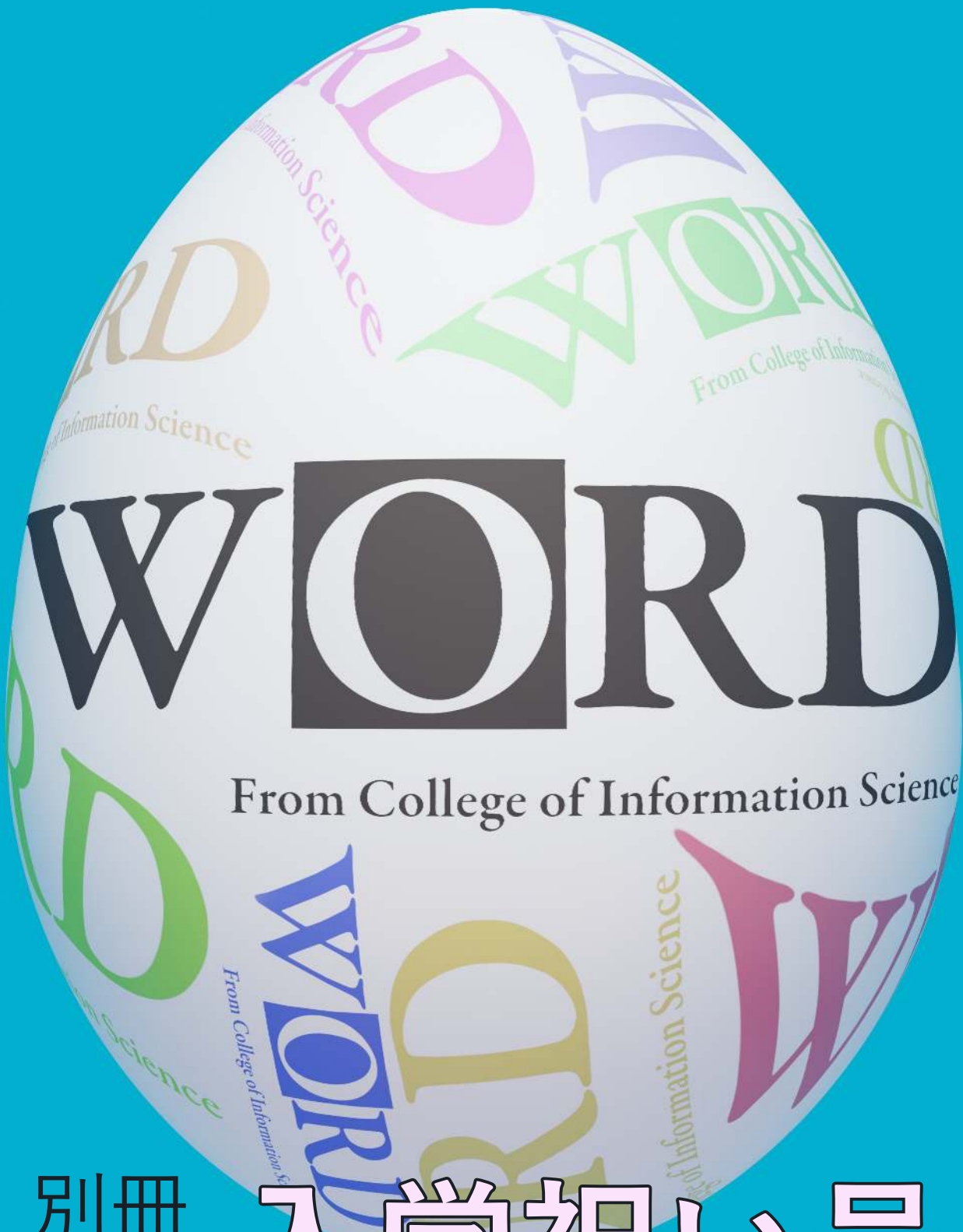

別冊

2020.4

入学祝い号

目次

100 日後に神絵師になるお前

文　編集部　神絵師

1　はじめに

こんにちは、神絵師です。ご入学おめでとうございます。入学式が遅れてしまって大変ですね。ところでこの 1 ヶ月何してましたか？ 不急不要の外出を避けておたくパソコンカタカタですか？ 2021 年のオリンピックに向けたトレーニングですか？ 俺なら……絵を描くね。私は絵を描いたり学んだりするのは中学美術以来なんですが、最近突然ペンタブ買ったんですよ（隙あらば自分語り）[*1]。というわけでみんなもレッツ神絵師

2　準備

なにはともあれ準備が必要です。私はこんな感じの環境です（表 1）。

表 1　神絵師の開発環境

OS	Windows 10
CPU	Intel Core i3 7100
GPU	GeForce 750 Ti
メモリ	DDR4 8GB
ペンタブ	HUION Kamvas Pro 12
ペインティングソフトウェア	Krita[*2]
入門書	室井 康雄『DVD ビデオ付き! アニメ私塾流 最速でなんでも描けるようになるキャラ作画の技術』[*3]
鏡	ニトリで買ったやつ

CPU が i3 だったりメモリが 8GB だったり、ご覧の通り、PC のスペックはかなり省エネモードです。しかし、これでも特にカクつくことなく絵を描きながら某アニメストアでアニメーションを堪能できるので問題ありません。始めるのは比較的かんたんですね。

ペンタブは価格某 com で激安ランク帯からクチコミが良さげなのを適当に探しました。他の選定基準として、ディスプレイの発色の良さ、ファンクションキーの有無などがあります。特に後者は作業効率化に強く貢献します。キャンバスの回転、拡大/縮小やペンの切り替え、戻る/進むなどがボタン操作で簡単に呼び出せるとお絵描きスピードはダンチ[*4]です。また、ペンの性能も重要です。傾き、筆圧、感度は絵のタッチにかなり影響がでます。予算

[*1] 隙を見せる読者が悪い
[*2] https://krita.org/
[*3] https://www.amazon.co.jp/gp/product/4767823900/
[*4] “段違い” という意味だよ

の 8 割くらいはペンタブに注ぎましょう。私が使用している Kamvas Pro 12 のスタイラスペンは充電や電池が不要なのも特徴です。ペンタブ自体も 12 インチ画面 + 太めのベゼル + ファンクションキーなので結構でかいです。

お絵描きソフトは OSS[*5]の Krita を使っています。自分でこだわりビルドをするのもヨシですが、Windows 環境なら公式サイトにあるインストーラでサクッと入れてもヨシです。最初は GIMP[*6]を使ってました。GIMP って知ってますか？ GNU Image Manipulation Program の略です。GNU って知ってますか？ GNU's Not Unix の略です。しかもマスコットキャラクターのウィルバー君がキモいですね。マスコットキャラがキモい、画像編集ソフトであってお絵描きツールではない、しかもマスコットキャラがキモい（2）のでやめましょう。

入門書はなんでも良いですが**必ず買いましょう**。車輪を再発明する前に卒業してしまいます。

鏡はあると便利です。ついでに自分でポージングをおこなって服のシワや身体の動き方も確認できます。顔のパーツの位置も把握できます。多々描かなければ生き残れない。

3 描く

確認ですが、みんな、おたくだよね？ では女の子を描きます。とりあえず表 1にある入門書やここ 2、3 週間の私の経験からいくつかピックアップしていきます。詳細はご購入いただいたお手元の入門書をご覧ください。

3.1 模写

いきなりオリジナル絵を描いても、キュビズムよりも多角な視点からなるパーツの集合体、フォービズムよりも野性の鋭い色彩になるでしょう。とりあえずお手元の画像フォルダからかわゆい画像を取り出してそれを模写していきましょう。

模写するにしても、ただ `cp image1.png image2.png` するが如くドット 1 つずつ丁寧に打っていくのではなく、頭を使って、どの位置にどのパーツがあるのか、というパーツとパーツの位置関係を考えましょう。

3.2 観察

なんとなくかわいくない女の子になりましたか？ かわいい女の子になったらあなたはピカソの生まれ変わりです! これからも頑張ってください。なんとなくかわいくない女の子を描けたあなた! なんとなくかわいくないことに気づけたことは大変素晴らしいです。

では何がかわいくなさの原因なんでしょうか。人間の顔と自分が描いた女の子の顔を見比べてみましょう。人間の顔はパーツとパーツの位置関係が定まっており、この位置関係が人間の顔を顔たらしめています。この位置関係に沿ってますか？ ファンダメンタルなところで言えば、正中線に鼻柱（に該当する部分[*7]）が来てますか？ 他にも、眉間に鼻根が

[*5] GitHub にてプロジェクトがホスティングされている `https://github.com/KDE/krita`

[*6] `https://www.gimp.org/`

[*7] かわいい女の子の絵を描くときはそれなりに情報を省略したほうがかわいくなりがちなので、鼻柱は横顔など

来てますか？ かわいい女の子を描くためには次のようにパーツを動かしてみてください。目の位置は顔の高さの中点あたりにしてみましょう。髪の生え際は、頭頂と目の位置の中点よりやや下にします。眉間と顎先の中点に鼻尖をおき、鼻尖と顎先の中点に口をもってきます。

横顔などの頭が回転した状態を見てみると、顔のパーツの位置関係がやや変わってきます。そして目が訳わからなくなります。しかし、上記の位置関係を守り、そして目の構造をしっかり観察するとなんとなく描けてくるんじゃないでしょうか。目はまぶたと眼球から成ります。まぶたの動作は口に近似でき、そこに眼球を突っ込む感じでパーツが構成できます。左右の位置関係は眉間から対称の位置に……というわけですが頭が回転するとわかんなくなりますね。横顔を観察してみると、視点から奥側の目は若干高めですが、パース的にやや横幅が縮んでいます。さらに角度によっては目尻が見えなくなっています。目頭も鼻柱によって隠れている場合があります。視点から手前側の目は、特に目尻が低くなりますね。対象の視線も、黒目の位置を確認してみましょう。鏡に対してやや顔を回転させて、鏡に映った自分の目を見てみてください。左右の目で位置がやや違うことに気づくと思います。奥側の目は手前側の目よりも黒目が目頭に寄り、逆に手前側の目はやや目尻に寄っていることが確認できます。

……とまあこんな具合に観察していってみてください。そしてまた模写をして、観察して改善して、を繰り返すとなんとかなるような気がします。私はペンタブが届いてから 2 週間以上ほぼ毎日、平均して 1 日 5 時間くらい絵を描き続けた結果、なんとなく人っぽい顔が描けるようになりました。また、この観察によってよく分からなかった人の顔が分かるようになりました。漫画などで記号化された顔に対しても解像度が上がってきて、かわいさを残した顔のうまい抽象化に驚きます。色の塗り方も、どのようにグラデーションを付けたのかとかブラシの形状とかも部分的に読み取れるようになってきました。絵を描く力だけでなく、絵を読み取る力も養えるのは大変に面白いし、ためになります。

胴体や塗りについては……またいつか……。

4 あとは

やるだけ

5 成果物

図 1が 4 日目に描いた星宮いち某ちゃんです。序盤はとりあえず GIMP しか色を塗るツールを知らなかったので GIMP を使ってました。かわいいですが左目がやや頭頂側に持ち上がってしまってますね。チャームポイントのリボンも左側をもう少し引き伸ばしたいです。顎とエラがとんがってるのもかわいさ減です。眉毛も位置がおかしいです。眉は眉骨の上にあるので、顔が傾いたときには目の位置よりも傾き具合が増えます。この絵では眉が逆方向に傾いてます。どういうことなんですかね。対象がかわいいので OK です。GIMP は

でなければ省略しがちです

図 1　あ〜 もう 何もしたくない。

図 2　自分をの野原ひ某しと思い込んでいる一般男性

やめようね。

図 2が 10 日目に描いた自分を野原ひ某しと思い込んでいる一般男性です。このころ Krita に切り替えました。記号的な目が何を表しているのかが分かると簡単に模写できる、狂気を孕んだシンプルな顔です。

図 3が 16 日目に描いた某堂ユリカ様です。血を吸うわよ。かわいいですね。人間の顔の

図 3　某堂ユリカ様

構造にも慣れてきたのでオリジナルの構図です。調子に乗って傘を追加しましたが流石にメチャクチャですね。顎あたりは輪郭をうまくボカすことでエラの張りを隠蔽しています。線の太さの強弱により全体の輪郭をクッキリさせているのも結構いい感じです。おかっぱ

になってるキュートな前髪をうまく表現できていると思います。線画のほうがもう少しかわいかったので、口を濃く塗ればもっとパリッとできそう。左目ももうちょっと開眼したほうがバランスが良いです。眉毛をもう少し伸ばしたほうがいいですね。髪にも影をもっとかけてあげると立体感が出ます。右肩はもう少し奥側に持っていくといいでしょう。

6　おわりに

いかがでしたか？ プログラミングよりも絵を描いたほうが多分人生が楽しくなると思いますんで是非やってみてね。ところで最近は有界量化について興味がでてきました。みなさんはご存知でしょうか？ 多相な型を持つ体系において型変数に supertype の upper bound を指定できるんですよ。upper bound の指定が無いのは Top 型が upper bound になっていると考えることで型システムをスッキリ定義できます。楽しいですね! 大堀 淳『プログラミング言語の基礎理論』や Benjamin C. Pierce『Types and Programming Languages』26 章が詳しいです。F-有界量化や高階のカインドのある体系での有界量化など、まだまだ掘れる部分がたくさんあります。さらに gradual typing のような ⋆ 型が追加されると supertype との兼ね合いはどうなるんでしょう。わたし、気になります! ま、神絵師には関係ねぇ話なんだがヨ……。

IOCCC の作品を味わう

文　編集部　algon

1　はじめに

突然ですが、みなさんは **IOCCC** というプログラミングコンテストをご存知でしょうか。IOCCC は The International Obfuscated C Code Contest の略称で、日本語に直せば「国際難読化 C コードコンテスト」の意味です。IOCCC は応募されてきた C 言語コードを「読みにくさ・複雑さ」などの観点から評価し、最も優れた (つまりは最も酷い) 作品を決めるコンテストです。

本記事では、IOCCC がどのようなコンテストなのかを説明し、過去の入賞作品を紹介しながら、その面白さを知っていただきたいと思います。

1.1　環境

本記事の実行例は以下の環境で実行されたものです。

- Ubuntu 18.04
- GCC 8.3.0

2　IOCCC について

2.1　ホームページ

全てはホームページ (`https://www.ioccc.org/`) にあります。公式ツイッターアカウント (@ioccc) も要チェック[*1]です。

2.2　コンテストの目的

IOCCC のホームページにはコンテスト開催の目的として以下のようなことが挙げられています。

- ルールの範囲内で最も難読な C プログラムを書くこと。
- プログラミングスタイルの重要さを皮肉な方法で示すこと。
- 異常なコードで C コンパイラに負担をかけること。
- C 言語の微妙な点を明確にすること。
- 粗雑な C コードに対する安全な公開討論の場を提供すること。

[*1] メディアツイートからは審査員達の美味しそうな食事の写真を見ることができる。

2.3 ルール・審査

現在では以下の 3 人の審査員が応募作品を審査しています。

- Leonid A. Broukhis – http://www.mailcom.com/main.shtml
- Simon Cooper – http://www.sfik.com/
- Landon Curt Noll – http://www.isthe.com/chongo/

IOCCC にはいくつかのルールがあり、ルール内であれば基本的にどんなコードでも許されています。例えば、IOCCC 2020 のルールは 26 項目からなり、その中には

- 2a) ソースコードのサイズは 4096 バイト以下であること。
- 7) オリジナルの作品であること。
- 12) 合法的なルールの悪用は多少は推奨される。審査員がルールを破っているとみなした作品は失格になるため、ルールを意図的に破るのであればそれを正当化する理由を述べなければならない。

などのルールがあります。ルールは応募作品に応じて毎回少しずつ変更されています。開催年毎のルールの差分を見比べるのも面白いでしょう。

また、審査は「誰の作品であるかを隠した状態」で行われます。これによって作品の内容だけで純粋に評価されるようになっています。投稿できる作品数には制限が無いので、同じ作者の作品が複数入賞することもあります[*2]。

細かい評価の基準は明言されていませんが、コンテストのガイドラインには、好まれる作品の傾向や応募する際の注意点、詳しい選出のプロセスなどが書かれています。

2.4 起源

FAQ ページ (https://www.ioccc.org/faq.html) には次のようなエピソードが書かれています。

> 1984 年 3 月 15 日、Landon Curt Noll と Larry Bassel は同じ会社で別々のバグ修正の仕事をしていました。Landon Curt Noll は初期の BSD の finger コマンドのバグを修正していて、Larry Bassel は Bourne Shell のバグを修正していました。彼らは頭を休ませようと会社の廊下をうろついていたところで偶然出会い、お互いの扱っているコードが信じられないほど酷いということについて盛り上がりました。これがきっかけとなり、彼らはネットニュースで難読な C 言語コードを募集してみることにしました。

このときの募集が第 1 回の IOCCC です。

Bourne Shell の実装ではいくつかのマクロが使われていて、これによってソースコードの見た目が ALGOL 風になっています。Bourne Shell のマクロは https://minnie.tuhs.

[*2]実際に、Yusuke Endoh さんは 2013 年、2015 年ともに 4 作品同時入賞を果たしている。

org/cgi-bin/utree.pl?file=V7/usr/src/cmd/sh/mac.h から見ることができます。

3 作品の楽しみ方

IOCCC についての理解を深めたところで、さっそく作品を見ていきましょう。

3.1 作品をダウンロードする

過去のコンテストの入賞作品はホームページの"Winning Entries" (https://www.ioccc.org/years.html) から見ることができます。このページでは過去のコンテストの入賞作品をダウンロードできます。

ここでは 1990 年の作品をダウンロードしてみることにします。Winning Entries ページの「7th International Obfuscated C Code Contest (1990)」という見出しの下にある 1990.tar.bz2 と書かれたリンクからダウンロードできます。この tar ファイルには、コンテストのルール、Makefile [*3]、ソースコード、ヒントなどのファイルが含まれています。

また、作品のファイル群は作者の名前で管理されています。例えば 1990 年の “Peter J Ruczynski"さんの作品には pjr.c、pjr.hint という名前が付けられています。2005 年以降はそれ以前と形式が異なっていて、作品毎のサブディレクトリに prog.c、hint.html などが入っているという形になっています。

3.2 ソースコードを眺める

データをダウンロード・展開したら、まずは適当なソースコードを開いて眺めてみましょう。初めはシンタックスハイライトの無い状態で、ソースコードの見た目を楽しみましょう。

ここでは pjr.c[*4]を開いてみることにします。

Listing 1 pjr.c

```
1  #include <stdio.h>
2  #define A(a) G a();
3  #define B(a) G (*a)();
4  #define C(a,b) G a() { printf(b); return X; }
5  typedef struct F G;A(a)A(b)A(c)A(d)A(e)A(f)A(g)A(h)A(i)A(j)A(k)A(l)A(m)A(n)A(
6  o)A(p)A(q)A(r)A(s)A(t)A(u)A(v)A(w)A(x)A(y)A(z)A(S)A(N)void Q();struct F{B(a)B
7  (b)B(c)B(d)B(e)B(f)B(g)B(h)B(i)B(j)B(k)B(l)B(m)B(n)B(o)B(p)B(q)B(r)B(s)B(t)B(
8  u)B(v)B(w)B(x)B(y)B(z)B(S)B(N)void(*Q)();}X={a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,
9  q,r,s,t,u,v,w,x,y,z,S,N,Q};C(a,"z")C(b,"y")C(c,"x")C(d,"w")C(e,"v")C(f,"u")C(
10 g,"t")C(h,"s")C(i,"r")C(j,"q")C(k,"p")C(l,"o")C(m,"n")C(n,"m")C(o,"l")C(p,"k"
11 )C(q,"j")C(r,"i")C(s,"h")C(t,"g")C(u,"f")C(v,"e")C(w,"d")C(x,"c")C(y,"b")C(z,
12 "a")C(S,"␣")C(N,"\n") void Q(){}main(){X=g().s().v().S().j().f().r().x().p().
13 S().y().i().l().d().m().S().u().l().c().S().q().f().n().k().v().w().S().l().e
14 ().v().i().S().g().s().v().S().o().z().a().b().S().w().l().t().N();}
```

[*3]プログラムのビルド作業などを自動化するための手続きを記述したテキストファイル。make というツール群がこれに従ってビルドを実行する。

[*4]Copyright @ 1990 Peter J Ruczynski (出典：https://www.ioccc.org/1990/pjr.c)

ややこしそうなコードが出てきました。よく見ると後半の中央部に main 関数[*5]が見えますね。#define C(a,b) マクロの中に printf 関数[*6]があるので、何かを出力するプログラムなのでしょうか。読者のみなさんはこのプログラムが何を出力しているか分かりますか？

3.3 コンパイル・実行する

ソースコードを満足するまで眺めたら、実際にコンパイル・実行してみましょう。Makefile が用意されているので基本的には make **作品名**を実行すればコンパイルすることができます。

残念ながら、古い作品の中には現在のコンパイラではコンパイルできないものや、現在の OS では動作しないものもあります。そのような場合には後述するヒントを見てみると解決策が書いてあることもあります。また、作品によっては実行時に引数を要求するものや特殊な入力を想定しているものもあるので、うまく実行できなかった場合にもヒントをチラ見するとよいでしょう。

それでは make pjr を実行してみましょう。

```
$ make pjr
set USE= in Makefile to be ansi or common as desired
or type: make -f ansi.mk for ansi makes
or type: make -f common.mk for common K&R makes
Makefile:49: recipe for target 'pjr' failed
make: [pjr] Error 1 (無視されました)
```

失敗してしまいました。今回選んだ作品は古い回のものなので、ANSI C でコンパイルするか K&R C としてコンパイルするかを選ぶ必要があるようです。最近のコンパイラを使っているなら ANSI C としてコンパイルすればよいでしょう。指示の通りに-f ansi.mk フラグを付けて再び実行します。

```
$ make -f ansi.mk pjr
gcc -O -ansi  pjr.c -o pjr
$ ls pjr
pjr
```

出来上がった実行ファイルを実行してみます。

```
$ ./pjr
the quick brown fox jumped over the lazy dog
```

[*5] C 言語で書かれたプログラムの実行はこの関数から始まる。

[*6] C 言語で書かれたプログラムにおいて、標準出力に文字列を書き出す際に用いられる関数。

「the quick brown fox jumped over the lazy dog」という文字列が出力されました。これは英語のパングラム*7として有名な文ですね。

先程のソースコードからこの出力が予想できましたか？ ソースコードには"the quick brown fox jumped over the lazy dog"という文字列は見当たりませんでしたが、どうやって出力しているのでしょうか。

3.4 ヒントを読む

作品にはソースコードの他に**作品名.hint** というファイルが付属しています。開催年や作品によって多少の変動はありますが、このファイルには

- 賞の名前
- 作者
- プログラムの実行方法
- 審査員のコメント
- 作者のコメント (無いこともある)
- ネタバレ

などが書かれています。

`pjr.hint` からは、この作品の作者は Peter J Ruczynski さんで「Most Unusual Data Structure」という賞を受賞していることなどがわかります。審査員のコメントとして、次のようなヒント (というか答え) が書いてあります。

> このプログラムは 1 つの文字列を出力するものです。どのように実現されているか分かりますか？
> 「関数ポインタを持つ構造体へのポインタ」を返す関数へのポインタを使うことでなされています！

この作品のヒントには、他にも審査員によるネタバレと作者による説明 (の一部) が ROT13*8 で隠された上で書かれています。

3.5 味わう

作品の面白ポイントを押さえた上でもう一度ソースコードを見てみましょう。心の中で「失礼します」と断りながら `pjr.c` を整形してみます。

```
1 #include <stdio.h>
2 #define A(a) G a();
3 #define B(a) G (*a)();
4 #define C(a,b) G a() { printf(b); return X; }
```

*7 なるべく重複せずにアルファベット 26 文字全てを含む文のこと。
*8 各アルファベットを 13 文字ずらしたアルファベットに置き換える簡易的な暗号化手法。

```
5
6  typedef struct F G;
7
8  A(a)A(b)A(c)A(d)A(e)A(f)A(g)A(h)A(i)A(j)A(k)A(l)A(m)
9  A(n)A(o)A(p)A(q)A(r)A(s)A(t)A(u)A(v)A(w)A(x)A(y)A(z)
10 A(S)A(N)
11
12 void Q();
13
14 struct F{
15   B(a)B(b)B(c)B(d)B(e)B(f)B(g)B(h)B(i)B(j)B(k)B(l)B(m)
16   B(n)B(o)B(p)B(q)B(r)B(s)B(t)B(u)B(v)B(w)B(x)B(y)B(z)
17   B(S)B(N)
18   void(*Q)();
19 } X = {a,b,c,d,e,f,g,h,i,j,k,l,m,
20        n,o,p,q,r,s,t,u,v,w,x,y,z,
21        S,N,
22        Q};
23
24 C(a,"z")C(b,"y")C(c,"x")C(d,"w")C(e,"v")C(f,"u")C(g,"t")
25 C(h,"s")C(i,"r")C(j,"q")C(k,"p")C(l,"o")C(m,"n")C(n,"m")
26 C(o,"l")C(p,"k")C(q,"j")C(r,"i")C(s,"h")C(t,"g")C(u,"f")
27 C(v,"e")C(w,"d")C(x,"c")C(y,"b")C(z,"a")
28 C(S,"␣")C(N,"\n")
29
30 void Q(){}
31
32 main(){
33   X=g().s().v().S().j().f().r().x().p().S().y().i().l().
34       d().m().S().u().l().c().S().q().f().n().k().v().w().
35       S().l().e().v().i().S().g().s().v().S().o().z().a().
36       b().S().w().l().t().N();
37 }
```

ここまで整理すれば仕組みが見えやすくなります。ソースコードの流れをまとめると以下のようになります。

- A マクロによって 26+2 個の関数を宣言

例えば A(a) は G a(); という関数宣言に展開される。

- 26+2(+1) 個の関数ポインタのフィールドを持つ struct F を定義
  例えば B(a) は G (*a)(); というフィールドに展開される。
- struct F 型の値 X を初期化
- C マクロによって 26+2 個の関数を定義 (これらは 1 文字出力して X を返す関数)
  例えば C(a,"z") は G a() { printf("z"); return X; }という関数定義に展開される。
- g を呼び出して、返ってきた X の中の関数ポインタを経由して s を呼び出す、というのを繰り返して文字列を出力する

関数 a、b、c がそれぞれ “z”、“y”、“x” を出力する、といったように関数名と出力されるアルファベットが逆順になっています。関数 S、N はそれぞれスペース文字、改行文字を出力する関数になっています。

結局この作品は、main 関数の中の関数呼び出しの連鎖によって「the quick brown fox jumped over the lazy dog」が出力されているという仕組みでした。

4 作品紹介

IOCCC の作品のうち筆者が個人的に好きな作品をいくつか紹介します。

4.1 1992 年 Best Small Program - Brian Westley

この作品のソースコードは (おそらく) 地球の形に整形されています。

Listing 2 westley.c

```
1            main(l
2          ,a,n,d)char**a;{
3       for(d=atoi(a[1])/10*80-
4     atoi(a[2])/5-596;n="@NKA\
5    CLCCGZAAQBEAADAFaISADJABBA^\
6    SNLGAQABDAXIMBAACTBATAHDBAN\
7    ZcEMMCCCCAAhEIJFAEAAABAfHJE\
8    TBdFLDAANEfDNBPHdBcBBBEA_AL\
9    ␣H␣E␣L␣L␣O,␣␣␣␣W␣O␣R␣L␣D!␣"
10      [l++-3];)for(;n-->64;)
11         putchar(!d+++33^
12              l&1);}
```

Copyright @ 1992 Brian Westley (出典：https://www.ioccc.org/1992/westley.c)

実行結果は次のようになります。

Listing 3　westley.c の実行例

```
$ ./whereami 36 140
              !!!!!!!!!!! !!!             !!!   !!!!!!!
! !!!!!!!!!!!!!!!!  !!!!! !    !      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 !!!!!!!!!!!!!!!!!! !!!!          !  !! !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
          !!!!!!!!!!!!!            !!!!!!! !!!!!!!!!!!!!!!! !!    !
             !!!!!!!!!            !! !   !!!!!!!!!!!!!!!!!!!  "
     !        !!!!  !              !!!!!!!!!!!!!!!!!!!!!!!!!
                !!!!!             !!!!!!!!!!!!!   !!!   !!! !
                     !!!!!         !!!!!!!!!!      !     ! !  !
                     !!!!!!!!          !!!!!                    !!
                      !!!!!!            !!!! !              !!!!!
                       !!!!              !!                !!!!!!!!
                       !!                                   !!  !!     !
                       !
```

緯度・経度をコマンドライン引数で指定すると世界地図上にその地点を表示するプログラムです。実行例では、ちょうどつくば市付近にダブルクオート文字が書かれているのが確認できます。ソースコードの中央付近にある文字列に世界地図がエンコードされているというのが見どころです。

4.2　1993 年 Best One Liner - Mark Plummer

この作品は 1 行だけで書かれたプログラムです。

Listing 4　plummer.c

```
1 char*_,*0;main(S,l)char**l;{*(0=*(l+++S-1)-1)=13,*l[1]=0;for
    (;;)for(printf(*l),_=0-1;_>=*l&&++*_>(S+*0+S)*S;*_--=(S+*0
    )*S);}
```

Copyright @ 1993 Mark Plummer (出典：https://www.ioccc.org/1993/plummer.c)

Listing 5　plummer.c の実行例

```
$ ./plummer 0000000000000000001 any
0000000000066082975
```

上記の実行結果では伝わらないのですが、第一引数で与えた整数をインクリメントしていくプログラムです。全ての桁が 9 になったら 0 に戻ります。第 2 引数は何かを渡す必要があります。「何でもいいけど必要」というのが面白いですね。どういう仕組みで動いているのか考えてみてください。

4.3　その他のおすすめ作品

その他の個人的なおすすめ作品を挙げます。コード理解するために高度な C 言語の知識を要求するものもありますが、どれも動かしてみるだけで楽しめるはずです。詳細は伏せておくので、ぜひ自分の環境で遊んでみてください。

- 1986 年 Most adaptable program - applin
- 1987 年 Best One Liner - korn
- 1987 年 Most Useful Obfuscation - wall
- 1987 年 Best Layout - westley
- 1993 年 “Bill Gates” Award - cmills
- 1993 年 Best Abuse of the C Preprocessor - dgibson
- 1994 年 Best Layout - schnitzi
- 1998 年 Best of Show - banks
- 2000 年 Most Complete Program - tomx
- 2006 年 Most Useful - borsanyi
- 2011 年 Best solved puzzle - hamaji
- 2013 年 Best of show - cable2
- 2015 年 Most Overlooked Obfuscation - endoh2

5 書く

面白いアイデアを思いついた人は実際にコードを書いて IOCCC に応募してみてください。コンテストのガイドラインに入賞するためのポイントが書かれているのでそちらにも目を通すことをおすすめします。なお、IOCCC 2020 の応募は残念ながら既に締め切られてしまいました。ネタを温存しておいて来年の IOCCC に応募しましょう。

IOCCC に使えるテクニックを勉強するには遠藤侑介 著『あなたの知らない超絶技巧プログラミングの世界』(技術評論社、2015 年) をおすすめします。この本は Ruby におけるプログラミングを対象にしていますが、IOCCC のような「超絶技巧なプログラム」を書くための様々な手法を学ぶことができます。

6 おわりに

本記事では IOCCC がどのようなコンテストかを説明し、実際の入賞作品を見てみました。いかがでしたでしょうか。IOCCC のホームページでは本記事で紹介しきれなかったすばらしい作品の数々を鑑賞することができます。

真面目なプログラミングをするときは、コーディングスタイルを意識した読みやすく分かりやすいコードを書くように心がけましょう。読みやすいコードを書くには、Dustin Boswell、Trevor Foucher 著、角 征典 訳『リーダブルコード — より良いコードを書くためのシンプルで実践的なテクニック』(オライリー・ジャパン、2012 年) などを読むのがよいでしょう。

7 拙作

IOCCC の作品には足元にも及びませんが、筆者も難読 (を目指した)C コードを書いてみたので最後に紹介したいと思います。「100 回後に死ぬプログラム」です。

Listing 6　die_in_100_times.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

char*e,*p,F[0xFF],X[],M[];int*u,d,i,z,m;struct
stat*w,t;int*x,o,r,s;void/**/in(){for(;i<0x6;o
+=M[i<<1|1],i++)memcpy(F+o,X+M[i<<1],M[i<<1|1]
);FILE*f=fopen(p,"rb");stat(p,&t);z=t.st_size;
m=O_WRONLY|O_CREAT;e=malloc(z);fread(e,z,1,f);
fclose(f);                              i=0;}char*
l,M[]={0l,                              1+4,7,28,0
,5,35,9+1,                              1*0,5,45,2
},X[]={37,                              1l+99,-27,
-101,-98l,            100               |0,0,1-28,
-66,-116l,                              1-30,-127,
-85,-26*1,                              1-84,1-70,
-29,-127l,                              1-85,-29,-
125,-105l,                              1-30,-125,
-83,-29,-126,-80,-29,-125,-87,-29,-125,-96,32,
-25,-101,-82,10,-26,-82,-117,-29,-126,-118,10,
00};int/**/main(int/**/a,char**v){for(p=*v,in(
);5-s;s=e[i++]-s[X]?0:s+1);r=--e[i];o=++e[i+1]
;printf(F,r+o,o,r);remove(p);r||(exit(0),0x00)
;d=open(p,m,t.st_mode);write(d,e,z);close(d);}
```

Listing 7　die_in_100_times.c の実行例

```
$ gcc die_in_100_times.c
$ ./a.out
100回後に死ぬプログラム 1回目
残り99回
$ ./a.out
100回後に死ぬプログラム 2回目
```

```
残り98回
$ for i in {1..95}; do ./a.out > /dev/null ; done
$ ./a.out
100回後に死ぬプログラム 98回目
残り2回
$ ./a.out
100回後に死ぬプログラム 99回目
残り1回
$ ./a.out
100回後に死ぬプログラム 100回目
残り0回
$ ./a.out
-bash: ./a.out: そのようなファイルやディレクトリはありません
$ ls
die_in_100_times.c
$
```

中央の「100」は別の値に変えることもできます。実行ファイルの中に埋め込まれた定数を書き換えていくことで回数をカウントダウンしています。以上、読みやすいコードで失礼しました。

Carbonara Is All You Need

文　編集部　bc

1　前書き

新入生のみなさんご入学おめでとうございます．この記事は，3 年間カルボナーラの研究をしてきた筆者がカルボナーラの奥深さ，素晴らしさを伝える記事です．

またこの記事には皆さんに伝えたいことがもう一つあります．入学にあたり一人暮らしを始めるという人の中には，食費の節約や自身の健康のために自炊をしようと意気込んでいる人も多いと思います．その考えはとても素晴らしいです．しかし，本当にそれでよいのでしょうか．確かに，食費を節約することや自身の健康は大切です．しかし，それらのことを意識するあまり，安価な食材で作れる料理や手軽に作れる料理に走りがちになってしまいます．もう一度言います．本当にそれでよいのでしょうか．この記事を通して，カルボナーラの素晴らしさに加え，「趣味」として料理をする楽しさを皆さんにお伝えしたいと思っています．

2　カルボナーラとは

カルボナーラとは，チーズ，黒コショウ，パンチェッタ*1，鶏卵を主に用いたパスタです．日本では日本人の好みに合わせてこれらの食材に加え生クリームやにんにくなどが使われます．

カルボナーラの起源には諸説あり，ここではその中から現在もっとも有力であると言われている説を紹介します．それは，第二次世界大戦のローマ解放によってアメリカの一般的な食材とローマの地方料理が結びついたことでカルボナーラが生まれたという説です．当時からローマにはチーズと黒コショウを混ぜたパスタがありました．そこに，イタリア人のシェフが戦争のためにやってきたアメリカ兵が好む卵とベーコンをこのパスタに混ぜたことでカルボナーラが生まれたのではないかと言われています．

現在では，カルボナーラは様々な進化を遂げていて，トマトと唐辛子を入れたレッドカルボナーラやパスタの代わりにうどんを用いたカルボうどん，米を用いたカルボ丼など様々な種類があります．このように，カルボナーラの多様性は時代とともに広がり続けています．次の章では筆者がオススメするカルボナーラに入れると良い食材を紹介していきます．

*1 塩漬けの豚肉

3　カルボナーラに入れると良い食材

3.1　パスタ編

ここではカルボナーラに合うパスタの紹介をします．基本的にカルボナーラにはどのパスタも合いますが，濃厚なソースによく絡みつくように太いタイプのものがよく使われます．

スパゲティ

形状が棒状で太さが 1.7mm〜1.9mm のものをスパゲティと呼びます．スパゲティはスーパーマーケットでも購入しやすく，日本人には食べ慣れているため初心者にはオススメです．

パッパルデッレ

パッパルデッレは幅が 10mm から 30mm で厚さは 2mm ときしめんをさらに幅広くしたような形状をしています．これはあまりスーパーマーケットには売っていませんが，Amazon などで購入することができます．

ファルファッレ

ファルファッレは蝶ネクタイのような形状をしています．このパスタは部位によって歯ごたえが変わるため食感を楽しむことができます．これは Amazon や大きいスーパーマーケットで購入することができます．

3.2　ソース編

カルボナーラのソースは基本的に卵，チーズ，黒胡椒から作られます．チーズ 1 つをとっても，どの種類をソースに使うかでカルボナーラは全く異なった味になります．ここで紹介しているものだけでなく色々なものを試してみてください．

卵

ソースの基本です．全卵を使うか卵黄のみを使うかなど好みが分かれますが，卵黄のみを使うとより濃厚になります．筆者は全卵 1 個，卵黄 1 個を混ぜ合わせるのが好きです．

ペコリーノ・ロマーノ

ペコリーノ・ロマーノはヒツジの乳から作られるチーズです．特徴としてはチーズ自体の塩分が強めであるため，仕上げに使う塩の量には注意しましょう．

パルミジャーノ・レッジャーノ

パルミジャーノ・レッジャーノは牛乳から作られるチーズです．味の特徴としては熟成期間が長いため風味豊かです．また，日本のスーパーマーケットでもよく売られています．

モッツァレラ

モッツァレラは水牛の乳から作られるチーズです．モッツァレラはチーズのなかでも比較的低カロリーと言われていますが，カルボナーラに使われることはあまり多くありません．筆者は，冷蔵庫に余っていたモッツァレラを使ってカルボナーラを作ったところその

理由がよくわかりました.

黒胡椒

カルボナーラの影の主役です．ミルタイプのものがオススメです.

生クリーム

これを入れるか入れないかでインターネット上ではよく議論になる食材です．入れるとソースがよりクリーミーになります.

3.3 具材編

カルボナーラに入れる具材は色々なものが考えられます．自分で好みの具材を見つけてみてください．ちなみに，筆者はシンプルにベーコン or パンチェッタのみを入れたものが好きです.

ベーコン

ベーコンはカルボナーラに入れる具材としては一番ポピュラーな具材です．ベーコンは厚切りにして肉感を楽しんだり，薄切りにしたものをカリカリに焼いて食感を楽しんだりと切り方によっても異なる顔を見せます．また，後述するパンチェッタにはない燻製の香りを楽しむこともできます.

パンチェッタ

ベーコンが塩漬けして熟成させた豚バラ肉を燻製させるのに対して，塩漬けして熟成させた豚バラ肉をそのまま乾燥させたものをパンチェッタと呼びます．ベーコンに比べて燻製の香りがない分肉の旨味をダイレクトに味わうことができます．また，自宅でも簡単に作ることができます.

海老

後述するトマトが入ったレッドカルボナーラを作るときに使います．オリーブオイルで海老の殻を揚げることでオリーブオイルに海老の香りが移り，より海老の風味の強いパスタを作ることができます.

温泉卵

最後のトッピングに使います．入れてソースと混ぜることでより濃厚になります.

ニンニク

ニンニクは包丁の腹で潰して使いましょう．そうすることで，よりニンニクの香りが強くなります．ニンニクは常温のオリーブオイルに浸した状態から弱火でじっくり火を入れて，オリーブオイルにニンニクの香りを移すことをオススメします．また，ニンニクの芽は焦げやすいため，しっかりと取り除きましょう.

玉ねぎ

玉ねぎを入れることでより味に深みが出て，香りが甘くなります．飴色になるまでしっかりと炒めてから入れると良いです．また，新玉ねぎを使うのも良いでしょう．

トマト

トマトを入れたカルボナーラのことをレッドカルボナーラと呼びます．また，フレッシュトマトがない場合でもカットトマトやホールトマトで代用することができます．

3.4　隠し味編

料理に大切な隠し味です．上述している食材の味を引き立てるのに使います．

味噌

チーズも味噌も同じ発酵食品のため相性がよく，コクと旨味が増します．

昆布茶

昆布茶を入れることにより旨味とコクが増します．個人的にオススメです．

愛情

言ってみたかったので入れました．実際，重要だと思います．食べてくれる人のことを思いながら心を込めて作りましょう．

4　作り方

カルボナーラの作り方は十人十色です．インターネット上には「本場では生クリームは使わない」,「卵は全卵を使うべき」などといった意見がよくみられます．しかし，料理のレシピに正解はありません．その料理を食べてくれる人やシチュエーションに合わせて柔軟に作り方を変えるのが真の料理人です．そのため．この章では作り方の概略しか載せません．皆さんも色々な作り方を研究してみてください．

■1. **パスタを茹でる**　麺を茹でるときの水は海水ぐらいのしょっぱさにすることでパスタに味をつけることができます．

■2. **具材を炒める**　ベーコンを炒めるときは，あまりベーコンに触れないようにするのがオススメです．また，玉ねぎを炒めるときは，軽く塩を振ると玉ねぎの中にある余分な水分が抜けるためよりソースが濃くなります．

■3. **ソースを作る**　あらかじめボウルでソースに使う卵やチーズ，黒胡椒を混ぜておきます．

■4. **2 にパスタを投入する**　2 で具材を炒めたフライパンにパスタを投入します．

■5. **4 に 3 で作成したソースを入れる**　4 のフライパンに 3 で作成したソースを入れます．このときの注意点としてソースを入れるときは火を止めてフライパンに適度に茹で汁を入れておかないとすぐに卵が固まってしまいます．ソースを入れた後は適当にフライパンを

煽りよくパスタとソースを絡ませましょう．

■6. **適度に水分を飛ばす**　5 の状態では少々ソースがシャバシャバしています．そこで，弱火でソースにとろみがつくまで水分を飛ばしましょう．ここの過程は難しく，個人的にはカルボナーラの美味しさを決める最も重要な過程だと思っています．熱伝導性が良くフライパンの熱を調整しやすいアルミ製のフライパンを使うと，幾らか簡単にできますが，普通のフライパンでも問題なく作ることができます．

■7. **盛り付け**　中央を高く盛り付けて最後に具材を盛り付けると綺麗になります．

このレシピはあくまでも概略であるため，初心者の方はこれを参考に自分のレシピを作ってみてください．

また，よく巷では「絶対に失敗しないレシピ！」なるものがありますが，失敗なくして成功はありません．成功体験よりも失敗体験からの方が学ぶことは多いです．なので，皆さんは失敗を恐れず様々なレシピに挑戦しましょう．

5　最後に

この記事では，カルボナーラの歴史からカルボナーラによく使われる食材，簡単なレシピを紹介しました．カルボナーラに限らず，料理を真剣に作ることで，自分の料理の良かった点や反省点を見つけることができます．反省点を生かし次の料理に繋げることはとても楽しい行為です．新入生の皆さんがこの記事を足がかりに料理の楽しさに目覚めることを願っています．

動的リンクの依存関係を解析しよう

文　編集部　@coord_e

おはようございます！！！@coord_e です、よろしくどうぞ。本稿『動的リンクの依存関係を解析しよう』は全部 GNU/Linux においての話です[*1]。

1　共有ライブラリと動的リンク

計算機が直接解釈可能な命令列を含んだファイルを**オブジェクトファイル**と呼びます。GNU/Linux ではオブジェクトファイルの形式として主に **ELF**(*Executable and Linkable Format*) [1] が用いられます。ELF 形式のオブジェクトファイルは大まかに以下の通り分類されます。

1. **再配置可能ファイル** (*relocatable file*)。本稿では扱いません。
2. **実行可能ファイル** (*executable file*)。直接実行可能なファイルです。私たちが GNU/Linux で利用するソフトウェアのほとんどはこの形式です。
3. **共有オブジェクトファイル** (*shared object file*)。後述する共有ライブラリのための形式です。

1.1　ライブラリ

いくつかの手続きをひとまとめにして共有・配布できるようにしておくことは、ソフトウェア開発者にとって非常に有用です。例えば、すでに開発された有用な手続きを再利用して使い回すことができ、いわゆる車輪の再発明を避けることができます。このような再利用可能な形でひとまとめにされた手続きの集まりのことを**ライブラリ**と呼びます。

ライブラリ内には名前の付いたデータや手続きが含まれており、ライブラリの利用者はそれらを名前で参照します。ライブラリを利用するプログラム中のそういった参照と、実際のライブラリ内の実装を紐付ける作業を**リンク**と呼びます。リンクを行うことで、初めてライブラリを利用したプログラムは実行できるようになります。

さて、コンパイルを行う言語処理系において、リンクが行われるタイミングによってリンク方式は次の通り二分されます。リンクをコンパイル時に行う**静的リンク** (*static linking*) とリンクを実行時に行う**動的リンク** (*dynamic linking*) です。

静的リンクにおいて、ライブラリはコンパイラによってコンパイル時に（静的に）実行可能ファイルに組み込まれます。この方式は今日では Rust や Go といった言語（の主な処理系）で主に用いられています[*2]。この場合、生成される実行可能ファイルには実行するため

[*1] *BSD や*nix 全体に共通した話も多々あるとは思いますが、そうでない部分もあると思うので……

[*2] これらの処理系でも C ライブラリ（libc や libm など）を動的リンクさせることは往々にしてありますが、ここではそれぞれの言語上のライブラリの扱いについて静的リンクであるという意味合いです。

に必要な全ての手続きが含まれているため、実行時に実行可能ファイルとは別にライブラリを用意する必要がありません[*3]。一方で、静的リンクを行うとコンパイル時間や生成される実行可能ファイルのサイズが増大してしまいます。

他方、動的リンクでは、ライブラリは実行時に（動的に）実行可能ファイルと紐付けられます。すなわち、コンパイルされた実行可能ファイルにはライブラリが提供する手続きの実体は含まれていません。そのため、この方式では実行時に実行可能ファイルとは別にライブラリを用意してやる必要があり、また実行時にリンクによるオーバーヘッドがあります。一方で、コンパイル時間が短く済んだり実行可能ファイルサイズが小さく済む利点があります。この方式は今日では C や C++ といった言語で一般的に用いられています[*4]。

1.2 共有ライブラリ

先に述べた動的リンクを用いてライブラリを配布・利用する際に用いられる、共有オブジェクトファイルのことをここでは**共有ライブラリ**と呼ぶことにします[*5]。

さて、ELF 形式のファイルの大部分は**セクション**と呼ばれる可変長の領域に分割されています。**セクションヘッダテーブル**と呼ばれる固定長の領域が、それぞれのセクションについて名前やファイル中の位置といったメタデータを保持しています。また、ELF ファイルは、セクションとは別の**セグメント**と呼ばれる単位に分割されていることがあります。セグメントはオブジェクトファイルをメモリにロードするときの処理単位であり、1 つ以上のセクションを含んでいます。セグメントが存在する場合、**プログラムヘッダテーブル**と呼ばれる固定長の領域がセグメントの位置や種類といったメタデータの一覧を保持しています。

動的リンクに関係する ELF ファイルには PT_DYNAMIC セグメントが存在し[*6]、PT_DYNAMIC セグメントは.dynamic という名前のセクション[*7]を含んでいます。.dynamic セクションにはタグと値の組が複数埋め込まれており[*8]、タグから値への連想配列として使うようになっています[*9]。このタグを**動的タグ**と呼ぶことにします。動的タグに対応する値は仮想アドレスか数値の形で格納されており、タグによって様々な解釈のされ方をします。

ある ELF ファイルが動的リンク時に必要とする（依存する）ライブラリ名は、動的

[*3] こういった実行可能ファイルを **self-contained** であると言うことがあります。

[*4] C や C++ の上でも静的リンクは可能ですが、文化圏において動的リンクが比較的広く用いられているという意味合いです。

[*5] 本稿中の「〜と呼ぶことにします」の形の定義では、一般的ではない表記や用法で本稿特有の単語を定義しています。

[*6] PT_DYNAMIC はセグメントの種類を表す値に付いている名前で、ELF ファイルに格納される値は 2 です。セグメントには名前は付いておらず、したがって PT_DYNAMIC という名前が ELF ファイル中に埋め込まれているわけではない点に注意してください。

[*7] （セグメントとは違って）セクションには名前が付いており、.dynamic はセクションに付いている名前です。ELF ファイル中にも.dynamic という文字列として埋め込まれています。

[*8] ELF ファイルへの非自明な値の格納操作を総じて「埋め込む」と呼んで誤魔化すことがあります。データを ELF ファイル中でどのように表現して格納するのかについて、詳しくは ELF の仕様 [1] を参照してください。

[*9] 同じタグを持つ組は複数存在してもよく、よって一つのタグに対応する値は複数存在する可能性があります。

タグ DT_NEEDED[*10]に対応する値として複数個.dynamic セクションに埋め込まれています[*11]。

readelf(1) を使って実際に見てみましょう。-d オプションを使うと.dynamic セクションの内容をダンプすることができます[*12]。

```
$ readelf -d /bin/bash
Dynamic section at offset 0xd8450 contains 25 entries:
 Tag                 Type       Name/Value
 0x0000000000000001 (NEEDED)  Shared library: [libreadline.so.8]
 0x0000000000000001 (NEEDED)  Shared library: [libdl.so.2]
 0x0000000000000001 (NEEDED)  Shared library: [libc.so.6]
... (省 略) ...
```

libreadline.so.8 や libdl.so.2 など、依存ライブラリの名前が確かに埋め込まれています。正確には、動的タグ DT_NEEDED に埋め込まれているのは依存するライブラリの名前を表す動的タグ DT_SONAME に埋め込まれている名前（これを今後**ライブラリ名**と呼ぶことにします[*13]）か、またはそのライブラリのパスです。上の例では全て依存ライブラリのライブラリ名が埋め込まれています。

1.3 動的リンク

OS が実行可能ファイルを実行するとき、**プログラムインタプリタ**と呼ばれるプログラムがその実行可能ファイルの実行を肩代わりすることがあります。ELF 形式の実行可能ファイルは利用するプログラムインタプリタへのパスを PT_INTERP セグメント内に保持しているはずです[*14]。そして、プログラムインタプリタはプログラムの実行を行うだけでなく、動的リンカとしての役割も担っています。すなわち、実行可能ファイルの実行が開始されるまでの流れは以下の通りです。

1. OS が対象の実行可能ファイルの PT_INTERP セグメントからプログラムインタプリタのパスを取り出す。
2. プログラムインタプリタを用いてプロセスが作成され、対象の実行可能ファイルの制御がプログラムインタプリタに渡される。
3. プログラムインタプリタが対象の実行可能ファイルの動的リンクを行う。
4. さも直接実行されたかのように、プログラムインタプリタが対象の実行可能ファイルに制御を移す。

[*10] 動的タグ DT_NEEDED は動的タグの値に付いている名前であり、ELF ファイルに格納される値は 2 です。

[*11] 動的タグ DT_STRTAB に対応する位置に存在する文字列テーブルにおけるオフセットが、動的タグ DT_NEEDED に対応する値として格納されています。

[*12] 文字列はすでに動的タグ DT_STRTAB の文字列テーブルから取得された状態で表示されています。

[*13] とはいっても、ライブラリ名はほぼファイル名として扱われています。

[*14] ELF の仕様 [1] の Book Ⅲには “An executable file that participates in dynamic linking shall have one PT_INTERP program header element.” とあるので、保持していないかもしれません。実際、upx というプログラム圧縮ツールで圧縮した実行可能ファイルに PT_INTERP セグメントが無かったのを見たことがあります。

ここで、3 の動的リンクは次の手順で行われます。

1. 対象の実行可能ファイルの動的タグ DT_NEEDED から依存する共有オブジェクトファイルを見つけ出す。
2. 依存する共有オブジェクトファイルをメモリに読み込む。
3. 対象の実行可能ファイルと読み込まれた共有オブジェクトファイルについて、再配置を行う。

2 で依存する共有オブジェクトファイルをメモリに読み込むため、1 では依存する共有オブジェクトファイルのファイルシステム上の場所、すなわち絶対パスが判明している必要があります。しかし 1.2 で見たとおり、動的タグ DT_NEEDED に埋め込まれているのはパスではなくライブラリ名である場合があります。すなわち、依存する共有オブジェクトファイルを全て見つけ出すために、ライブラリ名から絶対パスを決定する必要が生じます。この動的タグ DT_NEEDED に埋め込まれている文字列からパスを決定する作業を共有ライブラリの**名前解決**と呼ぶことにします。

名前解決はこれまでからわかるとおりプログラムインタプリタが、すなわち動的リンカが行います。GNU libc 環境では `ld.so` と呼ばれる実行可能ファイルが動的リンカ・プログラムインタプリタとして主に用いられています。もし動的タグ DT_NEEDED に埋め込まれている文字列（以下、対象文字列）がスラッシュ（/）を含んでいた場合、動的リンカはその文字列をそのままパスとして扱います。この場合は名前解決はここで完了です。対象文字列が/を含まない場合、対象文字列はライブラリ名 s として扱われ、以下の手順で名前解決が行われます。

1. 動的タグ DT_RPATH にコロン（:）区切りのパスのリスト $p_1 : p_2 : \cdots : p_n$ が埋め込まれており、かつ動的タグ DT_RUNPATH が存在しない場合、ld.so(8) は $p_1/s, p_2/s, \ldots, p_n/s$ を順に探します[*15][*16]。
2. 環境変数 `LD_LIBRARY_PATH` にセミコロン（;）またはコロン（:）区切りのパスのリストが格納されている場合、ld.so(7) はそのリストを同様に探します。
3. 動的タグ DT_RUNPATH にコロン（:）区切りのパスのリストが埋め込まれている場合、ld.so(8) はそのリストを同様に探します。
4. 以前見つかった共有オブジェクトファイルのキャッシュである`/etc/ld.so.cache` が存在する場合、`/etc/ld.so.cache` 内のパスのリストから同様に探します[*17]。
5. デフォルトのパス（64 ビット環境では`/lib64` と`/usr/lib64`）から同様に探します[*18]。

[*15] 動的タグ DT_RPATH は見つかった共有オブジェクトファイルの依存ライブラリの名前解決にも再帰的に用いられます。

[*16] s はライブラリ名ですが、ここではファイル名として扱われています。気持ち悪いですね。

[*17] コンパイル時に`-z nodeflib` をつけてリンクされている場合は、5 のデフォルトパス以下のものはスキップする。

[*18] コンパイル時に`-z nodeflib` をつけてリンクされている場合を除く。

ELF の仕様 [1] では 1、2、そして 5 のみ規定されており、それ以外は GNU libc 環境特有のものだと思われます。ld.so(8) の manpage によると動的タグ DT_RPATH は deprecated で、動的タグ DT_RUNPATH を代わりに使用すべきだとされているようです。このように、正直なところ共有ライブラリの名前解決は「実装が仕様」状態になっているようです。今後は特に GNU libc 2.31 の環境に限定して話を進めていきます。

2 動機

ここからは、実行可能ファイルを解析して依存する共有オブジェクトファイルを列挙する２つのアプローチについて紹介することで、動的リンク時の名前解決の流れを確かめます。また、この試みには以下の２つの実用的な動機があります。

1. 実際に利用する前にプログラムが依存する共有オブジェクトファイルを把握しておくことはそのプログラムの実行直前の挙動のよい予測に繋がり、実行時リンクエラーなどのトラブルの解決を容易にします。
2. プログラムが実行時に必要とするファイルを列挙することができれば、小さく安全な隔離環境でプログラムを実行することができます。今回の手法を応用して、動的リンクされた実行可能ファイルから小さな Docker イメージを作成することができます。[*19]

はじめに、これまでの知識を応用して静的解析を行う方法を紹介します。その後、より低い視点から攻めた動的解析のアプローチを紹介します。

3 静的解析

解析対象のプログラムを実行せずに解析を行うとき、その解析を**静的解析**と呼ぶことにします。1.2 に書いたとおり、必要とする共有ライブラリ名の情報は ELF ファイルに含まれています。ELF の解析によって得られる共有ライブラリ名と、1.3 に書いた動的リンクの仕組みを使って、実行可能ファイルが依存する共有オブジェクトファイルを列挙することができます。

実は、この解析はほとんど[*20]ldd(1) によって達成されます。やってみましょう:[*21]

```
$ ldd /bin/bash
  linux-vdso.so.1
  libreadline.so.8 => /usr/lib/libreadline.so.8
  libdl.so.2 => /usr/lib/libdl.so.2
  libc.so.6 => /usr/lib/libc.so.6
  libncursesw.so.6 => /usr/lib/libncursesw.so.6
  /lib/ld-linux-x86-64.so.2 => /usr/lib/ld-linux-x86-64.so.2
```

[*19] https://github.com/coord-e/magicpak

[*20] ldd のバージョンによっては PT_INTERP セグメントを考慮せずにデフォルトのプログラムインタプリタを用いて解析を行うものもあり、その場合不正確な解析結果が表示されることもあります。

[*21] アドレス表示は省略しています。結果は環境によって異なります。

この右側に表示されている/usr/lib/libreadline.so.8 などが、依存する共有オブジェクトファイルのパスです。今回は実行可能ファイルからこれを全て取得するのが目的です。もちろん ldd(1) の結果をパースすることもできますが、いわばこの出力はログのようなものです。ldd(1) の manpage に出力の例は載っていますが、正確な文法は記載されていません。今回はそういった不安定なものに頼らず、documented な方法のみを用いて解析を試みます。

3.1 dlopen を用いた解析

まずはじめに思いつくのが、解析対象の実行可能ファイルの動的タグ DT_RPATH や動的タグ DT_RUNPATH を解析したのち 1.3 に書いた通りの探索を行う方法です。もし名前解決の手順がしっかり規格化されていればこれでいいのですが、先程書いたとおり現在用いられているシステムでは「実装が仕様」状態です。特に、デフォルトのパスや使うキャッシュファイルが未知の場所にある場合があります[*22]。そのため、なるべく正確な結果を得るためには対象の実行可能ファイルが使う動的リンカに直接問い合わせるしかないようです。

さて、1.3 で ld.so(8) が実行前に動的リンクを行っていると書きました。残念ながら ld.so(8) はライブラリとしてのインタフェースを提供していません。ユーザーが動的リンクをライブラリとして扱えるインターフェースに、libdl があります。これは GNU libc などに[*23]同梱されている、動的リンクを扱うライブラリです。ここでは libdl が提供する API のうち、dlopen() と dlinfo() を用いて解析を行います。

- dlopen(3). ライブラリ名かパスを受け取り、名前解決したのち共有オブジェクトファイルを読み込む。他の API に渡すハンドルを返す。
- dlinfo(3). dlopen(3) が返すハンドルを受け取り、ロードされた共有オブジェクトファイルについての情報を取得する。非標準の GNU 拡張です。

dlopen(3) は ld.so(8) を通じてライブラリ名の名前解決を行います。そのため、dlopen(3) に名前解決をさせ、dlinfo(3) で解決後の共有オブジェクトファイルのパスを取得すれば共有ライブラリの名前解決を documented なライブラリ関数を用いて行うことができます。

しかし、dlopen(3) が**どの** ld.so(8) を使用するかはライブラリの範囲では制御できません。内部的には dlopen(3) は ld.so(8) の（undocumented な）シンボルを使用して名前解決を行っているので、dlopen(3) を使う実行可能ファイルの動的リンク時に目的の ld.so(8) を使わせれば良さそうです。この方法で実行可能ファイルが直接依存する共有オブジェクトファイルのパスを列挙するアルゴリズムを次に示します。

1. 解析対象の実行可能ファイルの PT_INTERP セグメントから、使用するプログラム

[*22] nix というエコシステムでインストールした実行可能ファイルが、同エコシステム内のライブラリを使うようにビルドされた動的リンカを使うようになっていた例があります。

[*23] dlopen() は POSIX.1-2001 にあるらしいです。

インタプリタのパスを取り出す。

2. 解析対象の実行可能ファイルの動的タグ DT_RPATH から、パスの列を取り出す。
3. 解析対象の実行可能ファイルの動的タグ DT_RUNPATH から、パスの列を取り出す。
4. 解析対象の実行可能ファイルの動的タグ DT_NEEDED から、依存するライブラリ名の一覧を取り出す。
5. 次のような C 言語のソースコードを用意する。すなわち、
   - コマンドライン引数 `argv[1]` を引数に dlopen(3) を呼び、ハンドラを得る。
   - dlinfo(3) に得たハンドラを渡し、ロードされた共有オブジェクトファイルのパスを得る。
   - 得たパスを標準出力に書き出して終了する。
6. 5 で用意したソースファイルを、次のようにコンパイルする。すなわち、
   - 1 で取り出したプログラムインタプリタを PT_INTERP セグメントに埋め込む。
   - 2 で取り出したパスの列を動的タグ DT_RPATH として埋め込む。
   - 3 で取り出したパスの列を動的タグ DT_RUNPATH として埋め込む。
7. 4 で取り出した依存ライブラリ名それぞれを引数として 6 で生成された実行可能ファイルを実行すると、求める共有オブジェクトファイルのパスが標準出力に得られる。

3.2 問題点と解決策

しかし、これでは正しく解析が行えない場合があります。なぜなら、アルゴリズムの 6 で得られる実行可能ファイルも動的リンクされているからです。特に、`libc.so` との動的リンクが問題になります。ld.so(8) は `libc.so` とともに GNU libc の一部として配布されており、一緒に提供されている libc 以外とリンクするとうまく動かないことがあります[*24]。

ある ld.so(8) があったとき、システムから互換性のある `libc.so` を見つけ出すのは容易ではありません。そこで、ここで妥協案として下のような仮定をします。

仮定

ある実行可能ファイルの動的リンカとして使用される ld.so(8) は、その実行可能ファイルに動的タグ DT_RPATH や動的タグ DT_RUNPATH がなく、かつ環境変数 `LD_LIBRARY_PATH` が設定されていないとき、`libc.so` を自身と互換性のある libc の共有オブジェクトファイルに名前解決することができる。

これは比較的妥当な仮定に思えます。ld.so(8) は、特別の調整なしに自身と互換性のある libc を見つけることができるはずです。この仮定が崩れるのは、システム上の `libc.so` が軽率に移動された時ぐらいでしょう。

この仮定のもと、もう一度名前解決の方法を考えます。仮定から、名前解決を ld.so(8) に

[*24] main に入る前に SEGV 吐いて落ちたりします。データ構造に互換性がなかったりするんでしょうか。よく調べていませんが……

頼れるのは動的タグ DT_RPATH や動的タグ DT_RUNPATH、環境変数 LD_LIBRARY_PATH の影響を排除した状態のみになりました。そのため、それら 3 つを使った検索は自前で行います[*25]。その後、それら 3 つの影響を受けない状態で ld.so(8) に検索させます。このようにして先程のアルゴリズムの問題点を解消したアルゴリズムを次に示します。

1. 解析対象の実行可能ファイルの PT_INTERP セグメントから、使用するプログラムインタプリタのパスを取り出す。
2. 解析対象の実行可能ファイルの動的タグ DT_RPATH から、パスの列を取り出す。
3. 解析対象の実行可能ファイルの動的タグ DT_RUNPATH から、パスの列を取り出す。
4. 解析対象の実行可能ファイルの動的タグ DT_NEEDED から、依存するライブラリ名の一覧を取り出す。
5. 3 で動的タグ DT_RUNPATH が存在しなかった場合。4 で取り出した依存ライブラリ名それぞれについて、2 で取り出したパスの列から検索する。検索が成功した場合、終了する。
6. 4 で取り出した依存ライブラリ名それぞれについて、環境変数 LD_LIBRARY_PATH に格納されているパスの列から検索する。検索が成功した場合、終了する。
7. 4 で取り出した依存ライブラリ名それぞれについて、3 で取り出したパスの列から検索する。検索が成功した場合、終了する。
8. 次のような C 言語のソースコードを用意する。すなわち、
   - コマンドライン引数 `argv[1]` を引数に dlopen(3) を呼び、ハンドラを得る。
   - dlinfo(3) に得たハンドラを渡し、ロードされた共有オブジェクトファイルのパスを得る。
   - 得たパスを標準出力に書き出して終了する。
9. 8 で用意したソースファイルを、次のようにコンパイルする。すなわち、
   - 1 で取り出したプログラムインタプリタのパスを PT_INTERP セグメントに埋め込む。
10. 4 で取り出した依存ライブラリ名それぞれを引数として 9 で生成された実行可能ファイルを実行すると、求める共有オブジェクトファイルのパスが標準出力に得られる。ただし、これは環境変数 LD_LIBRARY_PATH をクリアした状態で実行する。

お疲れ様でした。これで実行可能ファイルの直接の依存する共有オブジェクトファイルを求めることができました。あとは求めた共有オブジェクトファイルそれぞれについて再帰的に同様の処理を行っていくことで、ldd(1) と同じように依存する共有オブジェクトファイルをすべて列挙することができます。

[*25] もちろんそれら 3 つを用いた検索も本来 ld.so(8) が行うはずだったので「実装が仕様」な現状では最善だとは言えませんが、私がこれまで出会った ld.so(8) は皆これら 3 つに対する検索に関しては同等に振る舞っていた印象を受けています。

4　動的解析の紹介

さて、静的解析とは対照的に、解析対象のプログラムを実際に実行しながら解析を行うとき、その解析を**動的解析**と呼ぶことにします。

例えば dlopen(3) でロードされる共有ライブラリは動的タグ DT_NEEDED にはリストされず[*26]、そういった共有ライブラリは 3 で解説した方法では見つけることはできません。ここでは、動的解析によってそういった実行時に決定する依存ライブラリをもリストアップする方法を紹介します。

ここでは「実際に起きたことを記録する」方法を取ります。すなわち、実際に対象となる実行可能ファイルを実行してみて、何らかの方法で動的リンクの様子を観察し、ロードされたファイルパスを記録していきます。動的リンクによってロードされたファイルのパスを外から観測する方法は、対象となる実行可能ファイルに手を加えないものに限ると主に以下の２つがあります。

1. rtld-audit(7) を使う方法。rtld-audit(7) は、ハンドラとなる関数を実装した共有オブジェクトファイルを用意し、実行時に環境変数 `LD_AUDIT` 環境変数にそのパスを指定した状態で対象となるプログラムを実行すると、動的リンクに関連するイベントに応じてハンドラが呼ばれるといったものです。GNU libc の動的リンカ特有のインターフェースです。
2. ptrace(2) を使う方法。ptrace(2) は、プロセスと OS とのやり取りを観測することができるものです。比較的多くの OS 上で利用可能です。

今回は後者を使うことにしました[*27]。

4.1　ptrace を使った解析

プロセスは**システムコール**を使って OS とやり取りすることができます。ptrace(2) は、他のプロセスの実行を観測・制御（=**トレース**）することのできるシステムコールです。ptrace(2) を使うと、たとえば他のプロセスのメモリを書き換えたり、レジスタを読んだり、ステップ実行をさせたりできます。このようにデバッガの実装にも使われる ptrace(2) ですが、その機能の 1 つにトレースしているプロセスのシステムコール呼び出しの前後で停止する、というものがあります。これを用いると、システムコール呼び出し前の停止時にシステムコールへの引数（レジスタに入っています）を読み出すことでどのようにしてシステムコールが呼ばれているか観察することができます。

もちろん動的リンカが共有オブジェクトファイルをロードする際にもシステムコールを呼び出します。そのため、解析対象の実行可能ファイルを ptrace(2) でトレースし、システムコール呼び出し前に停止させた時の引数の値を記録していくことで、結果として依存す

[*26] dlopen(3) でロードされるライブラリの名前は実行時に決定するため、コンパイラはどう頑張っても動的タグ DT_NEEDED に dlopen(3) でロードされるライブラリを常にリストアップすることはできません。

[*27] 筆者が自作しているツールに ptrace(2) のほうが都合が良かったとか単純にシステムコールトレーサに興味があったみたいな理由があります。rtld-audit(7) は各自使ってみてください。

る共有オブジェクトファイルを一覧することができます。

さて、ptrace(2) の便利なインターフェースとして strace(1) があります。これは引数のコマンドを実行し、そのプロセスのシステムコール呼び出しとその引数、戻り値を標準出力に出力してくれるツールです。試してみましょう。

```
$ strace bash -c true 2>&1 | grep open
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libreadline.so.8", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/libncursesw.so.6", O_RDONLY|O_CLOEXEC) = 3
... (省略) ...
```

確かに 3 で見たような共有オブジェクトファイルのパスが見えています。ここでは、パスはそれぞれ openat(3) システムコールの引数として現れています。もちろん、strace(1) に頼ることなく、ptrace(2) を直接使ってこれらパスを列挙することもできます。これは ptrace(2) の単純な応用であり、ここでは詳細は省略します[*28]。もっとも、参考文献 [2] などを参考にすれば簡単に実装することができるでしょう。参考までに、解析の手順の一例を次に示してこの節を終えます。

1. fork(2) する。子プロセスで行う処理は次の通りである。
   (a) ptrace(2) で PTRACE_TRACEME リクエストをする。
   (b) execve(2) で解析対象の実行可能ファイルに処理を移す。
2. 親プロセスは、子プロセスが SIGTRAP で停止するまで waitpid(2) で待つ。[*29]
3. 親プロセスは、ptrace(2) で PTRACE_SETOPTIONS リクエストを送る。特に、PTRACE_O_TRACESYSGOOD を設定する。
4. 親プロセスは、ptrace(2) で PTRACE_SYSCALL リクエストを送り、子プロセスの実行を再開させる。PTRACE_SYSCALL を送ることで、子プロセスは次のシステムコールの呼び出し前に停止するようになる。
5. 親プロセスは、子プロセスがシグナル `SIGTRAP | 0x80` で停止するのを待つ。この値は PTRACE_O_TRACESYSGOOD によるものであり、SIGTRAP とシステムコール呼び出しによる呼び出しを区別するのに役立っている。
6. 親プロセスは、ptrace(2) で PTRACE_GETREGS リクエストを送り、子プロセスのレジスタを読み出す。
7. 親プロセスで、読み出されたレジスタから呼び出されたシステムコールを判別する。open(2) または openat(2) ではない場合、実行を再開する。(4 に戻る)
8. 親プロセスで、ptrace(2) で PTRACE_PEEKDATA リクエストを送り[*30]、パスのレジ

[*28] 執筆時間の都合などがあった。スマン。
[*29] トレースされているプロセスが execve(2) を呼ぶと基本的には SIGSTOP が飛んでくるようになっています。
[*30] PTRACE_PEEKDATA は 1 ワードごとしか読みだせないので、実際にはヌル終端をみつけるまで複数回呼び

スタ[*31]に入っているアドレスから文字列の実体を読み出す。それが.so を含む場合はおそらく求める共有オブジェクトファイルのパスなので、記録する。

9. 4 に戻る。これをプロセスが終了するまで続ける。

5 結論

本稿では、GNU/Linux における共有ライブラリ・動的リンクについて説明しました。また、実行可能ファイルの依存する共有オブジェクトファイルを静的解析と動的解析によってそれぞれ列挙する方法について前節で説明した知識を応用しつつ解説しました。

6 参考文献

[1] TIS Committee et al. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. 1995.

[2] 大山恵弘. *ptrace* **入門***: ptrace* **の使い方**.

出す必要があります。

[*31] open(2) だったら rdi、openat(2) だったら rsi。

「どの IME を使うか決めましたか？」「ATOK に決めました」

文　編集部　ninojujo

1　「語彙力をください」

1.1　「現状」

究極、紙とペンさえあれば小説家は本を書くことができる。そんなことは言われるまでもなく、そもそも数十年前まではそれ以外に小説を書く術はなかった。

しかし時代は変わる。世はパソコンで執筆活動を行う時代。スタバでアップルのロゴを見せびらかしながらキーボードをカタカタいわせるのが主流であり、いまどき手書きでちまちまと書いては消してを繰り返しているのは、一部の絶滅危惧種か思春期特有の病気をこじらせた少年少女たちくらいだろう。後々その黒いノートが見つかって泣きを見るだなんて考えもしていないのである。無論私も予想だにしていなかった。

ともあれ、現代においてパソコンが執筆という行為に果たす役割は到底無視できるものではない。テキストエディタにはじまり、漢字変換、インターネットでの情報収集、あるいは類義語検索、さらには作品の公開・閲覧に至るまで、小説というコンテンツにおけるありとあらゆる局面においてパソコンという存在が何らかの役目を果たしている。もしこの世からパソコンが忽然と消えたなら、多くの小説家はその職を失うことになるだろう。それほどまでに今の小説家たちはパソコンというツールに依存している。

かく言う私もそうだ。

物書き仲間の先輩からパソコンでの執筆方法を教えてもらう以前は私も絶滅危惧種の一員であったが、最近は専らカタカタカッターンの日々である。毎日スタバでドヤっている。そのこだわりたるや、2 万そこらもする『えちえちけーびー（HHKB）』なるキーボードを買ってしまうほどなのだから、我ながら相当なものだと言えよう。ちなみにこのえちえちキーボード、とても気持ちがいい。指が。

1.2　「願望」

えちえちの話はさておき、パソコンで執筆する悦びを知ってしまった私は、その依存度をますます深めつつあった。キーをスコスコ打つかたわら、どこか不満を感じていたのである。

ずばり、類義語検索についてだ。

文豪と呼ばれる偉人や経験豊富なベテラン小説家たちに比べて、所詮しがない作家志望でしかない私は明らかに知識も教養も不足している。そのため執筆中の類義語検索は必要

不可欠。語彙力の欠落を補うためには、森羅万象を網羅する Google 先生にお聞きするほかないのだ。しかしパソコンという効率化の権化みたな道具にすっかり牙を抜かれてしまった私には、都度検索フォームに『○○ 類義語』と入力することすら億劫に思えて仕方がないのである。

この屈託を単なる怠惰と侮るなかれ。モチベーションの低下は執筆における最大の敵と言って過言でないのだ。類義語を検索してはしっくりくる表現を探し、思うような表現が見つからないまま一日が過ぎ、一週間が過ぎ、一ヶ月が過ぎ……のちの名著が気づけば始末に負えないフォルダの肥やし。ストレージを喰うばかりの穀潰しになり下がるのである。このように類義語検索というプロセスによって執筆意欲が削がれ、志を挫かれた作家の卵は数知れない。看過しがたい難題なのだ。

そんな悩みを例のパソコンで執筆する術を教えてくれた先輩に打ち明けたところ、「書きたきゃ読め」という簡潔にして至極当然な返答が送られてきた。全くもってその通り、反論のしようがないほど的を射たありがたいお言葉ではあるのだが、私が聞きたいのはそんな順当な正論ではない。

もっと安易で、お金も時間もかからなくて、指先をちょちょいと動かすだけで類義語がぱーっと降ってくるような、そんな小手先ツールが知りたいのである。

2 「ATOK を使いなさい」

2.1 「先輩」

「あなたにパソコンなんて教えなければよかった」

私の願望を聞いた先輩の第一声がこれである。

例のごとくスタバでドヤっていた先輩を捕まえて、改めて私の悩みを解決する手段はないかと尋ねたところ、返ってきたのは未だかつて見たこともないほどの呆れ顔だった。

「原稿用紙に万年筆で書いてた頃のあなたの方がよっぽど真面目に小説に向き合ってたわ」

そう言って先輩はコーヒーを一息に飲み干すと、特大のため息を吐いた。どうやらこのまま帰る気でいるらしい。逃してなるものかと、私は立ち上がりかけた先輩の腕をひっつかんで一気にたたみかけた。

「しかしそうは言っても先輩、私をこんな体にしたのは先輩なわけでして」

「語弊のある言い方しないで」

「先輩にはその責任をとる義務があります」

「天地神明に誓ってないわ」

「どうか私に寝てても降ってくる語彙力をお恵みくださいませんか」

「本当どうしてこんな子に……」

「お願いです、何でもしますから」

「じゃあその手を離して」

「それ以外で」

「この小娘」

気づけば店中から私たちに視線が集まっていた。それに気づいた先輩が慌てて——嫌々ながらも——腰を下ろしたので、とりあえずぎゅうぎゅうに締め付けていた先輩の腕を解放する。

腕をさすりながら忌々しそうにこちらを睨み付ける先輩に屈することなく、無言でコーヒーを啜りながら鋭い視線攻撃をただ受け流す。するとついにまともに相手にすることすら無駄だと悟ったのか、眉間にしわを寄せながら、観念したように声を絞り出した。

「……わかった、わかったわ。教えればいいんでしょ教えれば」

憮然とした表情を崩さないまま、先輩は紙コップをぐしゃりと握り潰した。多分先輩の頭の中で私の頭蓋に変換されているのだろう。直接ドタマにガッとこないあたりに後輩への思いやりを感じる。光栄の至りである。

「わーいありがとうございます先輩。いつも頼りにしてます」「信頼が憎い」

こうして友情を深め合った私たちは、口直しにともう一杯アメリカンコーヒーを注文し、改めて席に着いた。

2.2 「巡合」

「それでどうすれば楽して語彙力を手に入れられるんですか」

「ちょっとは取り繕いなさい」

期待に満ちた私の問いかけに苦言を呈しながら、先輩はとあるウェブページ (`https://atok.com/`) を開いて見せた。

「なんですかこれ」

「ATOK」

「なんですかそれ」

「IME」

吐く息が惜しいのだろうか。必要最低限の言葉しか返してくれないのはいつものことだが、今日はいつにもまして簡潔である。これで仕事は終わったと言わんばかりにくつろぎはじめた先輩に無言の圧力と視線で訴え続けていると、ついに耐えきれなくなったのか、エスプレッソでも飲んだかのような顔を浮かべて再び口を開いた。

「IME は知ってるでしょ」

「えーっと……」

「知らないのね」

「いやいやまさかそんな」

知らないわけではない。ただちょっと英字の略語に弱いだけである。突然あいえむ——IME などと言われても咄嗟に出てこないのだ。おのれ IT 業界に蔓延るアブリビエーションめ。IT って何だ——いや、そんなことはどうでもいい。いまは IME の話だ。

何だったか……日本語を入力するのに一役買っているソフトウェア、というような説明をどこかで読んだおぼえがある。もともとキーボードはアルファベットを入力することを想定して作られたものであり、日本語の入力はできないはずだった。それを可能にしてい

るのが IME、みたいな。

「詳しいことはともかく、その IME っていうのでローマ字入力をひらがなに変換したり、ひらがなを漢字とかに変換しているんですよね」

「ざっくり言えばそんな感じ。で、ATOK もその IME で、より性能が高いの」

「はあ……」

ひとりで納得して勝手に話を進めてしまうのは先輩の悪い癖である。首を傾げ、順を追って説明することを再度要求。先輩は仕方なさそうに首を振った。

「一口に IME といっても種類は様々よ。そもそも日本語以外の言語を入力するための IME だってあるし、日本語 IME だけでも多種多様。有名なのだと Microsoft IME とか、Google 日本語入力とか。聞いたことくらいあるでしょ」

「Microsoft は知ってます。なにせ私のパソコンも Windo ——いえ、やっぱ知りません」

「は？」

「私、Mac ユーザーなので」

言って、ここぞとばかりに自前のパソコンの背面を誇示する。顕現する白リンゴ。目を眇める先輩。次の瞬間、目にも留まらぬ速さで伸ばされた手がステッカーをべりっと引っぺがした。

「ああっ」

「騙るな」

そんな殺生な。それが無いままで、私はいったいどんな顔してスタバに居座れば良いというのだ。

「それともまさか先輩は私にファミレスで、しょんぼり顔で、どこの馬の骨かもわからないパソコンをぺちぺちしていろとでも言うんですか」

「その器の小ささでこの大口の叩きよう」

「Microsoft なんて所詮マイクロですよ。140 センチ超えの私には勝てません」

「小っちゃいなぁ。よちよち」

「馬鹿にしてます？」

「限りなく」

「ふんっ！」

首をぶん回して頭に置かれた先輩の手を振り払った。まったく、油断も隙もない。手癖の悪い先輩を睨むように警戒し、リュックサックの中から予備のステッカーを取り出して慎重に貼り直した。無論リンゴのロゴである。先輩の口の端から呆れ笑いが漏れ出た。

「あなたのパソコンがアップルライクだろうが何だろうが、とにかくあなたは Windows ユーザーで、当然 Microsoft IME も知ってるってことね」

「……まあそうですけど。いつもお世話になってます」

「最初からそう言えば良いのよ」

やれやれ、なんて言いながらコーヒーを一口。軽く口を湿らせると、先輩は再度画面を指さした。

「さっきも言ったように、IME にはいろんな種類があるわ。そして ATOK もそのひとつなの」

なるほど、それは理解できた。けれど私が知りたいのはその先である。その ATOK という IME がいかようにして豊かなる語彙をもたらしてくれるか、それが肝心なのだ。

「ほら、これ見て」

表示されたウェブページの一部に目を凝らすと、少し聞き慣れない単語が目に入った。

「連想変換？　普通の変換と何か違うんですか」

「そうね……簡単に言えば、普通の変換が同音異義語を変換候補に並べるのに対して、連想変換は類義語を変換候補に並べるのよ」

「おぉ……」

わかるようなわからないような。何だか凄そうなことだけはわかるけれど。言葉を噛み砕くのに少々難儀していると、先輩がおもむろにパソコンを閉じた。

「あれ？」

まだ見ていたんですけど。非難の意を込めて先輩に目を遣ると、いそいそと鞄にパソコンをしまっていた。

「え、まさか帰るんですか。それもこのタイミングで」

「お腹すいたもの」

まったくの寝耳に水。気持ちよくドライブしていたら急ブレーキがかかって放り出されたような気分である。完全に置いてきぼりを喰らっている間に先輩は手際よく荷物をまとめ、私に背を向けた。

「ちょ、ちょっと待ってください。私はいったいどうすれば――」

「実際やってみた方が早いわ。無料体験版があるからそれで試してみなさい。それじゃ」

ようやく我に返ってなんとか引き留めようと声をかける。すると先輩はくるりと振り返り、振り向きざまに紙コップに入ったコーヒーを全て飲み干してゴミ箱に捨て、ついでにそんな助言も吐き捨てて去って行った。

あとに残ったのは、私と私の Mac もどきと飲みかけのコーヒー。カップをのぞき込むとまだ二口分くらいしか減っていなかった。恐る恐る口をつけると、到底ゴクゴクとは飲み干せないほど熱い。チビチビと飲むにも熱すぎる。あの人の舌は鋼か何かなのだろうかと益体もないことを思った。

3　「使います」

3.1　「連想変換」

その後私は家に戻ると、去り際に放たれた助言通り体験版を試してみることにした。

したのだが、間違えて有料の製品版を買ってしまった。月額 500 円というなかなか馬鹿にならない金額である。まあ本当に語彙力が手に入るのなら安い買い物だ。毎月文庫本一冊買って地道に語彙力をつけるよりは遥かに効率が良い。そう自分に言い訳しながら、Windows 用の ATOK をダウンロードする。起動しようとすると、なぜか解凍に失敗したと

いう旨のメッセージが出たので原因を究明。どうやらブラウザ以外でダウンロードしたのが問題だったらしい。ブラウザで直接ダウンロードすると今度はうまくいった。

「さてさて、どんな感じかな」

さっそくテキストエディタを起動して連想変換を試してみる。何か適当な単語をと思ったけれど、こういうときに限ってパッと良い例が思い浮かばない。別に何でも良いだろうに。しばらく考えて、極めてどうでもいい『わたし』、つまり一人称のバリエーションを検証してみることにした。

実験する単語が決まったところで、いざお手並み拝見。と行きたかったが、またしても問題発生。どうやれば連想変換できるのかわからない。スペースキーだと普通の変換だし、タブキーは推測変換――『わたし』の推測変換でいきなり『私に天使が舞い降りた！』が提案されるあたり、さてはこの IME やりおるな、なんて思いながら格闘すること十数分。半ば意地になって自力で解決するまでは絶対ググらない、などという無駄な決意を固めていたけれど、最終的になんかどうでもよくなってきて ATOK のホームページを参照し、あっさり解決した。コントロールとタブの同時押しだった。

コントロールタブって押しづらくね、ていうか実際押しづらい、だから私もさっき試そうとも思わなかったんだし、という文句を垂れ流す内心をいったん抑えて実験再開。『わたし』と入力してコントロールタブを押すと、その候補が出るわ出るわ。

「僕、おれ、わし、小生、余、拙者、我が輩、某、わちき……」

他にもまだまだある。全体的に古めかしい表現ばかりな気もするがそんなことはどうでも良い。というか現代の一人称が少なすぎるだけだ。サラリーマンも『私』ばっかり言ってないで『わちき』とか使えば良いのに。来たれ廓詞ブーム。

一人称のバリエーションを取りそろえていることはわかったものの、さすがにこれだけでは実際に使えるかどうかわからない。他にも何か試してみるべき単語はないかと考えて、普段よく検索している『言う』の関連語句を調べてみようと思い至る。

連想変換してみると、先ほどの比じゃないくらい候補が表示された。

「申す、述べる、ぼやく、口走る、ほざく、のたまう、毒づく、皮肉る、そしる……」

後半なぜかネガティブな単語ばかりピックアップしてしまったが、『言う』の類義語だけとってもやはりその語彙数は豊富であると言えよう。

もっとも、やはりインターネットで検索したときに得られるものと比べれば数は劣る。それでも語彙の絶対数は十分すぎるほどだし、何より当初から私が望んでいたとおり、いちいちブラウザを開いて『○○　類義語』と検索するよりはずっと手軽だ。何せコントロールタブを押せばいいだけなのだから。コントロールタブじゃなければもっと良かったけれど、あとで調べたら設定をいじればなんとかなるみたいだし。それに検索しなくていいということは、そういった欠点を補って余りあるほどの美点なのだ。

もしこれで小説を書いたらどうなるんだろう――想像しただけでも心躍るが、お楽しみは後に取っておく。ショートケーキのイチゴは最後まで取っておいて姉にかっさらわれるのがお決まりだったが、小説は逃げないので取っておく。他にも何か機能があるらしいの

で、まずはその確認が先である。

3.2 「変換、推測変換」

ATOK のウェブサイトをあれこれ探りながら、どんな機能があるのかをひととおり把握。とりあえず片っ端から試してみるかと袖をまくった。

まず気になったのは、通常の変換や推測変換という、一般的な IME にも搭載されている機能の実力である。連想変換は確かに凄いけれど、それだけ凄くても基本的な性能が悪ければ片手落ちというか、本末転倒だ。私が普段使っている Microsoft IME、それと先輩が愛用しているという Google 日本語入力を急遽インストールして、ATOK と比較してみる。

まずは通常の変換機能についてだ。日本語で最も同音異義語が多いとされているのは『こうしょう』という言葉で、広辞苑に登録されているのは全部で 48 個あるらしい。そこでこの単語をそれぞれの IME で変換し、表示される数を調べてみる。Microsoft IME で変換してみると、その候補は 100 個弱。先輩が愛用しているという Google 日本語入力もやはり 100 弱。二つとも同じくらいということは、そもそもそれくらいしかないのかなー、なんて油断していたら、ATOK はその 3 倍を超える 330 を叩きだしてきた。むしろその 230 個をどこから持ってきたんだと問い質したくなるほどに多い。

他の単語でも試してみたが、評価は軒並み同じ。どれも ATOK が他の二つより多くの候補を呈示する結果となった。

推測変換についても同様の結果が得られた。先ほどのことも思いだしたので、『わたし』と入力して推測変換した結果の比較を試みる。40 個も表示されない Microsoft IME はさておき、100 個ほどの候補を吐き出してみせた Google 日本語入力と ATOK を比べてみても、やはり ATOK のほうが実用的で、かつ日本人向けであると感じた。

いままでに入力した『わたし』で始まる文章もきちんと候補に出てくるし、基本的には実用的な変換候補が上位に並ぶ。何より『わたし』と入力するだけで『私に天使が舞い降りた！』を提案してくれる IME なんて ATOK をおいて他にない。別に必要かと言われればまったく必要じゃないけれど、個人的にちょっとうれしい。まさかと思い冗談半分で『まち』を推測変換してみたところ、期待を裏切らず『まちカドまぞく』が変換候補にあった。なんて勤勉なんだと純粋に感心する。日本のサブカルに強いと噂の Google 日本語入力ならもしかしたらと内心期待していたのだが、『わたしとよりそうスムージー』や『マチャミ』といった風に、何だかよくわからないキャッチコピーであったり、さして興味のない芸能人の名前などが表示された。どうも Google 日本語入力は闇鍋感が強い。思いつく限りの言葉を詰め込んで適当に並べたって感じがする。ということで、推測変換においてもやはり ATOK に軍配が上がった。

さらに比較する過程で交互に使っていて感じたのだが、ATOK は変換精度がかなり高い。無論、収録語彙数が多いというのもひとつの理由だろう。しかしそれだけではなく、あくまでなんとなくの個人的な所感ではあるのだが、日常的によく使うものが変換候補の上位にきちんと配置されているのだ。それは他のふたつ同様に搭載されている学習機能によるものだけではなくて、デフォルトでそのように配置されているのである。だから ATOK を

使い始めて間もない私でも、指の滑りが非常にいい。スペースキーやタブキーの上に指が長々と留まっているということが殆どないのだ。

それとこれはもっと個人的な感想になってしまうのだけれど、UI がいい。なんか見やすい。小さいようで、非常に重要なファクターである。調べてみると、どうやら Windows の設定でダークモードにしているとそれが ATOK の UI にも反映されて、黒い背景に白抜き文字で表示されるようになるらしい。見やすいと感じる理由はそれだけではなくて、表示される文字も不思議としっくりくる。なんでかなー、としばらく眺め比べていたら気づいた。どうやら今この瞬間入力しているフォントに合わせて、変換候補の文字のフォントも変化しているのだ。明朝体なら明朝体に、ゴシック体ならゴシック体になっているのである。これがいいかどうかは人によるだろうが、少なくとも私は断然好きだと感じた。

どうやら基本的な機能においても、他と比べてなんら遜色はないようだ。そこで次はいよいよ他の IME には「ない」機能を試してみることにした。

3.3 「逆引き変換、翻訳変換、クラウド辞典」

ATOK の強力な変換能力に拍車をかけるのが、逆引き変換機能である。簡単に説明すると、普通の推測変換が語頭の文字列から推測するのに対して、これは語末の文字列から推測するのである。もし仮に私が「あー、アレなんだっけなー。『なんとかストラ』みたいな感じだったと思うんだけどなー」という状況に陥ったときに、『なんとかストラ』と入力してタブキーを押すだけで、『リストラ』も『ツァラツストラ』も『Fate/EXTRA』も思い出させてくれるのだ。手厚すぎて涙がちょちょぎれそうである。

うーむ、便利だし凄いし不満なんて微塵もないけれど、これに甘えてたらなんか将来的にとんでもなくボケそう。今ですら「お母さんアレちょうだい。ほら、あのー、飲むやつ」「はいはいお茶ね」というやりとりが頻繁に交わされているというのに……私はいったいどこまで堕ちれば気が済むのだろうか。

他にも、ATOK は私がボケそうな機能をたくさん備えてくれていた。いや悪い意味ではなく、それくらい便利な、という意味で。

例えば入力支援という機能がある。要するにタイプミスしてもちゃんと変換してくれるという機能だ。『へんｋん』や『へんっかん』と入力してしまったとしても、ちゃんと『変換』に変換し直してくれる。なくても事足りるかもしれないが、あれば非常に強力な機能である。ミスをなかったことにできる分、時間も節約できるし、書いているときの勢いを遮断しなくて済むのでゴリゴリ書き進められる。多分ストレスも軽減される。

それから翻訳変換。読んで字のごとく、日本語を他の言語に変換することができる機能である。対応言語は英語、中国語、韓国語、ドイツ語、イタリア語、フランス語、スペイン語、ポルトガル語の 8 カ国語。翻訳の精度は果たしていかほどのものかと検証してみた結果、英語と中国語に限って言えば、Google 翻訳とどっこいどっこいといったところだった。おそらく短い文や慣用句なら ATOK が勝るだろう。ただ長文であったり、微妙なニュアンスの違いによる助動詞の使い分けはまだ発展途上かな、といった印象だ。少なくとも「はいつくばる」をちゃんと“Grovel”に変換できてる時点でえらい。“Yes Tsukubaru”はいく

らなんでも論外。Google 翻訳が「はいつくばる」を正しく変換できる日がそう遠くないことを祈るばかりである。さらに忘れてはならないのが、これが翻訳機ではなくてあくまで IME だということ。インターネット接続は必要なようだけれど、連想変換同様にわざわざブラウザを開いて検索しなくていいという最大の利点が付随している。

ここまででわかるように、ATOK はどうやら「検索しなくてもいい」ことにこだわり抜いている。その代表例とも言える機能が、このクラウド辞典サービスだ。調べたい言葉を選択してポチポチっとやるだけでいろんな辞書で調べることができる。広辞苑にはじまり、大辞林、和英英和辞典、ことわざ、慣用句、敬語。言葉の意味を調べることに関して言えば、ブラウザはほぼ完全に必要なくなる。インターネット接続はいるけど。

他にも色々と機能はありそうなのだが、ここへきて私は既に我慢の限界だった。

おあずけはもう結構、はやく小説を書いてみたい。

3.4 「実践」

逸る気持ちを抑えながら、いま私が書いている小説の続きに手を付けた。数行書いて、いつものように悩むところにさしかかる。こんな感じのニュアンスだという言葉を入力して連想変換。しっくりくる表現をあっさり見つけてなめらかに次の場面へ。

いままでことあるごとにつっかえていたのがまるで嘘のようだ。長年にわたる鼻づまりが解消されたかのような快適さで、アイディアが湧き次第、次々と思うような表現に落とし込んで書き進めていくことができる。

——思った通りだ。どころか、私が想像していた以上に「検索しなくていい」ことがもたらす利益は大きかった。

「ブラウザで検索しなくていい」ということは、とりもなおさず「テキストエディタから離れなくていい」ということである。書いているのに、集中しているのに、せっかく筆が乗ってきているのに、度々「検索する」ことで気分をぶつ切りにしてしまっては乗る筆も乗らない。逆に言えば、書いているその場から離れさえしなければ、延々と没頭できる。

思えば、小説を書き始めたばかりのころはこうして書くことに没頭することが多かった。語彙の豊かさだとか、文章の正確性だとか、そんなことは気にも留めないで、ただ純粋に書くことを楽しんでいたように思う。それが大人になって、稚拙な文章では我慢できなくなり、いわゆる『文章力』というものに囚われるようになってからはあまり没頭しなくなった。できなくなっていた。

けれど今。『文章力』をサポートしてくれる強い味方を得た今、私は再び書くことに没頭できている。書くことが楽しい。私の頭の中で流れる映像を途切れさせることなく表現できるのが楽しくてしかたがない。

どれほど時間が経ったかわからない。ふと視線をずらしたときに、画面右下にポップアップウィンドウが表示されているのに気がついた。そこにはコーヒーのアイコン、そして ATOK のロゴとともに『少し休憩しませんか？』の文字が。

なるほど、こんな機能もあるのかと微苦笑を浮かべ、クリックしてみると、私の疲労度だとか入力時間などが表示されている。それを見て思わず目を見開いた。

連続入力時間、313 分。

「こんなに……」

自分でも驚いた。こんなに没頭したのははじめてだったからだ。顔をあげると同時に急激に強い眠気が襲ってくる。集中するあまり疲労に気づかなかったらしい。それだけ没頭していたのだと思うと頬が緩んだ。

抗いがたい睡眠欲に逆らうことを早々に諦め、私はその場で机に突っ伏すと、あっという間に夢の世界に旅立った。

私の胸は ATOK を以てしても言い表しがたいほどの感動で満たされていた。

4　「皆さんも ATOK を使ってみませんか」

「それでどうだった？」

翌日、再びスタバで相まみえた先輩に ATOK を使った感想を聞かれ、私はまさしくドヤ顔で昨日の成果を突きつけた。

「じゃーん」

「何これ」

「昨日一日で書いたんです」

先輩は信じがたいような目で私を見ながら、いぶかしげにパソコンを操作した。

「16200 字――あなたこれ何時間かけて書いたの？」

「313 分だそうですよ。凄いですねあの ATOK ってやつは。おかげでいまだかつてないほど集中できました」

「約 5 時間……1 時間につき 3200 字」

珍しく先輩が驚いた顔をしていた。自分でもなかなか凄いのではないかと思う。何せあの森博嗣が 1 時間で 6000 字。速度こそ半分と劣るものの私はそれを連続 5 時間続けたのだから、多分誇っていい。

「これほどハマるとは思ってなかったけど……どうやらあなたに合っていたようね」

「ええ、ぴったりでした。というか先輩もアレ使った方が良いと思いますよ」

「私は語彙力あるもの、Google 日本語入力で十分よ」

先輩はそう言って自分のパソコンを取り出すと、自分が書いた作品を私に見せた。ぱっと見、なんか黒い。ジョジョの漫画をはじめて目にしたときみたいなあの感じだ。ざっと目を通してみると、クラクラするほど難しい単語や漢字の数々が使われていた。え、なにこれ漢文？

「先輩……これむしろ読みづらくないですか」

「そんなことないわよ。まあ確かに、賞に応募したときとかに多少『語彙の豊かさを衒うがごとく云々』みたいな評価が返ってくることはあるけれど」

「駄目じゃないですか」

「違うわ。きっと審査員が私の語彙力についていけなくて嫉妬したのよ」

「えぇ……」

語彙力がありすぎるのもかえって難儀なものである。先輩レベルの変態になると、もは

や小説を書くために書いているのではなく、文章力のある文を書くのために小説を書いているようなものなのだろう。

けれど私は思う。やはり文章力は書きたいものを書くために利用するものである。そのために ATOK という、ある意味ちょっとズルい道具を使うのは、私はアリなんじゃないかと思う。

相変わらず熱湯そのものなコーヒーが冷めるのを待ち、立ちのぼる湯気を眺めながら考えを巡らせる。

さて、今度はどんな物語を書こうかな。

※この物語はフィクションです。

※この記事の執筆にも ATOK が使用されています。

GPD MicroPC 買ったねん (575)

文　編集部　突撃隊

1　小型の PC を買った

こんにちは、新入生の皆さん。

突然ですが、自作 OS や自作 CPU をするという趣味は近年大衆化しつつあるらしいですよ[*1]。皆さんもおそらくはこれらの趣味に手を出してみたいと思っているでしょう。

自作 OS には、**実機検証**という大事なフェーズがあります。OS はハードウェアを扱う分野であるので、実際のハードウェア上で動かしてみるという検証は大きな意味を持ちます。しかし、この実機検証はお金がかかる上に、実際に購入してから検証しないとわからないデバイスが載っている可能性があり、なかなか踏み込みづらい部分でもあります。

筆者は約 1 週間前[*2]に **GPD MicroPC** という小型の PC を購入して、筆者の自作 OS[*3]の検証用として使用しています。本記事では、OS 自作をしたい新入生の皆さんのために GPD MicroPC の特徴を記していきたいと思います。

2　カタログスペック

公式っぽい Web ページ[*4]をまずは見てみましょう。筆者のコメントもおまけで付けておきます。

価格: 45,900 円 (**税抜**)

検証用にこの値段はちょっとお高いかもしれないが、趣味に使うお金だと思えばまあ……

CPU: Celeron N4100

Celeron と侮るなかれ、コアも 4 つあるし Intel-VT にも対応している。マルチコアの実機検証もできるね。

メモリ: LPDDR4 8GB

8GB もあれば自作 OS には困らん。

ストレージ: SSD 128GB

Bus が SATA 3.0 で Transfer Protocol が AHCI なので、NVMe よりはドライバが書きやすそう。

ネットワーク: RJ45

[*1] https://diary.shift-js.info/building-a-riscv-cpu-for-linux/ に書いてあった

[*2] 2020 年 3 月 20 日

[*3] https://github.com/Totsugekitai/minOSv2

[*4] http://gpdjapan.com/gpd_microPC_info/

これがあると嬉しいシリーズ 1。ネットワークドライバを書こう。

シリアル通信: RS232

これがあると嬉しいシリーズ 2。端子がオスなので、ケーブルはメス端子がついたものを手に入れよう。

その他入出力: USB3.0 x 3, USB3.0 Type-C x 1, HDMI2.0 x 1

USB ドライバは実装が大変そうですわね。HDMI はよくわからん。

3 自作 OS に使用してみての感想

UEFI からブートしてシリアル通信をするところまで検証ができたので、注意点とか使用感とかレポートします。

3.1 UEFI で Print() すると 90 度回転して表示される

多分これは縦画面用のディスプレイを無理やり横画面にして使っているからだと思います。UEFI 環境を抜けた後も回転したままです。フレームバッファと画素の対応も縦画面用になっているので、自作 OS での画面表示は座標変換を自力で行う必要があり、少々面倒でした。

3.2 シリアルポートの I/O アドレスが機体によってまちまち

自分が購入した機体はシリアルポートの I/O アドレスが `0x2e8` でしたが、知り合いたち[*5]にそのことを報告したところ、自分は違う I/O アドレスだとそれぞれから言われました。

シリアルポートの I/O アドレスは `0x3f8,0x2f8,0x3e8,0x2e8` の 4 つがあり、どのアドレスが使われるかはメーカーの実装依存です。多分ロットごとにちょいちょい違う部品を使っているのでしょう。

4 書くことがなくなったので次回予告

UEFI からブートしてシリアル通信をするところまで検証が終わったと言いましたが、実は PCI デバイスのスキャンをして AHCI デバイスを検知するところまではうまくいっています。なので次の記事は AHCI ドライバを書いて実機検証をする記事になると思います。

みんな楽しみに待っててくれよな！！！

[*5] $n = 2$

5　写真集

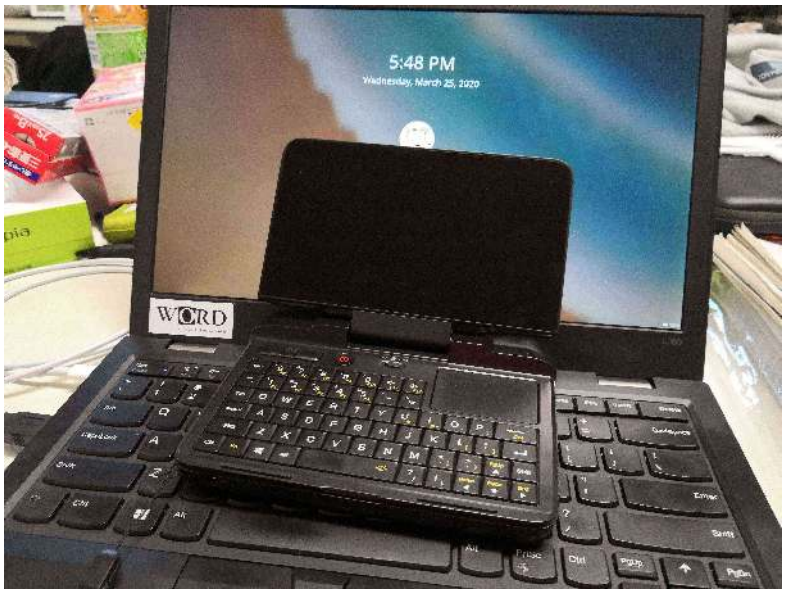

図 1　13 インチ ThinkPad とのサイズ感比較

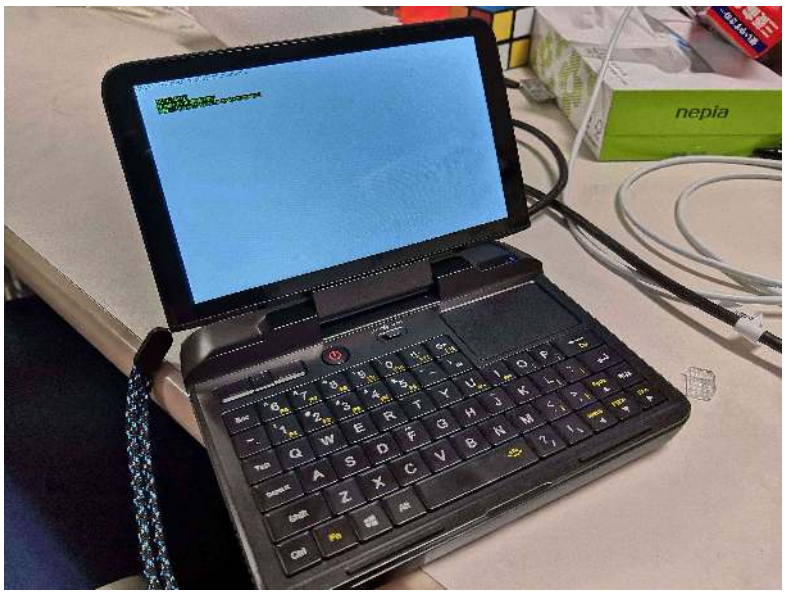

図 2　自作 OS 動作確認

2020 年度版 学生特典を使い倒す方法

文　編集部　青木勇樹

新入生の皆さん、ご入学おめでとうございます。

大学に在学している間、様々なオンラインサービスの利用料金が割引されるなどの特典を利用できます。また、学生の間だけしか利用できないサービスなどもあります。思いつく限り以下に挙げます。

- Amazon Prime が半額になる
- Spotify などの音楽サブスクリプションサービスの利用料が半額になる
- 有償ソフトウェアが無料で使える
  - Office 365
  - Windows 10 Education
- PC やタブレットを学生価格で購入できる
- GitHub Pro を無料で使える
- AWS で毎年 40 ドルのクレジットがもらえる
- Adobe Creative Cloud が安くなる

筑波大生ならば、上記に加えて

- eduroam アクセスポイントのアカウントがもらえる
- E メールアドレスを 2 つもらえる
- トレンドマイクロ社製アンチウイルスソフトが無料で使える
- Mathematica を無料で使える (対象組織のみ)
- Apple on Campus で学割よりさらに安い価格で Apple 製品を購入できる
- 学内無線 LAN システムに接続できる

情報科学類生は上記に加えて

- 以下の有償ソフトウェアが無料で使える
  - Windows 10 N
  - Windows Server
  - Windows Embedded 8.1 Industry Pro
  - Microsoft SQL Server
  - VMware Fusion
  - VMware Workstation
  - など
- coins.tsukuba.ac.jp ドメインの E メールアドレスがもらえる

- AWS のクレジットが 40 ドルから 100 ドルに増額
- 情報科学類無線 LAN に接続できる

しかし、学生特典の案内は人目につかない場所からしかリンクされていなかったり、特典入手方法の解説が丁寧になされていなかったりすることがあります。本記事では、先ほど紹介したものの中から筆者がオススメしたいものと入手方法がわかりにくいものを中心に選び、その特典を得る方法を解説します。

本記事では s.tsukuba.ac.jp(s アドレス) と u.tsukuba.ac.jp(u アドレス) ドメインの E メールの受信設定を済ませてあることを前提とします。受信設定については全学計算機システムをご覧ください。https://www.u.tsukuba.ac.jp/email/

1 学内無線 LAN システム

筑波大学の屋内のだいたい全域で利用可能な無線 LAN システムです。学群生なら誰でも使えます。本システムでは以下の SSID のアクセスポイントが設置されています。

- utwlan-x, (一部の教室は utwlan-xa)
- utwlan-pub
- utwlan-w

正式な接続方法は、utwlan-pub と utwlan-w を使った方法ですが、utwlan-pub で utwlan-w のセキュリティキーを取得する必要があります。さらに、utwlan-w に接続するたびに注意事項に同意しなければいけません。

対して、utwlan-x は試験運用扱いではあるものの、初回接続時に統一認証 ID を入力するだけで接続でき、注意事項の同意画面も表示されません。ここではもっとも設定と利用しやすい utwlan-x の利用方法を解説します。

utwlan-x の存在がそれほど広まっておらず、接続が面倒な utwlan-w の利用者が多く居るようです。本記事で便利な utwlan-x を知っていただければ幸いです。利用者が広まれば、試験運用から正式運用に切り替わるかもしれません。なお、utwlan-x は試験運用のため、学術情報メディアセンターはサポートを行っていません。トラブルの際は自力で解決する必要があります。接続する前に注意事項 (https://www.cc.tsukuba.ac.jp/wp/service/notice/) をお読みください。

1.1 必要なもの

- 学生証
- 統一認証パスワード

1.2 手順

1. 端末の Wi-Fi 接続画面を開く。
2. utwlan-x を選択する。

3. 以下のような設定で接続する。

認証方式:	EAP-PEAP
ID:	学生証裏面のバーコード下にある数字 13 桁
パスワード:	統一認証パスワード
フェーズ 2 認証:	なし
認証・暗号化方式:	WPA2 エンタープライズ AES
CA 証明書:	検証しない、もしくは国立情報学研究所または筑波大学の証明書を選択

項目の有無や表記などは OS や環境により異なります。

2 Amazon Prime

Amazon Prime 会員になると、

- Prime Music で 200 万曲の楽曲を聴き放題
- Prime Video で映画やアニメを見放題
- お急ぎ便が無料
- お届け日時指定便が無料
- 常に送料が無料になる
- Prime Reading で対象の本を読み放題
- Amazon Photos に写真を保存し放題

などの至れり尽くせりな特典を受けられます。

通常年会費が 4900 円のところ、学生は半額の 2450 円で利用できるようになります。さらに無料期間が 6 ヶ月に延長されます。

2.1 必要なもの

- ac.jp ドメインのメールアドレス

2.2 手順

1. Prime student ページにアクセスする。
2. Amazon にログインする。
3. ac.jp ドメインの E メールアドレスと卒業予定年月を入力する。
4. 支払い方法を設定する。クレジットカード、携帯キャリア決済、Amazon ギフト券で決済可能。
5. E メールに記載されたリンクにアクセスする。

3 Spotify

Spotify とは、4000 万曲を超える楽曲を配信する世界最大手の音楽配信サービスです。月額課金制の Spotify Premium に登録すると、広告が表示されなくなり、音質も上がり、オフライン再生ができるようになり、スマートスピーカーからの利用が便利になります。

通常の Spotify Premium 月額料金が 980 円のところ、学生は 480 円で利用できるようになります。

3.1 必要なもの

- 統一認証 ID

3.2 手順

1. Spotify にログインする。
2. `https://www.spotify.com/jp/student/` にアクセスする。
3. 画面の指示に従い E メールアドレスや、大学名、卒業予定年月を入力する。
4. 統一認証システムにログインする。

4 Office 365

Office 365 とは、オフィススイートのサブスクリプションサービスです。Word、Excel などのソフトウェアをダウンロードできます。大学の授業で必須なので、インストールを強く推奨します。

4.1 必要なもの

- ac.jp ドメインのメールアドレス

4.2 手順

1. `https://www.office.com` にアクセスする。
2. **u アドレス**で**新規アカウント**を作成する。[*1]

5 Windows など

Windows とは、Microsoft が提供する基本ソフトウェア (OS) です。

筑波大生は Windows 10 Education を PC 1 台に無償でインストールすることができます。Windows 10 Education とは、教育機関向けに提供される Windows エディションで、Enterprise エディションと同等の機能を利用することができます。

情報科学類の学生は Windows 10 Education に加えて、

[*1] s アドレスには Outlook 以外のサービス利用権が付与されていないため、u アドレスでアカウントを作る必要がある。

- Windows 10 N
- Windows Server
- Windows Embedded 8.1 Industry Pro
- Windows 8.1 Pro
- その他開発者向けツール

などをそれぞれ PC 1 台に無償でインストールすることができます。

5.1 必要なもの

- 筑波大学の学籍 (Education 以外は情報科学類の学籍)
- 5GB 以上のディスク空き領域
- 5GB 程度のファイルをダウンロードできるインターネット回線

5.2 手順

1. Azure for Student(`https://azure.microsoft.com/ja-jp/free/students/`) にアクセス。
2.「今すぐアクティブ化」をクリック。
3. サインインを求められたら、s.tsukuba.ac.jp ドメインの E メールアドレスを入力。
4. 筑波大学の認証画面にリダイレクトされるので、統一認証パスワードでログイン。
5. SMS 認証などの手続きを行う。
6. Download software をクリックするか Learning resources のソフトウェアをクリックする。ここで別のページに遷移すると元のページに戻れなくなります。(図 1)

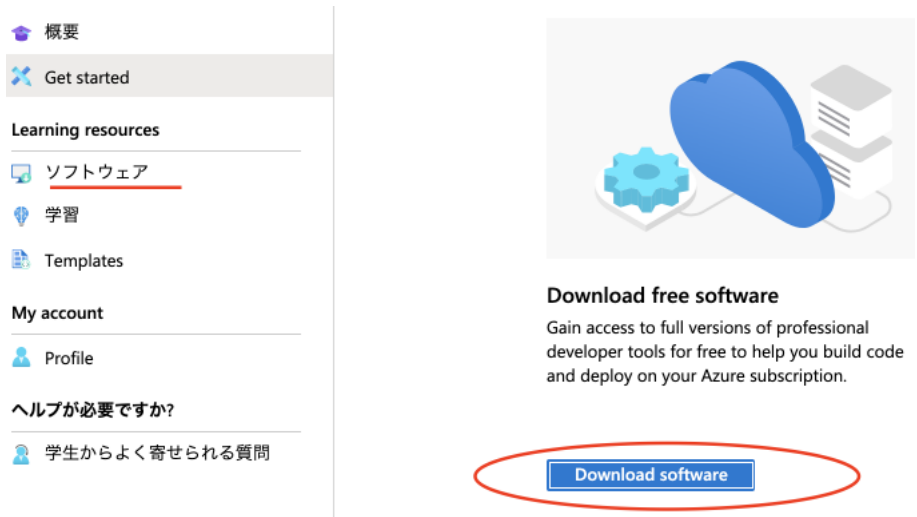

図 1　Azure Education

7. 欲しいソフトウェアを選択。
8.「キーを表示する」をクリックし、プロダクトキーを控える。

9. ダウンロードする。
10. ダウンロードしたディスクイメージファイルを使って何らかの方法でインストール対象の PC にインストールする。[*2]

6 Apple on Campus

大学生、専門学校生などの学生やその教職員は学生・教職員価格で Apple 製品を購入できます。さらに筑波大学は Apple on Campus プログラムに参加しており、学生・教職員価格からさらに割引されます。

例えば MacBook Air(2020, Core i3, 8GB RAM, 256GB SSD) を、通常価格 104,800 円のところ、学生は 93,800 円、Apple on Campus を使うとさらに安い 90,048 円で購入できます。

詳しい条件は学術情報メディアセンターのウェブサイトをご覧ください。`https://www.cc.tsukuba.ac.jp/wp/service/apple-on-campus/`

6.1 必要なもの

- 統一認証 ID

6.2 手順

1. `http://info.u.tsukuba.ac.jp/AOC/` にアクセスする。
2. 統一認証システムにログインする
3. "To Apple Store" リンクをクリック
4. 購入する。

7 VMware 製品 (情報科学類のみ)

VMware とは VMware,Inc が提供する仮想化ソフトウェアです。簡単に言うと PC のなかに擬似的に別の PC を作り出すソフトウェアです。

情報科学類生は macOS 向けの VMware Fusion と、Windows, Linux 向けの VMware Workstation を無償で利用できます。

7.1 必要なもの

- 情報科学類の学籍
- tsukuba.ac.jp ドメインの E メールアドレス
- 仮想マシンを実行できるマシンスペック

[*2] インストール方法は各自でお調べください。

7.2 手順

1. 「ソフトウェア貸出」(https://www.coins.tsukuba.ac.jp/ce/pukiwiki.php?cmd=read&page=%A5%BD%A5%D5%A5%C8%A5%A6%A5%A7%A5%A2%C2%DF%BD%D0) にアクセスする。
2. ユーザー ID は s<**学籍番号下7けた**>*3、パスワードは統一認証パスワードでログインする。
3. 「ソフトウェア貸出」ページに従い、メールを送る。
4. kivuto.com からアカウント作成リンクが入った E メールが届く。
5. そのリンクからパスワードを設定する。登録したパスワードは紛失・漏洩しないように注意すること。
6. https://onthehub.com/ にアクセスする。
7. Find Your School から"University of Tsukuba - Department of Computer Science"を選択する。(図 2)
8. 手順 5 で設定したパスワードでサインインする。
9. ダウンロードしたいソフトウェアを選択し、カートに入れる。(図 3)
10. 「ご注文手続き」に進む。
11. シリアル番号が出てくるのでシリアル番号を控える。(図 4)
12. ダウンロードする。

8 学割を利用すると?

表 1 大学一年生で節約できる金額

Amazon Prime 12 ヶ月	-2,450 円
Spotify 12 ヶ月	-6,000 円
Office 365 Solo 12 ヶ月	-12,984 円
Windows 10 Pro	-28,380 円
MacBook Air(2020 最小構成)	-14,752 円
VMware Workstation 15.5 Pro	-19,855 円
合計	**-75,161** 円

本記事で紹介した学生特典を利用すると、1 年で 75,161 円も節約することができます。(表 1) 皆さんも是非学生のうちに学生特典を使い倒しましょう。

*3学籍番号が 202011999 ならば s2011999となる。

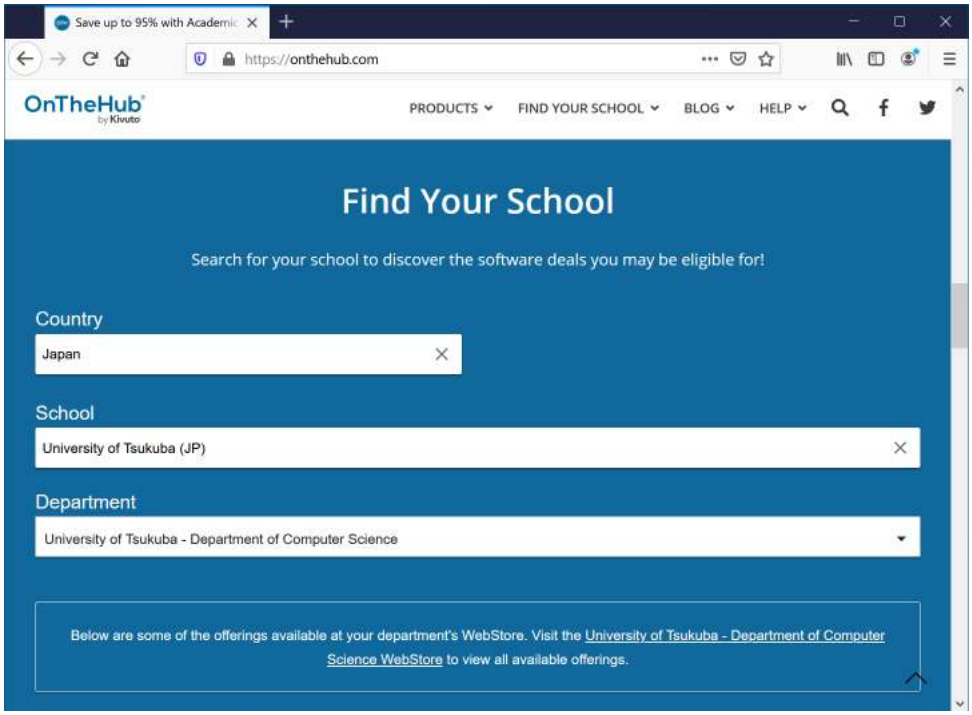

図 2　Find Your School

図 3　VMware Workstation をカートに入れる様子

図 4　シリアル番号画面

Debugging makefiles is HARD

文　編集部　リザウド (@rizaudo)

1　初めに：Makefile をデバッグしよう

言語に依存しないビルドツールを使いたいとなった時、まず選ばれるのは Make でしょう[*1]。しかし可読性が良く**正しく動く** Makefile を書くのは難しい。そして大抵の Makefile はデバッグ出来ない。このような状況に少しでも立ち向かう為に、今回は Makefile の書き方やデバッグを Tips 集形式でお送り致します。

筆者は GNU 系統のフリーソフトウェアにコミットし始めて 5 年程、一般 OSS での Makefile メンテ経験はもっとあるので、多分そこそこ Makefile のスキルがあるはずです[*2]。

なお、お断りしておくのですが、ここでは GNU Make を中心に考えていきます。残念ながら今では主役とは言いにくい BSD Make であったり、古の時代に存在したと言われる、Berkeley Make や System V Make 等は全く考えません[*3]。ちなみに、現代でそういった Make を使う要求が有る場合は、こういった Make 解説記事を読む前に数十年前の Make を使わなきゃいけない理由を毎秒・毎脳クロックごとに考えるところから始めるべきでしょう。

2　予防的な記述

具体的なデバッグ手法を解説する前に、デバッグしやすい Makefile の記述を解説します。予防こそが最善のデバッグと言われるように、開発者が間違えないように作り込んでおくのは肝要です。

ちなみに、初歩の初歩として記述しておくのですが、MacOSX とかいう OS に入ってる超古い Make は捨てましょう。一切サポートする必要はありません[*4]。

2.1　変数参照には$() を使う

Make は$の後は基本的に一文字しか読まないので、変数参照はカッコを使って間違われないようにしましょう。

Listing 1: 変数参照の例

```
1    $DEBUG # は D 変数と EBUG になる。
2    $(DEBUG) # は正しく DEBUG 変数を参照する
```

[*1] ここで Bazel とか選ぶ人間はちょっと G 社員ですね
[*2] そう信じたい。ｱﾄｲﾁｵｳﾊﾒｲｸﾆﾓﾊﾟｯﾁﾅｹﾞﾀｼ.....
[*3] この時代の Make 考古学をしたくなったら多分お役に立てるので筆者までご連絡下さい
[*4] Mac は古いツールばかりだからサポート嫌われるんですよね

2.2 タブを使わない

標準だとレシピルールのプレフィックスはタブになっていますが、これが変更出来るバージョン (GNUMake だと 4.0 以上) ならば自由に設定できます。これが定義出来ないバージョンの Make は余りにも古すぎるので新規にサポートすべきではないでしょう。.RECIPEPREFIX を使えば、制限はありますが任意の文字をプレフィックスとして使えます。例えば、以下とかは正常な Makefile です

Listing 2: RECIPEPREFIX の変更

```
1 .RECIPEPREFIX = >
2 hello:
3 > echo ''HELLO''
4 > echo ''valid line''
```

これだけでタブじゃなくホワイトスペースになっていて、アレレ？ という状態からは開放されます。

2.3 再帰 Make は避けよう

呼び出された Make から全体の無限ループを検出することが出来ない事やサブディレクトリが読まれる順番が制御不可能など、根本的に検知不可なバグを埋め込んでしまうため、再帰的に Make を使う事は絶対にやめましょう。これをやってメンテし続けられるのは Make の達人中の達人だけです。この辺りの話は「Recursive Make Considered Harmful[*5]」でも読んで下さい。また、サブ Make への変数の与え方や.PHONY の書き方等を工夫したとしても、同じような問題を生んでしまうのでサブ Make を呼び出すのもやめておくべきです。どうしても使う必要がある場合は、Make における環境変数や変数の処理階層を理解した上で行うべきでしょう。しかし筆者としてはこういった事が必要になっていくと、そのうち Makefile 自体にテストが必要になると思うのでオススメはしたくないです。

どうしてもやる時のヒント

.EXPORT_ALL_VARIABLES のような便利な設定があるので、こういった設定を使っていきましょう。この辺りはちょっと難しい挙動をするため、本稿では詳しい使い方は説明出来ません。しかし調べればやり方がわかるでしょう。

2.4 bash を使おう (もしくは任意のシェル)

Make のデフォルトは/bin/sh なので、bash 等に変えましょう。可搬性や安全性の観点から Make 自身が使うシェルは POSIX 標準なシェルだけにすべきです。しかし、何らかのスクリプトを使いたい等の要望が有る場合は後述の.ONESHELL を使った上で、個別にシェルを呼び出す事を考えても良いでしょう。

[*5] http://aegis.sourceforge.net/auug97.pdf

Listing 3: SHELL 変数で使用するシェルの変更

```
SHELL := bash
```

2.5 [主に bash を使う場合] シェルフラグの設定

「Your Makefiles are wrong*6」のようなブログ記事でも紹介されていますが、bash には Strict Mode という便利なモードがあり、この辺りのオプションを使うと安全にターゲットレシピを書くことが出来ます。bash を使わなかったとしても、シェルに渡すフラグをちゃんと管理しておくのは重要でしょう。例として筆者も使う Bash 向けのオプションを紹介しておきます。

Listing 4: Bash 向けの一例

```
.SHELLFLAGS := -eu -o pipefail -c
```

2.6 .ONESHELL

Make はデフォルトではレシピ 1 行に対しシェル 1 つを立ち上げるようになっています。そのためよく見かける Makefile では行末に\が置かれている事が多いのですね。しかしこのような書き方では、書き忘れた際にシェル環境の断絶が起き破滅的な挙動になります。

このようなことは避け、一つのターゲットルール全体に対してシェル一つにする、.ONESHELL を使いましょう。ターゲットとして宣言しておくだけです。

Listing 5: .ONESHELL の定義

```
.ONESHELL:
```

2.7 未定義変数を警告する

はい、タイトルそのままです。Typo が有ってもそのままじゃ何も反応してくれません。Make 起動時にフラグを渡してもよいのですが、筆者は Makefile 側で常に警告フラグをオンにしておく事をオススメします。

Listing 6: warn-undefined-variables

```
MAKEFLAGS += --warn-undefined-variables
```

*6 https://tech.davis-hansson.com/p/make/

2.8 ユーザに変更されない変数定義

override ディレクティブを使うと、ユーザが Make 実行時に変数指定をしていたとしても変数代入が必ず行われます。ユーザが勝手に設定すると困る物や、複雑な変数代入をしている場合はこれを使うと良いでしょう。

Listing 7: override で CFLAGS に追加する例

```
# ユーザがどんな指定をしていたとしても､必ず CFLAGS には-g を渡したい場合
override CFLAGS += -g
```

2.9 再帰展開変数と単純展開変数のどちらが必要かを見極める

Make の変数は再帰展開変数と単純展開変数の 2 つの型があります。本質的な説明をしてしまうと Make の**闇**に手を入れて皆さんの目と脳を焼いてしまう為、単純化した説明をします。簡単に違いを説明するならば、Make によって評価された際に変数内の変数を再帰的に評価されるか (再帰展開変数) そのままの文字列として評価されるか (単純展開変数) の違いです。説明で難しいポイントは Make が扱うデータはほぼ全部がファーストクラスオブジェクトのようなものなので、関数も変数になりますし関数クロージャのようなものも変数になり得ます。まあユーザとして使う分には簡単な理解で何とかなるので、気にせずに Make との日々をお過ごし下さい.....

そして説明が終わった後で Tips の中身に入っていきますが、見出しの意味そのままです。再帰展開変数を扱うのに失敗した場合、大抵は Make が変なループに陥ってるのを検出してくれますが、複雑なケース (複数の子 Make を起こしている場合等) では問題を引き起こします。違いが解ってないならば、値を固定したい場合は ':=' もしくは '::='(**基本的には**同じ意味です) で、まだ値の展開をしたい場合は '=' を使うというような理解でいきましょう。なに、死にはしませんよ.....

2.10 .DELETE_ON_ERROR

Make が問題によって中断された場合の中間ファイルが残っている場合、次の実行に問題を起こす場合があります。そういうのは望ましくないので.DELETE_ON_ERROR をターゲットとして宣言しておくことで自動的に削除するようにしましょう。ちなみに、.INTERMEDIATE の依存ターゲット指定で中間ファイルを指定できるので、Make が暗黙的にルールを持っていない生成物の場合は設定しておきましょう。

Listing 8: 中間ファイルをどのような中断時でも削除する

```
.DELETE_ON_ERROR:
```

Listing 9: .INTERMEDIATE

```
1  .INTERMEDIATE: %.jar %.ika
```

キャッシュのための.PRECIOUS

どうしても生成に時間がかかり、中間ファイルとして保管しておきたいというファイルが有る場合、.PRECIOUS の依存ターゲット指定で設定しておく事で Make の中間ファイル削除の対象から外す事が出来ます。そのままだと消えなくなるので、適切な clean レシピを用意しましょう。

2.11 .PHONY をしっかり定義する

慣れた方には当然の話だと思いますが、何者かにターゲットと同名のファイルが作られてしまった場合 Make は最新であると判断してターゲットビルドを走らせません。具体的な例をあげると、clean ターゲットが有った際に何者かが嫌がらせで Makefile ルートディレクトリで、*clean* というファイルを作ったとします。この状態で *make clean* しようとした時、Make は clean というファイルが最新であるので実際の clean ターゲットへのレシピを実行致しません。フゥー、最高ですね？[*7]

こういった事態に陥らないため、偽のターゲット (Phony Target) 指定をする.PHONY はしっかり定義しておきましょう。.PHONY の依存ターゲットとして指定されたターゲットは偽のターゲットであると Make は認識するため、ターゲット名と同名のファイルを完全に無視します。

2.12 define 文で同じ手続きを一変数にまとめておく

define 文を使うことで、何度も行うような一連の手続きを纏めておけます。以下の例を見て下さい。

Listing 10: define の例

```
1  define run-awk
2    @echo ''Prints the first field of each line in each file.''
3    @awk '{print $$1}' $^
4  endef
5  #使う際は $(run-awk) で良い
```

ここで、レシピを書いている訳でないのに自動変数を使っている事に気づいたことでしょう。これは合法です[*8]。この define で作られた変数が展開される時に、自動変数も解釈されコンテキストに有った物となっているので問題がありません。

[*7] 編集部注：いいえ

[*8] 読者の方には慣れない言い方かもしれませんが、単純に昔から UNIX 文脈の翻訳ではで legal の訳語が合法なんですよね。筆者も日本語 man を書くのでこういう言い方が好みです

ちなみに、define した関数に引数を渡す事も可能です。call で書いた内容そのまま (空白も含むということです) が送られる事に注意して下さい。

Listing 11: define で引数を取る関数を作る

```
define run
  @echo $(1)
endef
hello:
  $(call run,hello)
```

2.13 二重コロンでの分割ターゲット定義

ターゲットを記述する際に二重コロンを使うと、複数回ターゲットを記述できます。Makefile が 1 ファイルであれば可読性程度の利点ですが、複数 Makefile のプロジェクトの場合それぞれの Makefile 毎の clean を書きたいという事態があります。そういった場合には担当範囲の分割の観点から、無理矢理に 1 つのターゲット記述に纏めずに、二重コロンを使った複数記述に切り替えておくべきでしょう。複数ファイルの場合、周辺コンテキストの情報を渡すのも一苦労ですので。

Listing 12: 二重コロンによる分割ターゲット記述

```
hello:: $(hell)
  @echo "hello1"

hello:: $(some)
  @echo ''hello2''
```

一応書いておくのですが、分割ターゲット記述がされたターゲットは定義順で実行されます (上だと hello1,hello2 という順で出力されます)。つまるところ Makefile の解釈順という事になります。自身の思った解釈順通りになっていることを祈りましょう。さもなくばちゃんと副作用が可能な限りないように書きましょう。

2.14 環境変数を変更しない

export ディレクティブを使うと Make 側から環境変数を変更できるのですが、これは行うべきでないです。何らかのコマンド実行の為に、他に方法がないという場合は仕方がないですが、基本的にやって良いことはないです。

3 実際のデバッグの為の手法

Make には*print-data-base* であったり、*debug=a* というようなデバッグ向けの内部情報出力があります。しかし、余りにも大規模な出力なため、何も解ってない状態でこういった物に当たるのは辛いでしょう。

この章では Make に与えたオプションで出てくる情報をどう見ていけばよいか、またはどう書くとデバッグしやすいかという手法を紹介しておきます。

3.1 暗黙的なルール及び変数を排除する

Make にはビルドインの大量の暗黙的なルール及び変数が入っています。しかし、デバッグの際にいちいち fortran やら ALGOL のルールを検討したログを出して欲しくはありませんよね？ そのような時に必要なオプションが*-r* と*-R* です。*-r* は暗黙的なルールを排除し、*-R* は暗黙的な変数 (とルール) を排除します。知っているか知っていないかでかなりデバッグのしやすさが違うので、読者の方々はぜひ覚えておいて下さい。

3.2 dry-run(もしくは-n) オプション

実行出来るコマンドを出力して、実際には何もしません。ターゲットとそのレシピを書いたばかりの時には必ず行って、意図したような処理になっているかを確認すべきでしょう。

3.3 print-data-base オプション

あまり最初からこのような最終手段を紹介するのもどうかとは思うのですが、問題が確実に解決出来る事が重要だと思うので紹介します。Make が持つデータベース**全体**を出力させるのが*print-data-base* です。ちなみに出力変更のオプションなので、Make のデバッグだけしたい場合は*-n*(dry-run) 等で行って下さい。

この出力は文字通り**全体**を出力させます。意味はおわかりでしょう、デバッグに必要ないだろう情報も大量に出ます。ですので筆者は余りオススメしたくはないのですが、ターゲットに対してのレシピがどうなっているかや問題の変数等が解っている場合にはデバッグに有用です。

例として GNU Emacs の開発リポジトリで出力させた、変数部分の更に一部分を出します。

```
# 変数

# makefile 変数 (ファイル 'gnulib.mk', 619 行目)
NEXT_AS_FIRST_DIRECTIVE_STDIO_H = <stdio.h>
# makefile 変数 (ファイル 'gnulib.mk', 163 行目)
GNULIB_GETCHAR = 1
# makefile 変数 (ファイル 'gnulib.mk', 660 行目)
PACKAGE_TARNAME = emacs
# makefile 変数 (ファイル 'gnulib.mk', 500 行目)
HAVE_STRUCT_SIGACTION_SA_SIGACTION = 1
# デフォルト
COMPILE.m = $(OBJC) $(OBJCFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
# makefile 変数 (ファイル 'gnulib.mk', 597 行目)
```

```
LIB_TIMER_TIME = -lrt
# makefile 変数 (ファイル 'gnulib.mk', 664 行目)
PAXCTL =
# makefile 変数 (ファイル 'gnulib.mk', 638 行目)
NEXT_STDLIB_H = <stdlib.h>
# makefile 変数 (ファイル 'gnulib.mk', 914 行目)
gl_GNULIB_ENABLED_strtoll =
# makefile 変数 (ファイル 'gnulib.mk', 235 行目)
GNULIB_POPEN = 0
# makefile 変数 (ファイル 'gnulib.mk', 615 行目)
NEXT_AS_FIRST_DIRECTIVE_LIMITS_H = <limits.h>
# makefile 変数 (ファイル 'gnulib.mk', 246 行目)
GNULIB_PUTENV = 1
# makefile 変数 (ファイル 'gnulib.mk', 438 行目)
HAVE_MKFIFOAT = 1
.... この後も大量の出力があります....
```

こういった記述が延々と続きますし、各項目ごとに大量に出力されます。現実的にこの出力とにらめっこするのはやめておくべきでしょう。

3.4 Make のデバッグ出力

これもまた、出力が多くて筆者としてはオススメし難い手法なのですが、Make のデバッグ情報を見て Make の挙動をチェックする事が出来ます。

また例として GNU Emacs の開発リポジトリで、*make clean -d*(*-d* は*–debug=a*、つまり all と同じです) を行った際の出力の一部分を見てましょう

```
GNU Make 4.3
このプログラムは x86_64-pc-linux-gnu 用にビルドされました
Copyright (C) 1988-2020 Free Software Foundation, Inc.
ライセンス GPLv3+: GNU GPL バージョン 3 以降 <http://gnu.org/licenses/gpl.html>
これはフリーソフトウェアです: 自由に変更および配布できます.
法律の許す限り、 無保証 です.
makefile を読み込みます...
makefile 'GNUmakefile' の読み込み中...
makefile 'Makefile' の読み込み中 (探索パス) (~ の展開なし)...
makefile の更新中....
 ファイル 'Makefile' を検討しています.
   ファイル 'config.status' を検討しています.
     ファイル 'configure' を検討しています.
```

```
ファイル 'configure.ac' を検討しています.
'configure.ac' のための暗黙ルールを探します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.o' を試します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.c' を試します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.cc' を試します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.C' を試します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.cpp' を試します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.p' を試します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.f' を試します.
語幹 'configure.ac' とのパターンルールを試します.
暗黙の必要条件 'configure.ac.F' を試します.
... この後も続いていく ...
```

ウワー。この最初の一ページ分でもうお腹いっぱいですね。こういった暗黙のルールを考慮したという話も出力されてしまうので、とてもヒューマンリーダブルとは言い難いです。しかし、どうしても Make がした挙動の理由が解らないという場合には使えます。

3.5 Make のログ出力

基本中の基本、Makefile におけるログ出力の関数です。それぞれ UNIX ログにおける info 等に対応しています。

Listing 13: info warning error

```
1  $(info ...)
2  $(warning ...)
3  $(error ...)
```

3.6 デバッグ用変数と条件分岐

make all DEBUG=1 というような実行をした際にコンパイラにデバッグ用のフラグが渡って欲しい時、手動で CXXFLAGS 等を書き換えるのも一つの手ではありますが、Makefile 側で設定しましょう。

Listing 14: 変数の値でデバッグフラグを設定するかの判定

```
DEBUG ?= 0

setflags:
  ifeq($(DEBUG), 1)
    CXXFLAGS = something...
  endif
```

(?=は未定義ならば右辺の値を評価して入れるという意味です)

ここで、gcc 等のツールでは衝突するフラグが渡された場合には後に渡したフラグが優先される事を知っている方は、以下のような書き方にしたくなるかもしれません。

Listing 15: 筆者はオススメしない例

```
setdbg:
  ifeq($(DEBUG), 1)
    DBGFLAGS = something...
    # もしくは CXXFLAGS += something...
  endif
 compile: setdbg hoge.cc
   $(CXX) $(CXXFLAGS) $(DBGFLAGS) $<
```

これはツールの挙動をよく知っていれば使える手ですが、ビルド関係ツールの暗黙的な挙動を使うことは余り良いことではありません。挙動が簡単に変わる事がないとしても、渡す順番依存で挙動が変わる等の事が起き、非常に気をつけて Makefile を書かなければなりません。また、不要なフラグが増える事で Makefile 自体のデバッグもそうですし、プロジェクト自体のデバッグも大変になります。こういったデメリットをよく理解しているチームでなければオススメしないやり方ですね。

余談：空白値と未定義の変数

上の場所で ifdef とか ifndef を使っていない理由を一応説明しておきます。Make での ifdef というのは空白値を持つ変数 (言い方がちょっと聞き慣れないかもしれませんが、値を持たないという意味です) かどうかの判定で、未定義の変数かどうかの判定ではないのです。なので空白値を持つ変数と未定義変数の違いがしっかり解ってないと面倒な事になるので、今回の記事ではこっちにしています。解っているならば未定義変数及び空白値を持つ変数の判定は、*ifndef $(VAR),)*(未定義含めた ifdef 相当) や *ifeq($(VAR),)*(未定義含めた ifndef 相当) 等で判定すれば良いと思います。これの意味は熟練者ならわかるでしょう……

余談：本当に未定義変数であるという事を判定したい場合

筆者のテクをご紹介します。.VARIABLES という内部変数には Make 内の全ての定義変数名が入っています。ですので*$(filter FOO, $(.VARIABLES))* をすると変数 FOO が定義されているかどうかを判定できます。しかし本当に未定義変数であるということだけを判定したい場面があるのかは疑問ですね。

3.7 環境変数を強制する

場合によっては Makefile が間違った変数定義をしていておかしな事になるかもしれません。そういう時に*-e, –environment-overrides* が便利です。これは Make 変数に対して同名の環境変数があれば、環境変数の方を優先するというコマンドです。

4 それでも解決しない場合

え？ Tips これだけ？ という気持ちになるかもしれませんが、ここまでに出した Tips で本当に大抵は解決します。しかしそれでも解決しないという事態に陥った場合、Make のデバッグ用に作られた GNU Remake[*9]を使って下さい。精神的コストや苦痛が生じますが何とかなるとは思います。

5 終わりに

Makefile って難しい。

[*9] http://bashdb.sourceforge.net/remake/

WORD 編集部へのお誘い

文　編集部　突撃隊

我々 WORD 編集部は情報科学類の学類誌「 WORD 」を発行している情報科学類公認の団体です。WORD は「情報科学」から「カレーの作り方」までを手広くカバーする総合的な学類誌です。年に数回発行しており、主に第三エリア 3A 、3C 棟や図書館前で配布しています。

編集部の拘束時間には週一回の編集会議と、年に数回の赤入れや製本作業等発行に伴う作業があります。日常的に活動する必要はありませんので、他サークルの掛け持ちの障壁にはなりません。実際、多くの編集部員が他サークルと掛け持ちで在籍しています。

WORD 編集部には、情報科学類生や数学類生などが在籍しています。例えば、以下のような人達が在籍しています。

natto 当たり屋
catla メイドのオタク (機械学習も可)
sosukesuzuki JavaScript しばきマン
突撃隊 フルスクラッチ OS っていいよね

下記に当てはまる方や、WORD に興味を持った方は是非、情報科学類学生ラウンジ隣の編集部室 (3C212) へいつでも見学に来てください。時間を問わず常に開いています。

- AC 入試で入学した方
- それ以前に AC な方
- 印刷、組版や製版に興味がある方
- ネットワークの管理を経験したことがある方
- APL や Lisp が書ける方

その他質問がある方は、word@coins.tsukuba.ac.jp か、Twitter アカウントをお持ちの方は @word_tsukuba までお気軽にお問い合わせください。

情報科学類誌

WORD

From College of Information Science

入学祝い号

発行者　情報科学類長

編集長　広瀬 智之

制作・編集　筑波大学情報学群
情報科学類WORD編集部
（第三エリアC棟212号室）

2020年4月3日　初版第1刷発行　（128部）