**Reinvigorating Rogue with Modern C++ Coding**

I would like to begin by stating that this project was not at all easy for me. This code was so old that it didn't even compile in C using GCC out of the box. I had to make a few simple changes, as the language had changed a bit where it no longer supported something the old code was trying to do. Converting the code to be supported in C++ using g++ was a much less trivial task. The big issue to tackle here was how much differently functions are treated now compared to 1980s style C. Once this was done, the compiler just took me through file by file, alphabetically, solving some errors in each file.

The finger-breaking work that this project required was changing the function parameters of every function that had any parameters. In C, when Rogue was developed, function headers and declarations did not feature any parameters between the parentheses. The parameter lists were between the function header and the function definition, so between the header and curly braces. For example:

int add ( )

int x, y;

{

int z=x+y;

return z;

}

Modern C++ and C++ compilers look for those parameters within the parentheses of the function headers and declarations.

In addition to changed parameter lists, it was also not commonplace for functions to have declared prototypes at the start of whichever files used them. It is now common practice to use function prototypes for any function, and one will get a compilation error when a function is called before the compiler sees at least a prototype, although one can put the function definition above the call if avoiding prototypes is desired. This was not necessary in old C. Separate compilation was a nightmare, as function prototypes were not even necessary in files that called the function but did not have the function implementation. To abate this, I found the easiest fix was to add function prototypes in rogue.hpp, which is #included in every .cpp file of Rogue, for every function that gave me a compilation error of "not declared in this scope".

Another unexpectedly difficult task was realizing that certain functions being used were no longer properly supported by the libraries the code had included. There were a few times I had to go find the right library that should be used to support certain functions, such as getpwuid. I'm not certain if I'm correct in thinking the libraries had changed somehow, or if I just needed new libraries to support the environment that I am now running the program on nearly 40 years later.

Once all the compilation errors were solved, I then had a whole slew of no problems: linker errors. There were a few errors of the same kind, but the nature of the error is essentially that multiple of the same object(s) were being instantiated for each file that object(s) was being used in. The main culprit of this was the d\_list struct from rogue.hpp, which was trying to be instantiated in numerous files, and the linker didn't like this. To combat this, I declared the object(s) as extern. Based on the coding, I'm assuming the object(s) were implicitly defined as extern back then, but this is no longer supported. Here is a before and after of the fix for d\_list:

Before

﻿

|  |
| --- |
| struct delayed\_action { |
|  |

|  |
| --- |
| int d\_type; |
|  |

|  |
| --- |
| int (\*d\_func)(); |
|  |

|  |
| --- |
| int d\_arg; |
|  |

|  |
| --- |
| int d\_time; |
|  |

} d\_list[MAXDAEMONS];

After

|  |
| --- |
| struct delayed\_action { |
|  |

|  |
| --- |
| int d\_type; |
|  |

|  |
| --- |
| int (\*d\_func)(); |
|  |

|  |
| --- |
| int d\_arg; |
|  |

|  |
| --- |
| int d\_time; |
|  |

};

...

﻿extern struct delayed\_action d\_list[MAXDAEMONS];

Finally, I made some slight changes to make the code feel more modern. For instance, I got rid of all uses of the word "register", as compilers can handle this on their own these days for improved performance. In addition, I got rid of all instances of C's "malloc" in favor of C++'s "new" keyword. I also tried to add in some object oriented programming flare by incorporating some inheritance hierarchy. I didn't actually use inheritance in the nature it is supposed to be used, I simply publicly inherited a few things and made it compile. The reason for this is that I didn't want to critically alter the code when it needs to maintain is core functionality anyway. What I really did was make "magic\_item" and "traps" as "objects". The reason I did this was so that they could maintain their functionality as their respective structs, but, perhaps, in a new version of Rogue, code could be written to use them as "objects". This could be useful if the player wanted to use them in an unconventional fashion, such as hurling the object for damage. Or, maybe, there could be armors of "magic\_type" for defense, and then there would be merit to having a "magic\_type" armor inherit from "object", which has "o\_ac" for armor type. I also made the "monster" struct inherit from "things", as "things" are defined as people or monsters by the original developers. In general, I think inheritance is useful in these cases to treat "objects" and "things" in a more realistic sense. Here is the code change:

struct object {

int o\_type; /\* What kind of object it is \*/

coord o\_pos; /\* Where it lives on the screen \*/

char o\_launch; /\* What you need to launch it \*/

char o\_damage[8]; /\* Damage if used like sword \*/

char o\_hurldmg[8]; /\* Damage if thrown \*/

int o\_count; /\* Count for plural objects \*/

int o\_which; /\* Which object of a type it is \*/

int o\_hplus; /\* Plusses to hit \*/

int o\_dplus; /\* Plusses to damage \*/

int o\_ac; /\* Armor class \*/

int o\_flags; /\* Information about objects \*/

int o\_group; /\* Group number for this object \*/

};

struct magic\_item : public object {

char\* mi\_name;

int mi\_prob;

int mi\_worth;

magic\_item(char\* a, int b, int c) : mi\_name(a), mi\_prob(b), mi\_worth(c) {};

magic\_item(char\* a, int b) : mi\_name(a), mi\_prob(b) {};

};

struct trap : public object {

coord tr\_pos; /\* Where trap is \*/

char tr\_type; /\* What kind of trap \*/

int tr\_flags; /\* Info about trap (i.e. ISFOUND) \*/

};

The reason I say that I didn't use inheritance here the way it is intended is because I didn't actually exploit the benefit of the derived class being able to take from the base class anywhere in the code. I simply made constructors for the objects so that the code could continue utilizing brace enclosed initialization lists to instantiate members of the derived classes where necessary. I would've had to have dug really deep into the code and make serious changes to exploit the true nature of inherited properties, and I wanted to keep the game functioning as is.