4.プログラムを書く。

さてテキストファイルを編集する方法も覚えました。つぎはいよいよプログラミングです。C言語によるプログラム作成の手順を説明します。

4.1.Cによるプログラミング。

C言語はUNIXと非常に密接な関係があります。UNIXのOS自身かなりの部分がCで書かれています。UNIXのシステムサービスを呼び出す場合もCから呼び出すのが最も効率よく行くようになっています。またシェルから呼び出されるコマンドを作る場合も引数の受渡など、Cで書くのが最も簡単です。ですから、データ収集ソフトなどのシステムソフトを書こうというユーザはCを知らなければなりません。

C言語については非常に多くの入門書が出ていますし、パソコンなどでもC言語コンパイラーは手に入ります。ですからここではC言語そのものの解説はしません。別のところで勉強してください。ただC言語の文法は非常に簡素です。FORTRANのユーザがCを学習するのは難しくありません。ましてやPASCALユーザにとっては記号の置き換えだけのように見えるかも知れません。beginを{に、endを}に、=を==に、:=を=になど。

Cの学習で特に重視してやってもらいたいのはポインタと構造体の理解です。文法が簡単なかわりに非常に多くの関数が用意されています。それらに目を通すことも重要です。Cでは関数の引数が値渡しなのもFORTRANユーザには最初奇妙に見えるかも知れません。最近ではC++というCを拡張した言語がもてはやされていますが、いきなりC++から始めることは好ましくありません。ある程度大きなCのプログラムを書いた経験からオブジェクト指向とは何かを理解してC++に移って行くほうがC++のプログラマーとしても望ましいと思います。

4.2.ソースファイルを書く。

Cによるプログラミングをはじめましょう。まずはソースファイルを編集する必要があります。すでにテキストエディタの使い方は知っていますね。いろんな教科書に載っているようにHello World!からいきましょう。次のような内容のファイルを作ってください。名前を`hello.c`とします。

```
/*        hello.c              */
#include  <stdio.h>
main()
{
    printf( "Hello! World.\n" );
}
```

少し解説しましょう。Cでは/*と*/に囲まれた範囲がコメントになります。2行目の文はstdio.hというファイルを読み込む(includeする)命令です。このプログラムの中で使われる関数や定数を定義します。#includeはプログラムとして何か実行する意味ではなくコンパイラにたいしてコンパイルするときにこういうことをしなさいと教えています。こういう命令は#ではじまります。main()はこの名前の関数がここで定義されますと宣言しているものです。その中身は{}で囲まれた部分です。printfは標準出力に文字列を出力しなさいというライブラリー関数です。実はこれがstdio.hで定義されています。Cのライブラリー関数やUNIXのシステムコールはmanで見ることができます。例えば

```
% man printf
```

でprintf関数の解説を得ることができます。そこには#include <stdio.h>が必要だということも書かれています。このソースファイルはコンパイルし実行することができます。

4.3.コンパイル、リンク、実行する。

さきほどのhello.cをコンパイルリンクして実行してみましょう。

```
% cc hello.c
```

ccはCコンパイラ呼出しですが、この命令はほっておくと実行可能なファイルまで作ってしまいます。ccがリンカーまで呼び出してしまうのです。リンカーはldといいますが、ldを単独で呼び出すと、C言語の使うライブラリーを明示して呼び出す必要があるため、面倒です。ccはCで必要なライブラリーを取り出してリンクするようにldに命令します。ですから、コンパイルせずリンクするためだけにもccを使います。ccは狭い意味のコンパイラではありません。Cの言語プロセッサと考えるとよいでしょう。これ

は他の言語についても言えます。f77やcxxもしくはCC(C++のこと)も同じように使います。
　さて、では出来上がった実行可能ファイルを呼び出してみましょう。

```
% a.out
Hello! World.
%
```

これもUNIXらしい所ですが、ccは何もいわないでおくと実行可能ファイルa.outを作ります。lsで確かめてください。ldの作った実行可能形式ファイルをa.outファイルということもあります。当然ですがそれでは困ります。今の場合実行可能ファイルにはhelloという名がつくのが自然ですね。そのように指定することは当然できます。

```
% cc -o hello hello.c
```

-o helloというスイッチが実行可能ファイルの名前をhelloにしなさいと指定しています。めでたく、

```
% hello
Hello! World.
%
```

つぎに下のようなプログラムを書いてみましょう。ファイル名をargs.cとします。

```
/* args.c */
#include  <stdio.h>
main( argc, argv )
int       argc;
char      **argv;
{
int       i;
for( i = 0; i < argc; i++ )
          printf( "arg #%d = %s\n", i, *argv++ );
}
```

これをコンパイルします。

```
% cc -o args args.c
```

Cプログラムではそれが呼び出されるときに引数リストがargc、argvという二つのmainへの引数で渡されます。argcには引数の数が、argvは引数リストへのポインタが入ります。第１番目の引数はコマンドの名前です。このargsというプログラムは引数の値を順に表示するプログラムです。実行してみましょう。

```
% args
arg #0 = args
% args abc
arg #0 = args
arg #1 = abc
% args "This argument contains spaces"
arg #0 = args
arg #1 = This argument contains spaces
% args *
arg #0 = args
arg #1 = args
arg #2 = args.c
arg #3 = hello
arg #4 = hello.c
%
```

プログラムに引数がどのように渡されるかがよくわかります。最後の例では*をシェルがそのディレクトリ内のファイル名に置き換えました。詳しいことはCの教科書を見てください。

　ここまではファイル１つだけで作る簡単なプログラムでした。実際にはいくつかのファイルに分割されたプログラムを書くことの方が多いでしょう。その場合は他のシステムと同じように個々のファイルをオブジェクトモジュールにコンパイルして、最後にそれらをリンクして実行可能イメージを作ります。例えばmain.cとsub.cからtestというプログラムを作ることを考えましょう。main.cからmain.oを、sub.cからsub.oをつくり、最後にmain.oとsub.oからtestを作ります。

```
% cc -c main.c
% cc -c sub.c
```

```
% cc -o test main.o sub.o
```

最初の２行ではmain.cとsub.cをそれぞれコンパイルしています。-cというスイッチはコンパイルのみを行ない、ソースファイルの名前から.cをとり.oをつけてオブジェクトモジュールとして保存させるものです。最後の行はソースがみんなオブジェクトモジュールなのでリンクだけしてくれます。-o testは実行可能ファイル名をtestにします。ccの詳しい説明はman ccで見てください。

4.4.ライブラリーをつくる。

複数のファイルを作ってプログラムを書くようになると、たくさんのオブジェクトモジュールができてきて、それらの名前を一つ一つccコマンドの中で書くことが面倒になります。こういうときはオブジェクトコードのライブラリーを作ってcc(正確にはld)にそこから必要なものを探してもらうようにします。ライブラリーを作ってみましょう。関数が記述されたファイルをsub1.c、sub2.c、sub3.cとします。それぞれをコンパイルします。

```
% cc -c main.c
% cc -c sub1.c
% cc -c sub2.c
% cc -c sub3.c
% ls
sub1.c          sub2.c          sub3.c          main.c
sub1.o          sub2.o          sub3.o          main.o
%
```

実行可能ファイルを作るのに

```
% cc -o test main.o sub1.o sub2.o sub3.o
```

とやってもよいのですが、サブルーチンの集合をライブラリーにします。

```
% ar r mylib.a sub1.o sub2.o sub3.o
ar: creating mylib.a
% ranlib mylib.a
```

```
% cc -o test main.o mylib.a
```

まずarはアーカイバーです。複数のファイルを集めて一つのアーカイブファイルにします。第１引数rは置き換えを意味します。mylib.aの中に同じ名前のファイルがあれば置き換えます。なければ追加します。アーカイブファイルの中に何が入っているかは、

```
% ar t mylib.a
```

で見ることができます。詳しくはman arで見てください。次のranlibはアーカイブファイルをランダムライブラリーに変換します。ldが見てわかる形になります。最後にccでtestという実行可能形式を作っています。以後関数ファイルを修正したときは

```
% cc -c sub2.c
% ar r mylib.a sub2.o
% ranlib mylib.a
% cc -o test main.o mylib.a
```

などを繰り返せばよいわけです。

ここでライブラリーについて一般的なことを少し説明します。UNIXの標準ライブラリーは名前がlibではじまり.aなどで終わります。そして/lib、/usr/lib、/usr/local/libの中にあります。試しにls /libなどやってみてください。ですから必要なライブラリーをリンクする場合、ccでは-lオプションを使いますが、はじめのlibと.から後は省略して与えます。たとえばlibX11.sa.1.0を使いたいときは-lX11という指定の仕方をします。またそのライブラリーファイルが上述したディレクトリの中にないときは-Lオプションで指定します。

```
% cc -o testX testX.o -L/usr/openwin/lib -lX11
```

という風になります。共通に使うライブラリーは同じコンベンションで命名するべきでしょう。

ライブラリーを作る場合、その関数の型や必要な定数などを宣言するファイルを別に用意することになります。通常はそのファイルを#includeでよみこみます。C言語のシステムではインクルードファイルは通常.hで終わる名前になっています。標準的なインクルードファイルは/usr/includeにあります。ccはそこを探しに行きますが、自分の.hファイルが別のところにあるときは-Iオプションでその場所を指定します。

```
% cc -c -I~/include sub1.c
```

この場合は自分のホームディレクトリの下にあるincludeディレクトリから.hファイルを探させるものです。

arを使ってライブラリを作りましたが、arは任意のファイルをアーカイブするツールです。言い換えると中身が何かをarは知りません。ですから、いくつもの関数を一つのファイルで記述して.oファイルを作ると、.aにつっこんでも取り出すときは.oファイルごと取り出されてしまいます。効率的なライブラリを作るには関数毎別々のファイルにすべきです。でも普通はそんなややこしいことはしませんね。一つのファイルに関連するいくつもの関数を定義するほうがずっと書きやすいはずです。そうやって書かれたファイルを関数単位でばらばらにしてくれるツールがあります。言語に依存するのは当然ですが、f77の場合はfsplit、Cの場合はcsplitというコマンドが用意されています。あるオブジェクトファイルの中にどのようなシンボルが定義されているか調べるコマンドはnmです。検査の対象はオブジェクトファイルでも実行可能ファイル(a.out形式)でもライブラリでも結構です。

4.5.メイクを使う。

このようにC言語によるプログラム開発はccなどを繰り返し呼び出して行きます。先程の例を見てもわかるように毎回長い命令を入力するのは面倒です。当然ですが、こういった要求に答えてくれるツールが用意されています。makeといいます。makeは指定されたファイルの変更日付を比べて、例えばソースファイルの方がオブジェクトファイルより新しければコンパイルしてくれます。オブジェクトファイルの方が実行可能ファイルより新しければリンクしてくれます。もちろんこういったファイルの依存関係やコンパイルなどの仕方はある書式で書かれていなければなりません。その依存関係などの書かれたファイルは通常makefileもしくはMakefileというなまえにしておきます。makeコマンドは現在のディレクトリの中にあるmakefileもしくはMakefileという名前のファイルを読み込んで実行します。先程の例を考えてみましょう。makefileのなかみは例えば、

```
# makefile demo

test:     main.o mylib.a
          cc -o test main.o mylib.a
```

```
mylib.a:  sub1.o sub2.o sub3.o
          ar r mylib.a sub1.o sub2.o sub3.o
          ranlib mylib.a

main.o:   main.c
          cc -c main.c

sub1.o:   sub1.c
          cc -c sub1.c

sub2.o:   sub2.c
          cc -c sub2.c

sub3.o:   sub3.c
          cc -c sub3.c
```

#で始まる行はコメントです。書かれている順番に意味があります。上の方の式から評価されて行きます。書式は、

標的ファイル：　依存ファイル [依存ファイル...]
　　コマンド
　　...

標的ファイルの日付が依存ファイルの日付より古いとコマンドが実行されます。依存ファイルが下の方で標的ファイルになっていると先にそちらが評価実行されます。いまの例ではtestとmain.oが比較されますが、次のところでmain.oがmain.cと比較されていますからまずmain.oとmain.cが比較されます。main.cが新しいとコマンドcc -c main.cが実行されます。この結果main.oはtestより新しくなりますからコマンドcc -o ...が実行されます。このように一番上に実行可能ファイルを標的にした行を置き、下にそれが必要とするオブジェクトファイルの作り方を並べます。このコマンドの前には必ずTAB文字が入っていなければなりません。

　基本的にはこれでよいのですが、main.oからsub3.oまで同じような記述が続いています。依存ファイルが.cで標的ファイルが.oですから、cc -cを呼び出すことは明らかです。実はmakeはそのことを知っていますのでcc -cの行は書く必要がありません。さらにmain.oの依存ファイルが同じ名前の.cしかありませんから、main.o:

main.cの行も必要ありません。もちろんmain.cが例えばインクルードファイルなどにも依存する場合は記述する必要はあります。これらは暗黙の規則と呼ばれ、SUNの場合は/usr/include/makeにあるdefault.mkというファイルに記述されています。もし現在のディレクトリにもdefault.mkがあるときはそちらが優先されます。自分用の暗黙の規則が作れるわけです。

例えばコンパイルの時に自分のホームディレクトリの下のincludeというサブディレクトリにあるファイルをインクルードする場合を考えましょう。そのまま書くと次のようになります。

```
main.o:     main.c
            cc -c -I/home/iwashi/include main.c

sub1.o:     sub1.c
            cc -c -I/home/iwashi/include sub1.c

...
```

全部の行に絶対パスで-Iオプションを付けました。これでも動くわけですがあまり格好いいとは言えません。それにもし違うところのファイルをインクルードしなければならなくなったときは全部書き直さなければなりません。これはちょっと大変です。そこでマクロを使います。マクロは別のところで定義できるシンボルで、実行時に該当する文字列に置き換えられます。依存関係を書く前に書いておきます。今の例では次のようになります。

```
INCLUDEMINE = -I/home/iwashi/include

...

main.o:     main.c
            cc -c $(INCLUDEMINE) main.c
...
```

INCLUDEMINEというマクロを勝手に作りました。定義は=を使ってやります。=の右辺が全部値になります。続けて空白をおいて別のオプションを書いても構いません。呼び出すときには$(に続けてマクロの名前を書いて最後に)を書きます。マクロの定義の中に別のマクロの参照を書いても構いません。また環境変数を書くことも出来ます。例え

ば通常HOMEという環境変数は自分のホームディレクトリを表わしています。そのことを使えば、

```
MYINCLUDE = -I$(HOME)/include
```

という風にも書けます。特にメイクファイルはプログラムを作る処方箋であるわけですからソースプログラムと同じくらい重要です。それを計算機毎に書き換えたりするのは好ましくありません。大きなプログラムのメイクファイルの場合別の計算機に持っていっても書き換えなくていいようにマクロを考えましょう。計算機毎にディレクトリ構造は違っているかも知れません。それは環境変数として定義してメイクファイルの中では参照するだけにします。例えば、環境変数で

```
% setenv PROJECTDIR /home1/exp/project
```

などと定義してmakefileには

```
PROJECTINCDIR = $(PROJECTDIR)/include
PROJECTLIBDIR = $(PROJECTDIR)/lib
CPPFLAGS = -I$(PROJECTINCDIR)
LDFLAGS = -L$(PROJECTLIBDIR) -lproj
```

などとします。この場合もしPROJECTDIR環境変数が定義されていないとその部分が無視されます。注意してください。例えばCPPFLAGSは-I/includeということになってしまいます。

　makeには他にも様々な機能が盛り込まれています。実例を先程のdefault.mkで見てみましょう。以下はdefault.mkの冒頭の一部分です。

```
# @(#)default.mk          1.21 SMI Copyright 1986

SUFFIXES = .o .c .c~ .cc .cc~ .s .s~ .S .S~ .ln .f .f~ .l .l~ \
     .mod .mod~ .sym .def .def~ .r .r~ .y .y~ .h .h~ .sh .sh~ \
     .cps .cps~
.SUFFIXES: $(SUFFIXES)

#    C language section.
CC=cc
```

```
CFLAGS=
CPPFLAGS=
LINT=lint
LINTFLAGS=
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) \
     -target $(TARGET_ARCH:-%=%) -c
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) \
     -target $(TARGET_ARCH:-%=%)
LINT.c=$(LINT) $(LINTFLAGS) $(CPPFLAGS) $(TARGET_ARCH)
.c:
     $(LINK.c) -o $@ $< $(LDLIBS)
.c.ln:
     $(LINT.c) $(OUTPUT_OPTION) -i $<
.c.o:
     $(COMPILE.c) $(OUTPUT_OPTION) $<
.c.a:
     $(COMPILE.c) -o $% $<
     $(AR) $(ARFLAGS) $@ $%
     $(RM) $%
```

ここで=はマクロ定義に使われています。$はマクロの呼出しです。２文字以上の名前を持ったマクロの呼び出しは$のあと()でくくる必要があります。.c.o:という標的がありますが、これが.cファイルから.oファイルを作る暗黙の規則です。そのコマンドに$<というマクロが使われています。これは暗黙の規則の中で使われるマクロで依存ファイルの名前として使われます。同様に$@は標的ファイル名、$*が標的のベース名(.cなどを取った残りの部分)などです。.SUFFIXES:という標的は特殊なもので依存部分にサフィックスとして使う文字を宣言します。.oや.cはここで宣言されているので暗黙の規則で使うことができます。

　この中でCFLAGSなどのマクロが空白のままになっています。自分のmakefileの中で例えば

```
     CFLAGS=-I~/include -L~/lib
```

などを宣言すれば暗黙の規則の中でもこの値が使われます。自分のmakefileの中で暗黙の規則を定義することももちろんできます。

4.6.デバッガーを使う。

プログラムがだんだん大きくなるとデバッグも難しくなっていきます。ソースにデバッグのための出力文を書いたりする程度でできるデバッグもありますが、デバッガーを使うほうがずっと効率的です。UNIXの標準デバッガーはdbxです。

dbxではCのソースコードレベルのデバッグができますが、プログラムが使っているシンボル(変数名や関数名など)をdbxに教えてあげるために、コンパイルの時に特別の手当てが必要です。それは具体的にいうとccに-gオプションを付けるということです。これまでに出てきた例ですと、

```
% cc -g -c main.c
```

といった風にやります。makeを使うならば、CFLAGSに-gを加えておいてもよいでしょう。

実行可能形式のファイルができるとデバッグです。起動は

```
% dbx test
```

でtestという実行可能ファイルをデバッグします。dbxだけでデバッガーに入って中で

```
(dbx) debug test
```

でもできます。ここで(dbx)はdbxの出すプロンプトです。dbxは対話型プログラムでデバッグのための様々な機能を持っています。helpコマンドがdbxの中で使えます。

dbxはモジュール単位のデバッグを行ないます。モジュールは個別にコンパイルされた各々です。よく使う命令をあげておきます。詳しくはman dbxやdbxのhelpで見てください。

debug	あらたなデバッグセッションの開始
quit	dbxの終了
run	実行
rerun	再実行
stop	実行中断場所や変数、条件の設定
step	一行ずつ実行。関数呼び出しでその関数にはいる。

next	一行ずつ実行。関数呼び出しでその関数に入らない。
trace	一行ずつ表示しながら実行する。
display	変数などをそのつど表示させる。
print	変数の値を表示する。
dump	すべての局所変数を表示する。
modules	デバッグ対象のモジュールのリストを表示。
file	現在の対象ソースファイルの切り替え/表示。
func	現在の対象関数の切り替え/表示。
list	ソースファイルの内容を表示。
help	ヘルプ
where	現在のスタックの内容を表示
up	スタックの内容に従って親の関数へ切り替え
down	スタックの内容にしたがって子の関数へ切り替え

ほかにも色々命令が用意されていますが、これだけ知っているだけでも十分仕事になると思います。デバッグしたい関数をfileやfuncで選んでlistして行番号を見てstopでブレークポイントを設定してrunします。printなどで変数の内容を確かめながらstepやnextでじわじわと進んでバグを見つけます。具体的に見ていきましょう。まずdbxを起動します。

```
% dbx a.out
Reading symbolic information...
Read 521 symbols
(dbx)
```

ここでまずソースを表示させましょう。

```
(dbx) list main
   14   int      argc;
   15   char     **argv;
   16   {
   17            int i;
   18
   19            for( i = 0; i < argc; i++ )
   20            {
(dbx)
```

list mainは関数mainを表示します。多分途中までしか表示されません。続けてlistと入力します。次の行から表示されます。

```
   21                          ana_arg( *argv );
   22                          argv++;
   23              }
   24      }
(dbx)
```

デバッグが必要な関数はana_argです。表示させてみましょう。

```
(dbx) list ana_arg
   26   int        ana_arg( a )
   27   char       *a;
   28   {
   29              int             i;
   30              char            *p;
   31              printf( "Argument = %s ", *a );
```

まずこのプログラムを実行して見ます。mainの19行目で止めます。

```
(dbx) stop at 19
(7) stop at "/home/iwashi/src/myprog.c":19
```

stop atでブレークポイントを設定しました。(7)はコマンドナンバーです。このstopに与えられました。statusで見てみましょう。

```
(dbx) status
(7) stop at "/home/iwashi/src/myprog.c":19
(dbx) delete 7
(dbx) status
(dbx) stop at 19
```

コマンドナンバー7をdeleteで削除しました。その結果stautsは空になりました。ブレークポイントは解除されたことになります。話を続けるためにもう一度stopしておきま

す。次にrunで実行します。ブレークポイントにたどり着いて止りました。ここで変数の中身を見てみましょう。

```
(dbx) run
Running: a.out
stopped in main at line 19 in file "myprog.c"
   19               for( i = 0; i < argc; i++ )
(dbx) print argc
argc = 1
(dbx)
```

ポインターの場合print *argvなどと指定します。cの参照がそのまま使えます。次にここから一行ずつ順に実行させます。stepです。

```
(dbx) step
stopped in main at line 21 in file "myprog.c"
   21               ana_arg( *argv );
(dbx) step
```

ここでもう一度stepすると関数ana_argに入ります。nextとすると関数に入らず次の行に進みます。contで実行再開します。rerunで再度走り直しします。runやrerunには引数を渡せます。この例ではargcやargvに反映します。

　ある程度デバッグが進んだプログラムを走らせていると、通常はうまく走っているのだけれどたまに例えば次のようなメッセージを出力してこけてしまうことがあります。

```
% badprg
Arithmetic exception (core dumped)
%
```

いわゆるコアを吐くというやつです。実際にcoreというファイルが出来ているはずです。演算例外が発生しました。ゼロで割り算をしたのかも知れません。他にSegmentation faultなども起こります。Bus errorと表示される場合もあります。とんでもないところを参照した場合に発生します。このcoreとは大変貴重なものです。あまり嫌わないでください。死後診断(postmortem debugging)がこのcoreファイルを使って出来ます。

```
% dbx badprg core
```

```
...
program terminated by signal FPE (integer diveide by zero)
(dbx) where
zero_divide( ) at 0xef77a89c
sub3(), line 19 in "badprg.c"
sub2(), line 12 in "badprg.c"
sub1(), line 8 in "badprg.c"
main( argc = 1, argv = -268436644), line 4 in "badprg.c"
(dbx)
```

coreはプログラムの死体です。どこでどのように死んだかがわかります。whereで現在のスタックの内容が表示されます。見たところbadprg.cファイルの19行目、sub3という関数の中でゼロによる割り算が起こったようです。まずfuncコマンドでsub3にフォーカスします。

```
(dbx) func sub3
```

printでsub3の中の変数を表示させてみましょう。色々な手掛かりが得られるはずです。スタックの中を上に移動します。sub3はsub2から呼び出されているのでsub2にフォーカスが移ります。

```
(dbx) up
Current function is sub2
   12               sub3( );
(dbx)
```

同じようにdownで下に下がります。

　デバッグの必要によっては16進数表現で表示したい場合があります。整数は通常10進数として出力されてしまいます。そこで

```
(dbx) print i
‘myprog‘ana_arg‘i = 196608
(dbx) &i/X
0xeffffa5c:  0x00030000
(dbx)
```

このようにprint iだと大きな数が表示されてよくわからない場合も16進でかくと「あらなんだ」ということもあります。上の例ではiのアドレスを&iで指定してその内容を/Xで表示させました。/Oで8進数、/Fで浮動少数点数が同じように表示できます。