Monetization of Development Tools for Fun and Profit

Albin Eldstål-Damlin
Your Company Here
Reasonable rates apply
albin@legit.name

***Abstract*—In this paper we propose methods and strategies to raise profit from freely available and open-source development toolchains such as GCC. We illustrate techniques to maximize ~~player~~ developer engagement and drive further purchases once the system is in place.**

***Index Terms*—monetization, microtransactions, free-to-play, gamification**

I. Introduction

In the past decade, monetization of free content has become a leading business strategy for companies in a range of tech industries such as computer games and mobile apps. A wide variety of techniques have been developed and refined to drive users to continuously pay for additional content, either in a storefront fashion or using a more randomized "loot-box" approach. This has caused a surge of "Freemium" titles, requiring no up-front purchase and recouperating development costs by selling in-game items.

A previously untapped market segment is that of *software development tools*, such as compilers, static analysis tools, debuggers, etc. Many of these are available either free of charge or under an open-source license, making them perfect candidates for freemium-style monetization.

In this paper, we examine `g++`, the C++ compiler of the GNU Compiler Collection (GCC). Consulting its manual [1], we find that the standard distribution comes with an ample selection of options and switches; this suggests that there is room for expansion.

We explore the possibilities of adding microtransactions to the `g++ frontend.`

II. User Motivation

The first problem is convincing the user to enter the ecosystem, taking the step from Libre to Freemium. To entice, we must introduce a *killer feature* that is not available elsewhere. What sets a *killer* feature apart from a regular feature is that the killer feature is unique and indispensable. This feature can be entirely cosmetic (such as a novel output decoration), but it is preferable to give the user some qualitative improvement. To come up with such a feature, we identify a common problem users have with our chosen product and devise a solution.

In the case of `g++`, a common complaint is the verbosity and obscurity of some error outputs. In many small- and mid-sized codebases, a typo can lead to error messages exceeding the size of the code! Furthermore, it isn't always clear to the novice programmer what the cause is of these pages of output. Listing 1 shows an example of an STL type error [2], with the full error message from `g++` in appendix A.

```
#include <vector>
#include <algorithm>

using std::vector;
using std::find;

int main()
{
  int a;
  vector< vector <int> > v;
  vector< vector <int> >::const_iterator it
      = std::find( v.begin(), v.end(), a );
}
```

Listing 1. A program with confusing errors

A good candidate for a killer feature, then, is error output of improved quality, readability and accuracy. The implementation of such a feature is beyond the scope of this article.

III. Intermission

If you have a software project, monetization effort or just want to show off pictures of your cats to the world, you **need** a website. Don't have any coding skills? Don't know where to start? Check out SquarePeg, one of the world's leading providers of all-in-one web hosting solutions. SquarePeg lets you effortlessly create your site using one of *four* beautiful templates. It's as easy as drag-and-drop. Almost nothing to install, almost nothing to update, almost ever! Visit http://www.squarepeg.com/sigbovik for 20% off your first purchase today!

IV. In-tool currency

A key technique to drive purchases is an intermediate single-purpose currency, which can either be earned by using the tool or directly purchased for real-life money. Such a currency serves to disconnect the actual cost of offered add-ons from the apparent cost, in addition to locking in a greater amount of real-world money by inconvenient exchange rates.

In our examples, we introduce the `g++`-specific "L2 Cash" (L$) and present all purchase options to the user (except for L$ itself) in L$.

The user is rewarded in small amounts of L$ for various normal use of the software (for example 1L$ per 10 seconds

spent compiling code) as well as extraordinary achievements (compile with substantial changes without errors on the first try? 1L$ per 10 lines of code!).

Another avenue for monetization is to sell ad-space and reward the user for being exposed to advertising. Targeting g++ opens the non-traditional option of modifying the standard library, for example by randomly including advertising in printf() output in exchange for L$.

V. Suitable Features

Traditionally, premium add-ons can be divided into three categories along a spectrum: *Cosmetic*, *Convenience* and *Pay-to-Win*. The special nature of a compiler also offers a fourth option which is useful to us: *Non-standard language features*.

Cosmetic features do not alter the user's experience, only the apperance of some elements. In an online game, this may be a special player avatar or some rare piece of equipment that nevertheless does not afford the player any advantage. There are relatively few possibilities for a purely cosmetic add-on to a productivity tool such as a compiler.

Convenience features can automate or simplify otherwise tedious tasks, saving the user time but not otherwise providing any competitive edge. An example might be automation of *grinding*, mindless mass-production work. In a development setting, this is akin to well-crafted preprocessor macros and automatic generation of boilerplate code.

Pay-to-Win features directly improve a user's chance of success. In a competitive game, this may be a stronger weapon or a higher-capacity backpack. These features are generally ill-received, as they are perceived to throw off the balance of the game. Since development tools are usually single-player, or cooperative multiplayer, these features are better thought of as productivity-enhancements.

Non-standard language features offer an avenue for lock-in, by luring the user into writing programs that will not work without our premium add-ons. This is ideal for monetization, and a particularly good feature could even warrant recurring payment. We will also exploit features like this for licensing reasons, detailed in Section VII.

VI. User Interface

The most obvious way to expose new extensions to the user is via command-line switches. We use the + prefix to show features that can be purchased, together with their price.

```
albin@SquarePeg:~$ g++ --help
Usage: g++ [options] file...
Options:
...
  -pass-exit-codes    Exit with highest error code from a phase

Premium Features:
  +nice-errors        Provide human-readable output (500L$)
```

A common optimization is to sneak the premium options in among the free ones, to make them more difficult to ignore.

```
albin@SquarePeg:~$ g++ --help
Usage: g++ [options] file...
Options:
...
  -pass-exit-codes    Exit with highest error code from a phase
  +nice-errors        Provide human-readable output (500L$)
```

To drive conversion rate, however, it is useful to advertise the available options more strongly. A color hint is a good start.

```
albin@SquarePeg:~$ g++ --help
Usage: g++ [options] file...
Options:
...
  -pass-exit-codes    Exit with highest error code from a phase
  +nice-errors        Provide human-readable output (500L$)
```

At the end of every output, we append the user's current account balance. This provides a reminder that the player is earning while they play. Of course, we also remind them how to easily increase their account balance.

```
albin@SquarePeg:~$ g++ --help
Usage: g++ [options] file...
Options:
...
  -pass-exit-codes    Exit with highest error code from a phase
  +nice-errors        Provide human-readable output (500L$)

  Balance: 1210L$
  +cash=[amount]      Add additional L$ to account ($2.99 / 100L$)
```

Another well-known strategy is to provide tiered pricing, i.e. a better exchange rate for bulk currency purchase. By identifying beginners (more likely to make small purchase) and power-users (more likely to make bulk purchases), we can gently nudge them toward a transaction.

```
albin@SquarePeg:~$ g++ --help
Usage: g++ [options] file...
Options:
...
  -pass-exit-codes    Exit with highest error code from a phase
  +nice-errors        Provide human-readable output (500L$)

  Balance: 10L$
  +cash=100           Additional 100L$ to account ($2.99)
  +cash=1000          Additional 1000L$ to account ($25.99) <- Most popular
  +cash=5000          Additional 5000L$ to account ($39.99)
  +cash=10000         Additional 10000L$ to account ($69.99) <- Best deal
```

VII. Skirting Open-Source Licensing

g++ is released under the GNU General Public License (GPL), which requires that any modification or addition is also released under the same license. To counteract this, we devise a strategy for compliance with *the letter* of the license while side-stepping the *spirit*.

By ensuring that our premium features are implemented with heavy reliance on our own non-standard language features, we prevent the spread of our proprietary modules to non-paying users. By extension, if we also hide the documentation of these non-standard features behind our pay-wall, we inhibit non-paying users trying to port the released source code to free tools.

References

[1] G. Project, "Gcc manual: Invoking gcc." https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc/Invoking-GCC.html#Invoking-GCC, 2018.

[2] Kebs, "Code golf entry." https://codegolf.stackexchange.com/a/10470, 2016.

APPENDIX A
TYPICAL G++ OUTPUT

```
In file included from /usr/include/c++/7.3.1/bits/stl_algobase.h:71:0,
                 from /usr/include/c++/7.3.1/vector:60,
                 from explod.cpp:1:
/usr/include/c++/7.3.1/bits/predefined_ops.h: In instantiation of 'bool __gnu_cxx::__ops::
    _Iter_equals_val<_Value>::operator()(_Iterator) [with _Iterator = __gnu_cxx::
    __normal_iterator<std::vector<int>*, std::vector<std::vector<int> > >; _Value = const int]':
/usr/include/c++/7.3.1/bits/stl_algo.h:120:14:   required from '_RandomAccessIterator std::
    __find_if(_RandomAccessIterator, _RandomAccessIterator, _Predicate, std::
    random_access_iterator_tag) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<std::
    vector<int>*, std::vector<std::vector<int> > >; _Predicate = __gnu_cxx::__ops::
    _Iter_equals_val<const int>]'
/usr/include/c++/7.3.1/bits/stl_algo.h:3907:28:   required from '_IIter std::find(_IIter, _IIter
    , const _Tp&) [with _IIter = __gnu_cxx::__normal_iterator<std::vector<int>*, std::vector<std
    ::vector<int> > >; _Tp = int]'
explod.cpp:...
/usr/include/c++/7.3.1/bits/predefined_ops.h:241:17: error: no match for 'operator==' (operand
    types are 'std::vector<int>' and 'const int')
  { return *__it == _M_value; }
In file included from /usr/include/c++/7.3.1/bits/stl_algobase.h:67:0,
/usr/include/c++/7.3.1/bits/stl_iterator.h:866:5: note: template argument deduction/
        substitution failed:
/usr/include/c++/7.3.1/bits/predefined_ops.h:241:17: note:   'std::vector<int>' is not derived
/usr/include/c++/7.3.1/bits/stl_iterator.h:337:5: note: candidate: template<class _IteratorL,
    class _IteratorR> bool std::operator==(const std::reverse_iterator<_IteratorL>&, const std::
    reverse_iterator<_IteratorR>&)
     operator==(const reverse_iterator<_IteratorL>& __x,
     ^~~~~~~~
/usr/include/c++/7.3.1/bits/stl_iterator.h:337:5: note:   template argument deduction/
        substitution failed:
```