (process 112759) exited normally]
(gdb)
```

1.3 Key new concepts

Info

Debugging
The process of finding and fixing bugs in your code. There are many techniques to do this, from the simple print statement to the powerful gdb.

Info

gdb, the GNU debugger
A tool that allows you to run your code step by step, inspect the values of the variables, set breakpoints, and even modify the values of the variables during runtime.

2 Smart pointers

Memory management is one of the most error-prone aspects of programming. Smart pointers are a type of object that acts like a pointer, but also manages the memory of the object it points to. Smart pointers implement the RAII (Resource Acquisition Is Initialization) idiom, which ensures that resources are properly managed and released when they are no longer needed. Additionally, they provide copy and move semantics and clear ownership semantics.
Consider this code:

Source code

```
void foo(){
  int* ptr = new int(42);
  // .. stuff ..
  if (condition) {
    // Oops! We forgot to delete ptr
    return;
  }
  delete ptr;
}
```

Manual memory management is error-prone, as it is easy to forget to release the memory, leading to memory leaks. Smart pointers help to avoid these issues by automatically managing the memory for you.

Advice

You should strive for a "zero new" policy in your code. That is, avoid using raw pointers and new/delete as much as possible. Instead, use smart pointers to manage memory automatically. (Do not even consider using malloc/free).
You should always try to use the highest level of abstraction, in particular, try this order when managing resources:

1. Use the standard library containers (`std::vector`, `std::map`, etc).
2. Use the standard library smart pointers (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`).
3. Raw pointers and new/delete should be your last resort.

There are three types of smart pointers in C++:

2.1 `std::unique_ptr`

Info

`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope.
Key features:

- Exclusive ownership: Only one `unique_ptr` can own the object at a time.
- Automatically deletes the owned object when it goes out of scope.
- Cannot be copied, but can be moved.
- Lightweight and efficient (no overhead compared to raw pointers).

Example

Source code

```cpp
#include <memory>
void foo(){
  std::unique_ptr<int> ptr = std::make_unique<int>(42);
  // Use ptr like a regular pointer
  *ptr = 100;
  // No need to delete, memory is automatically freed when ptr goes out of scope
}
```

Advice

Use `std::unique_ptr` is very useful when dealing with polymorphism, for instance:

Source code

```cpp
std::unique_ptr<Shape> read_shape(std::istream &is) {
  std::string shape_type;
  is >> shape_type;
  if(shape_type == "circle") {
    // Like new
    return std::make_unique<Circle>(center, radius);
  }
  //... other shapes ...
}
```

Since smart pointers behave just like raw ones, they follow the casting rules of raw pointers. For instance, you can cast automatically a derived class to a base class.

2.2 `std::shared_ptr`

Info

`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Multiple `shared_ptr` objects may own the same object, and the object is destroyed when the last remaining `shared_ptr` owning it is destroyed. By using shared pointers, you are guaranteed that you never have an "use-after-free" bug.
Key features:

- Shared ownership: Multiple `shared_ptrs` can own the same object.
- Keeps track of how many `shared_ptrs` own the object (reference counting).
- Automatically deletes the owned object when the last `shared_ptr` is destroyed.
- Can be copied and moved.
- Slight performance overhead due to reference counting.

Example usage:

Source code

```cpp
#include <memory>
void foo(){
  std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
  {
    auto ptr2 = ptr1; // Both ptr1 and ptr2 now own the same object
  } // ptr2 goes out of scope, but the object is not deleted
} // ptr1 (last reference to object) goes out of scope, object is deleted
```

2.3 std::weak_ptr

Info

`std::weak_ptr` is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by `std::shared_ptr`. It must be converted to `std::shared_ptr` in order to access the referenced object.
Key features:

- Does not participate in owning the object.
- Does not affect the object's lifetime.
- Must be converted to a `shared_ptr` to access the object.

Example usage:

Source code

```cpp
#include <memory>
std::shared_ptr<int> shared = std::make_shared<int>(42);
std::weak_ptr<int> weak = shared;
// To use the weak_ptr, we need to lock it to create a shared_ptr
if (auto locked = weak.lock()) {
  std::cout << *locked << std::endl; // Prints 42
 } else {
  std::cout << "Object has been deleted" << std::endl;
 }
```

Advice

`std::weak_ptr` is one of those things that you will seldom use.

2.4 Custom deleters

Sometimes you need to manage resources that are not simple memory allocations. For instance, you might need to close a file, release a mutex, or free a resource that is not a simple pointer. In these cases, you can use custom deleters with smart pointers.
Custom deleters are functions or function objects that are called when the smart pointer goes out of scope. They allow you to perform custom cleanup actions when the smart pointer is destroyed.
Example

Source code

```cpp
#include <memory>
#include <iostream>
void close_file(FILE* file) {
  std::cout << "Closing file" << std::endl;
  fclose(file);
}
void foo(){
  std::shared_ptr<FILE> file(fopen("file.txt", "r"), close_file);
  // Use file like a regular pointer
} // close_file is called automatically when file goes out of scope
```

Note that the functionality in this code is already provided by `std::fstream`, but this is just an example.

2.5 Key new concepts

Info

Smart pointers
Types that behave like pointers, but manage the underlying memory for you. There are three types:

`unique_ptr`: A pointer that owns the memory it points to. When it goes out of scope, the memory is freed.
`shared_ptr`: A pointer that can be shared among multiple owners. The memory is freed when the last owner goes out of scope.
`weak_ptr`: A pointer that can observe a `shared_ptr` without owning it. It can be used to avoid circular references.
We can control how the resource is released by using custom deleters.

3 Exercises

3.1 Debugging codes

Goal

Lets explore some basic debugging techniques:

- Just printing
- Using gdb

For that, we are going to use a simple code that has a couple bugs that cause crashes.

Source code

```
#include <iostream>
using namespace std;
void explode() {
  int* ptr = nullptr;
  *ptr = 10;
}
class Buggy{
  int value = 0;
public:
  Buggy(int value) {
    value = value;
  }
  void do_something() {
    auto mod = 123%value;
    std::cout << "The remainer of 123 divided by value is " << mod <<
    ↪  std::endl;
  }
};
int main() {
  Buggy b(10);
  explode();
  b.do_something();
  return 0;
}
```

Learning goal

Familiarize with finding bugs in codes.

Milestone

Use `std::cout` to find the lines that is causing the crashes and fix the bugs.

Milestone

Restore the buggy version and use gdb instead. We will not modify it further.
Set breakpoints in the lines that are causing the crashes and print the values of the variables to see what is going on.

hint

You will need to compile the code with the -g flag to include debugging information.
Its a good idea to use the -O0 flag to avoid optimizations that can make the code harder to debug.

Milestone

Use gdb to modify the values of the variables during runtime, so that the program doesn't crash.

hint

You can use the "set" command to modify the value of a variable.
Calling "new" is a bit awkward inside gdb, you need to write "operator new(type)" instead.

Advanced milestone

Privilege escalation 2
Go back to the privilege escalation exercise from some weeks ago. You can solve it without writing a single line of code.
Try to do so by running the program in gdb.

3.2 Smart pointers

Goal

Practice using smart pointers to manage memory and understand their behavior in various scenarios.

Learning goal

Gain practical experience with `unique_ptr` and `shared_ptr`, and understand when to use each type of smart pointer.

Milestone

Using `std::unique_ptr`
Create a simple class called Resource with a constructor that prints "Resource acquired" and a destructor that prints "Resource destroyed". Then, write a function that creates a `std::unique_ptr` to a Resource object and returns it. In the main function, receive this `unique_ptr` and demonstrate how it automatically manages the lifetime of the Resource object.
Contrast it with a raw pointer.

Learning goal

Understand the logical flow of a code that uses smart pointers.

Milestone

Fix this code by using smart pointers in the fun() function.

Source code

```
#include <iostream>
using namespace std;
struct Database {
  std::vector<int> data;
  Database(): data(1000000){}
};
void fun(){
    Database* p = new Database();
}
int main(){
    // Infinite Loop
    while (1) fun();
}
```

Warning: If you run this code as is, it can crash your computer. Be careful.

Learning goal

See how smart pointers can help you avoid memory leaks.

Advanced milestone

`std::shared_ptr` with custom deleter
Allocate a `FILE` pointer with `fopen` and use a `shared_ptr` with a custom deleter to close the file when the last shared pointer to it goes out of scope.

Advanced milestone

Custom Deleter with `std::unique_ptr`
Create a `std::unique_ptr` that manages a dynamically allocated array of integers. Implement a custom deleter function that properly deallocates the array and prints "Array deleted" when called.
Demonstrate the usage of this custom deleter with the `unique_ptr`.

hint

`std::make_unique` does not allow custom deleters, so you will need to use the constructor itself and specify its template arguments. Use the `decltype` keyword to get the type of the deleter function.