Linux C言語プログラミング

1．プログラミング上の注意点

1.1　スタックとヒープ

　メモリ上にデータを確保する方法にはいくつかの種類がある．プログラムの内部にあらかじめ確保される領域だけでなく，auto変数の実体として関数の実行時にそれぞれ確保されるスタックや，malloc()関数などで動的にデータを確保するヒープ領域がこれにあたる．

　バッファオーバーフロー攻撃と呼ばれるセキュリティホールに対する攻撃がある．これは，スタック上に確保される関数の戻り先を書き換えてしまうことで任意のコードを実行させようという攻撃である．メモリにデータがどのように配置されるかを意識しておくことが必要である．

1.2　メモリ上のデータ

　32ビット，64ビットといったCPUのデータバス幅に対して，これらのデータがメモリ上にどのように並べられているかを意識してプログラムを記述すべきである．

2．Cプログラムのビルド工程

2.1　コンパイル

　gccは，C言語プログラムのコンパイラとして有名である．正確には，gccは，「コンパイラ・ドライバ」と呼ばれる一連のプログラム・ビルド手順を順次実行するためのコマンドである．gccでC言語プログラムをコンパイルする手順は以下の通りである．

1. プリプロセッサによるファイルのインクルードやマクロの展開
2. アセンブラへの展開とアセンブル作業によるオブジェクトコードへの変換
3. オブジェクト・ファイルの結合やライブラリとのリンクを実施

　実行形式のファイルを，’list01.c’をコンパイルして生成するコマンドをリスト2.1に示す．

リスト2.1　実行形式ファイル生成

|  |
| --- |
| $ gcc list01.c –o list01 |

　ここで，-oオプションを指定しなければ，実行形式ファイルの名称はa.outとなる．

表2.1　gccオプション

|  |  |
| --- | --- |
| オプション | 意味 |
| -o | 実行形式ファイルの名前をつける |
| -c | オブジェクトファイル（.o）生成まで行う |
| -S | アセンブラファイル（.s）生成まで行う |

2.2　ファイルの分割

　複数のソースファイルをコンパイルして，別のファイルの関数を呼び出すことも可能である．

リスト2.2　実行形式ファイル生成（複数ソース）

|  |
| --- |
| $ gcc list01A.c list01B.c –o list01 |

　たとえば，list01.c，list01A.c，list01B.cの内容をリスト2.3，2.4，2.5としたとき，list01の実行結果（リスト2.6）は同じになる．

リスト2.3　list01.c

|  |
| --- |
| #include <stdio.h>  void function1 () {  printf (“Hello from function No.1\n”);  }  void function2 () {  printf (“Hello from function No.2\n”);  }  int main () {  function1 ();  function2 ();  return 0;  } |

リスト2.4　list01A.c

|  |
| --- |
| #include <stdio.h>  void function1 () {  printf (“Hello from function No.1\n”);  } |

リスト2.5　list01B.c

|  |
| --- |
| #include <stdio.h>  void function2 () {  printf (“Hello from function No.2\n”);  }  int main () {  function1 ();  function2 ();  return 0;  } |

リスト2.6　実行結果

|  |
| --- |
| $ ./list01  Hello from function No.1  Hello from function No.2 |

2.3　リンク

　分割したそれぞれのソースプログラムは，個別にオブジェクトファイルへコンパイルすることができる．コマンドの例をリスト2.7に示す．

リスト2.7　分割したソースのオブジェクトファイルへのコンパイル

|  |
| --- |
| $ gcc list01A.c list01B.c –c  $ ls  list01A.c list01A.o list01B.c list01B.o |

　これらのオブジェクトファイルを別途リンクして，実行形式を作成することも可能である．コマンドの例をリスト2.8に示す．

リスト2.8　オブジェクトファイルのリンク

|  |
| --- |
| $ gcc list01A.o list01B.o –o list01 |

　分割コンパイルには，プログラムの一部分を修正したときには，その影響がおよぶ範囲だけ再コンパイル，リンクすればよいというメリットがある．

2.4　ビルドの自動化

　Cで書かれたプログラムをコンパイルして実行形式までビルドし，さらに関連するリソースを適切に配置したり，環境に合わせて設定ファイルを作成したりといった作業は，コマンドを組み合わせて実行する．コマンドを記述したファイルを実行することでビルドを自動化するファイルとしてシェルスクリプトがあり，ビルドを自動化するツールとしてmakeツールがある．

2.5　シェルスクリプト

　シェルスクリプトは，コマンドをファイルに記述し，そのファイルを実行することで，記述されたコマンド列を実行する仕組みである．2.3節で述べた分割コンパイルの手順をそのまま並べたシェルスクリプトをリスト2.9に示す．

リスト2.9　分割コンパイルを行うシェルスクリプト

|  |
| --- |
| #!/usr/bin/sh  gcc list01A.c –c  gcc list01B.c –c  gcc list01A.o list01B.o –o list01 |

　リスト2.9のファイルをlist01.shとすると，このファイルに実行属性を与え，ソースコードの置かれているディレクトリで起動することで自動ビルドを行うことができる．その例をリスト2.10に示す．

リスト2.10　シェルスクリプトの実行

|  |
| --- |
| $ chmod +x list01.sh  $ ./list01.sh |

　シェルスクリプトは，分割コンパイルのメリットであった，修正時の再コンパイル効率が良いことを損ねている．

2.6　make

　2.6.1　Makefile

　ビルドの自動化を実現し，元データとそこから生成されるファイルの依存関係を把握し，さらに各ファイルのタイムスタンプから生成物を作り直す必要があるかどうかを判断する機能を備えているツールである．

　「元データから生成物が作成される」というルールを，’Makefile’という名前のファイルに記述する．’make –f *filename*’のように，-fオプションに続けて指定することで，他の名前をもつファイルをメイクファイルとして指定することもできる．

　メイクファイルにおいては，元データと生成物の関係を，「生成物：元データ」というようにコロンで結んで記述する．1つの生成物に対して複数の元データを関連付けることができる．また，その変換手段をその次の行に記述する．ただし，この変換手段を記載する行は「タブ」による空白のインデントで始めなければならない．2.5節と同じケースをメイクファイルで書き直すと，リスト2.11のようになる．

リスト2.11　Makefileの例

|  |
| --- |
| list01: list01A.o list01B.o  gcc list01A.o list01B.o -o list01  list01A.o: list01A.c  gcc -c -o list01A.o list01A.c  list01B.o: list01B.c  gcc -c -o list01B.o list01B.c |

Makefile内の処理を実行するにはmakeコマンドを用いる．リスト2.12に例を示す．

リスト2.12　make

|  |
| --- |
| $ make  gcc –c –o list01A.o list01A.c  gcc -c -o list01A.o list01A.c  gcc -c -o list01B.o list01B.c |

　2.6.2　マクロと擬似変数

　マクロは値を置き換える機能で，簡単に言えばMakefile内での定数である．擬似変数は，定義されたマクロを用いる変数である．リスト2.11をマクロと擬似変数を用いて書き直したものをリスト2.13に示す．

リスト2.13　マクロと擬似変数の例

|  |
| --- |
| CC = gcc  TARGET = list01  OBJS = list01A.o list01B.o  $(TARGET): $(OBJS)  $(CC) $^ -o $@ |

　Makefileでよく利用される擬似変数を表2.2に示す．

表2.2　Makefileで利用される擬似変数

|  |  |
| --- | --- |
| 擬似変数 | 説明 |
| $@ | ターゲットファイル |
| $^ | ターゲットが依存するすべてのファイル |
| $< | 依存ファイルのうち最初のファイル |
| $? | ターゲットより新しいすべての依存するファイル |
| $+ | Makefileと同じ順番の依存ファイル |
| $\* | ターゲットの名前から拡張子を除いたもの |

3．ライブラリ

3.1　さまざまなライブラリとAPIの利用

　ライブラリを活用したプログラミングを行いたいときには，そのライブラリ本体のパッケージだけでなく，そのライブラリに対応した開発パッケージ（-devel）もインストールすべきである．C言語によるライブラリの開発パッケージには，減っだファイルが必ず含まれている．

　ライブラリには，科学技術計算からデータベースの利用，ユーザインタフェースやさまざまなデバイスへのアクセスなど，多種多様なものが存在する．これらのライブラリが提供する機能はAPI（Application Program Interface）というインタフェースを利用して使うことができる．

3.2　ライブラリとのリンク

　ライブラリに依存しているプログラムは，システムにそのライブラリがインストールされていなければ正しく動作しない．実行時には，ライブラリが提供するコードと結合（リンク）されて初めて，正しいプログラムのコードとなり，プログラマの意図した通りに動作する．

　オブジェクトコードとライブラリの結合方法には，ライブラリのコードをプログラムのビルド時に実行形式に組み込む静的リンクライブラリと，共有ライブラリの2つがある．また，共有ライブラリは，実行形式のメモリへのロード時にリンクされる動的リンクライブラリと，必要に応じてメモリへロード，リンクされる動的ローディングの2種類に分かれる．

3.3　ヘッダファイル

　多様なライブラリが提供する機能をCのプログラムで利用するためには，そのライブラリが提供する関数や構造体の情報を知る必要がある．すなわち，提供される関数の引数としてなにを与え，またそれらはどのような型でなければならないか，その関数が戻す返り値の型や意味はなにか，あるいはライブラリで利用する構造体の内部構造はどうなっているかなどの情報がわからなければ，ライブラリが提供する機能を活用することはできない．それらの情報が記載されているファイルがヘッダファイルである．

　ヘッダファイルは，#includeディレクティブを用いて取り込まれる．リスト3.1にcファイルの例を，リスト3.2にヘッダファイルの例を示す．cファイルではなにもしないメイン関数が定義されており，ヘッダファイルでも関数function1()のプロトタイプ宣言を行うだけである．しかし実際にこの関数は使われていないため，エラーにはならない．

リスト3.1　list02.c

|  |
| --- |
| #include “list02.h”  int main () {  return 0;  } |

リスト3.2　list02.h

|  |
| --- |
| extern void function1(); |

　list02.cの実行形式ファイルを作成しても意味がないので，ビルド工程1「プリプロセッサによるインクルードやマクロの展開」を見てみよう．リスト3.3に，ビルド工程1の段階で中断し，その結果を標準出力に吐き出して処理を終えるコマンドと実行結果を示す．

リスト3.3　プリプロセッサの動作(1)

|  |
| --- |
| [tomoki@localhost ctests]$ gcc -E list02.c  # 1 "list02.c"  # 1 "<組み込み>"  # 1 "<コマンドライン>"  # 1 "/usr/include/stdc-predef.h" 1 3 4  # 1 "<コマンドライン>" 2  # 1 "list02.c"  # 1 "list02.h" 1  extern void function1();  # 2 "list02.c" 2  int main () {  return 0;  } |

　#includeディレクティブに指定するファイル名は，””で囲むか，<>で囲むかの2種類の方法で指定することができる．ダブルクォーテーションによる指定では，そのソースファイルが存在するディレクトリを先に探す．一方の<>という指定では，システムが用意したディレクトリを先に探す．分割コンパイルのために用意したヘッダファイルを扱うときは前者，ライブラリを利用するためにそのライブラリ向けに用意されたヘッダファイルを扱うときには後者の形式で記述することが一般的である．

　ヘッダファイルは，コンパイラの型チェックの辞書として使うこともできる．リスト3.3で見たように，インクルードされたヘッダファイルは，ビルド時にcファイルへコードが埋め込まれるため，ヘッダファイル内で，cファイルで使用する関数のプロトタイプ宣言をすることができる．extern宣言は，このプログラムの外に定義されていることを明示する宣言である．リスト3.2は，関数のプロトタイプ宣言をしているヘッダファイルの例である．

3.4　外部コードの利用

　標準ライブラリに含まれているprintf()が，実際にどのように呼び出されているかを見てみよう．リスト3.4に，「Hello World!」と出力するプログラムを示す．このプログラムをアセンブラのコードに変換するコマンドと，その実行結果をリスト3.5に示す．

リスト3.4　list03.c

|  |
| --- |
| #include <stdio.h>  int main () {  printf ("Hello World!");  return 0;  } |

リスト3.5　list03.s

|  |
| --- |
| [tomoki@localhost ctests]$ gcc -S -fno-builtin list03.c  [tomoki@localhost ctests]$ ls  list01A.c list01B.c list02.c list02.h list03.c list03.s  [tomoki@localhost ctests]$ cat list03.s  .file "list03.c"  .section .rodata  .LC0:  .string "Hello World!"  .text  .globl main  .type main, @function  main:  .LFB0:  .cfi\_startproc  pushq %rbp  .cfi\_def\_cfa\_offset 16  .cfi\_offset 6, -16  movq %rsp, %rbp  .cfi\_def\_cfa\_register 6  movl $.LC0, %edi  movl $0, %eax  call printf  movl $0, %eax  popq %rbp  .cfi\_def\_cfa 7, 8  ret  .cfi\_endproc  .LFE0:  .size main, .-main  .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-4)"  .section .note.GNU-stack,"",@progbits |

　「call printf」の行でprintf()が呼び出されているのがわかる．この段階のコードを機械語に変換することで，list03.oがオブジェクトコードとして作成される．実際に呼び出されてプログラムの一部となるコードは，ライブラリとしてシステムに格納されているコードである．呼び出されるそれらのコードとlist03.oのコードを結びつける仕事をするコマンドがリンカ，ldである．通常はコンパイラ・ドライバgccから呼び出されるため目にすることは少ないが，この過程を経てはじめて実行形式が用意されるため，極めて重要な工程である．

3.5　静的リンク

　静的リンクライブラリは，すでにコンパイルされたオブジェクトコードを再利用できる形にまとめたものである．慣習的に，拡張子として.aが付けられている．Linuxを含む一般的なUnix系のOSでは/libや/usr/lib，/usr/local/libといったディレクトリに格納される．コマンドで実際にリストアップすると，リスト3.6のような結果が得られる．

リスト3.6　静的リンクライブラリの表示

|  |
| --- |
| [tomoki@localhost ctests]$ ls -l /usr/lib64/lib\*.a  -rw-r--r--. 1 root root 1338 8月 3 02:46 /usr/lib64/libbsd-compat.a  lrwxrwxrwx. 1 root root 15 10月 7 20:25 /usr/lib64/libbsd.a -> libbsd-compat.a  -rw-r--r--. 1 root root 22846 8月 3 02:46 /usr/lib64/libc\_nonshared.a  -rw-r--r--. 1 root root 1338 8月 3 02:46 /usr/lib64/libg.a  -rw-r--r--. 1 root root 2664 8月 3 02:15 /usr/lib64/libieee.a  -rw-r--r--. 1 root root 11336 8月 3 02:15 /usr/lib64/libmcheck.a  -rw-r--r--. 1 root root 1780 8月 3 02:46 /usr/lib64/libpthread\_nonshared.a  -rw-r--r--. 1 root root 54406 8月 3 02:46 /usr/lib64/librpcsvc.a |

　静的リンクライブラリの利用は非常にシンプルである．静的リンクのメリットとして，一度実行形式ができあがってしまえば，そのアプリケーションの動作は比較的安定したものとなることが保証される点も指摘できる．共有ライブラリのような動的リンク形態では，実行時になるまで呼び出し先のコードが確定しない．これは，次に実行すること期には動作が変わっている可能性があるということを意味する．すなわち，ビルド時にライブラリのコードを実行形式に組み込んでしまう静的リンクであれば，万が一ライブラリが削除されたとしても，すでに必要なコードは組み込まれているため問題ない．

　静的リンクは，実行形式にライブラリのコードを組み込んでしまう点がデメリットでもある．生成される実行形式のサイズが大きなものになってしまうからである．

　gccのデフォルト設定では，静的リンクでなく動的リンクが用いられている．gccで静的リンクを利用するには，ビルド時に-staticオプションを付ける．

リスト3.7　静的リンク

|  |
| --- |
| [tomoki@localhost ctests]$ gcc list03.c -o list03static -static  /usr/bin/ld: -lc が見つかりません  collect2: エラー: ld はステータス 1 で終了しました |

　libc.aが存在しないためエラーが返ってきた．

3.6　動的リンク

　共有ライブラリは，さまざまなプログラムから同時に利用されるようにデザインされたライブラリで，シェアードライブラリとも呼ばれる．共有ライブラリは，”Shared Object”の頭文字から.soという拡張子のファイル名で格納されている．

　共有ライブラリを利用する方式では，実行形式がビルドされた段階で呼び出し先のアドレスが確定していない．プログラムが実行される直前に，ld.soあるいはld-linux.soという動的リンカによってライブラリへリンクされ，実行可能になる．これを，静的リンクに対して，動的リンクという．

3.7　動的ローディング

　共有ライブラリの一般的な利用手順においては，プログラムがロードされた直後に，動的リンカによって呼び出し関係の関係が図られる．一方の動的ローディングでは，プログラムがロードされ，コードの動作が開始した後でも一部のコードについてはリンクが完結していない場合がありえる．動的ローディングによる外部関数の利用は次の手順で実現できる．

1. dlopen()で対象とするオブジェクトファイルをオープンする
2. dlsym()でその中から実行したい関数へのポインタを取得する
3. その関数ポインタで示されるコードを呼び出す

　動的ローディングをするプログラムをリスト3.8に示す．

リスト3.8　list04.c

|  |
| --- |
| #include <dlfcn.h>  int main () {  // オブジェクトファイルをオープン  void\* handle = dlopen ("./list04.so", RTLD\_LAZY);  // 関数"function"へのポインタを取得  void (\*func)(void) = dlsym (handle, "function");  // functionを実行  (\*func) ();  // オブジェクトファイルをクローズ  dlclose (handle);  return 0;  } |

　リスト3.9に共有ライブラリのソースコードを示し，リスト3.10に共有ライブラリの作成方法を示す．リロケータブルなコードを生成する-fPICオプションを付けること，および共有ライブラリとしてリンクする-sharedオプションを付けることで共有ライブラリとする．

リスト3.9　list04dl.c

|  |
| --- |
| #include <stdio.h>  void function () {  printf ("Hello World!\n");  } |

リスト3.10

|  |
| --- |
| [tomoki@localhost ctests]$ gcc list04dl.c -fPIC -shared -o list04.so  [tomoki@localhost ctests]$ ls  list01A.c list02.c list03.c list04.c list04dl.c  list01B.c list02.h list03.s list04.so |

　リスト3.11に，動的ローディングをするプログラムの実行結果を示す．

リスト3.11

|  |
| --- |
| [tomoki@localhost ctests]$ gcc list04.c -o list04 -ldl  [tomoki@localhost ctests]$ ls  list01A.c list02.c list03.c list04 list04.so  list01B.c list02.h list03.s list04.c list04dl.c  [tomoki@localhost ctests]$ ./list04  Hello World! |

　list04dl.cのコードを書き換えて，再度コンパイルしてみても，その変更が実行結果に反映される．このように，動的ローディングはプログラムに大きな柔軟性を与える．この動的ローディング機構を上手に活用すると，プラグインやアドオンと呼ばれる拡張機能を利用して，機能を後から追加できるようにアプリケーションを設計することが可能である．本体プログラムを再コンパイルすることなく，新しい機能を備えたモジュールを動的にロード，リンクすることで拡張機能の追加が可能となる．

3.8　ライブラリを見つけたら

　ライブラリをインストールしたとき，まず使い方を確認する．よくできたライブラリであれば，しっかりしたマニュアルだけでなくサンプルプログラムも用意されているだろう．不幸にもそのような関連文書が整備されていない場合は，ライブラリのソースコードにあたってみればすべて解決する．その場合，まずヘッダファイルを辞書代わりに用いる．これらを頼りにすれば，ライブラリの使い方を理解することができるはずである．

3.9　APIの変更

　Cのプログラムからライブラリの関数を呼び出して使うときには，APIが適合していなければならない．ライブラリに互換性のない変更が加えられれば，プログラムに変更を加えていなかったとしても正しくコンパイルできなくなることがある．そこで，実行時にリンクする共有ライブラリで後方互換性を確保するために，sonameという仕組みが用意されている．ライブラリをアップデートしたときにsonameの仕組みを活かして整合性を保つには，管理者権限でldconfigを実行する必要がある．利用しているライブラリをアップデートしたときには，こまめにチェックする習慣を付けよう．

4．オプション解析

4.1　コマンドライン・オプション

　一般に，オプションを含むコマンドライン引数を与えることで，操作対象のデータを指定したり，動作条件を変化させたり，アプリケーションに対してさまざまな指示を与えることが可能である．ターミナルエミュレータやCUI（Character User Interface）端末上のシェルを用い，コンピュータと対話的な操作を行っている場合は，コンピュータへの操作はプロンプトに投入するコマンドとして文字列で明示的な命令を与える．gccやmakeなどがそれにあたる．GUIの環境においてはより直感的な操作が採用されているが，これらの環境でも，アプリケーション側の起動インタフェースが変わらない限りは，内部でコマンドライン操作と同様の起動操作が行われ続けるだろう．

4.2　コマンド引数の解析

　main()関数は，2つの引数をとる．2つの引数とは，そのプログラムが起動されたときのコマンドライン文字列に関する情報を得るための変数である．それらの引数は，慣習的にargc（argument count）とargv（argument vector）が用いられる．コマンドラインから引数を取得するプログラムをリスト4.1に示す．リスト4.2はその実行例である．

リスト4.1　list05.c

|  |
| --- |
| #include <stdio.h>  int main (int argc, char\* argv[]) {  int i;  for (i = 0; i < argc; i++) {  printf ("%d : %s\n", i, argv[i]);  }  return 0;  } |

リスト4.2　コマンド引数の表示

|  |
| --- |
| [tomoki@localhost ctests]$ gcc list05.c -o list05  [tomoki@localhost ctests]$ ./list05 foo bar baz  0 : ./list05  1 : foo  2 : bar  3 : baz |

　引数argcは，コマンドライン引数に並べられた単語の数をとり，argvにはそれぞれの単語へのポインタが格納された配列へのポインタが代入される．argv[0]には，実行ファイル名が代入されていることがわかる．

　実は，main()関数には3つ目の引数がある．それは，環境変数である．リスト4.3に，プログラム起動時にセットされている環境変数の中身を表示するプログラムを示す．また，その実行結果をリスト4.4に示す．

リスト4.3　list06.c

|  |
| --- |
| #include <stdio.h>  int main (int argc, char\* argv[], char\* envp[]) {  int i = 0;  while (envp[i] != NULL) {  printf ("%d : %s\n", i, envp[i]);  i++;  }  return 0;  } |

リスト4.4　環境変数の表示

|  |
| --- |
| [tomoki@localhost ctests]$ gcc list06.c -o list06  [tomoki@localhost ctests]$ ./list06  0 : XDG\_VTNR=1  1 : SSH\_AGENT\_PID=9543  2 : XDG\_SESSION\_ID=2  3 : HOSTNAME=localhost.localdomain  4 : IMSETTINGS\_INTEGRATE\_DESKTOP=yes  5 : GPG\_AGENT\_INFO=/run/user/1000/keyring/gpg:0:1  6 : TERM=xterm-256color  7 : SHELL=/bin/bash  8 : XDG\_MENU\_PREFIX=gnome-  9 : VTE\_VERSION=3803  10 : HISTSIZE=1000  11 : WINDOWID=39845895  12 : QTDIR=/usr/lib64/qt-3.3  13 : QTINC=/usr/lib64/qt-3.3/include  14 : IMSETTINGS\_MODULE=IBus  15 : QT\_GRAPHICSSYSTEM\_CHECKED=1  16 : USER=tomoki  17 : LS\_COLORS=rs=0:di=38;5;27:ln=38;5;51:mh=44;38;5;15:pi=40;38;5;11:so=38;5;13:do=38;5;5:bd=48;5;232;38;5;11:cd=48;5;232;38;5;3:or=48;5;232;38;5;9:mi=05;48;5;232;38;5;15:su=48;5;196;38;5;15:sg=48;5;11;38;5;16:ca=48;5;196;38;5;226:tw=48;5;10;38;5;16:ow=48;5;10;38;5;21:st=48;5;21;38;5;15:ex=38;5;34:\*.tar=38;5;9:\*.tgz=38;5;9:\*.arc=38;5;9:\*.arj=38;5;9:\*.taz=38;5;9:\*.lha=38;5;9:\*.lz4=38;5;9:\*.lzh=38;5;9:\*.lzma=38;5;9:\*.tlz=38;5;9:\*.txz=38;5;9:\*.tzo=38;5;9:\*.t7z=38;5;9:\*.zip=38;5;9:\*.z=38;5;9:\*.Z=38;5;9:\*.dz=38;5;9:\*.gz=38;5;9:\*.lrz=38;5;9:\*.lz=38;5;9:\*.lzo=38;5;9:\*.xz=38;5;9:\*.bz2=38;5;9:\*.bz=38;5;9:\*.tbz=38;5;9:\*.tbz2=38;5;9:\*.tz=38;5;9:\*.deb=38;5;9:\*.rpm=38;5;9:\*.jar=38;5;9:\*.war=38;5;9:\*.ear=38;5;9:\*.sar=38;5;9:\*.rar=38;5;9:\*.alz=38;5;9:\*.ace=38;5;9:\*.zoo=38;5;9:\*.cpio=38;5;9:\*.7z=38;5;9:\*.rz=38;5;9:\*.cab=38;5;9:\*.jpg=38;5;13:\*.jpeg=38;5;13:\*.gif=38;5;13:\*.bmp=38;5;13:\*.pbm=38;5;13:\*.pgm=38;5;13:\*.ppm=38;5;13:\*.tga=38;5;13:\*.xbm=38;5;13:\*.xpm=38;5;13:\*.tif=38;5;13:\*.tiff=38;5;13:\*.png=38;5;13:\*.svg=38;5;13:\*.svgz=38;5;13:\*.mng=38;5;13:\*.pcx=38;5;13:\*.mov=38;5;13:\*.mpg=38;5;13:\*.mpeg=38;5;13:\*.m2v=38;5;13:\*.mkv=38;5;13:\*.webm=38;5;13:\*.ogm=38;5;13:\*.mp4=38;5;13:\*.m4v=38;5;13:\*.mp4v=38;5;13:\*.vob=38;5;13:\*.qt=38;5;13:\*.nuv=38;5;13:\*.wmv=38;5;13:\*.asf=38;5;13:\*.rm=38;5;13:\*.rmvb=38;5;13:\*.flc=38;5;13:\*.avi=38;5;13:\*.fli=38;5;13:\*.flv=38;5;13:\*.gl=38;5;13:\*.dl=38;5;13:\*.xcf=38;5;13:\*.xwd=38;5;13:\*.yuv=38;5;13:\*.cgm=38;5;13:\*.emf=38;5;13:\*.axv=38;5;13:\*.anx=38;5;13:\*.ogv=38;5;13:\*.ogx=38;5;13:\*.aac=38;5;45:\*.au=38;5;45:\*.flac=38;5;45:\*.mid=38;5;45:\*.midi=38;5;45:\*.mka=38;5;45:\*.mp3=38;5;45:\*.mpc=38;5;45:\*.ogg=38;5;45:\*.ra=38;5;45:\*.wav=38;5;45:\*.axa=38;5;45:\*.oga=38;5;45:\*.spx=38;5;45:\*.xspf=38;5;45:  18 : SSH\_AUTH\_SOCK=/run/user/1000/keyring/ssh  19 : SESSION\_MANAGER=local/unix:@/tmp/.ICE-unix/9429,unix/unix:/tmp/.ICE-unix/9429  20 : USERNAME=tomoki  21 : GNOME\_SHELL\_SESSION\_MODE=classic  22 : PATH=/usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin:/home/tomoki/.local/bin:/home/tomoki/bin  23 : MAIL=/var/spool/mail/tomoki  24 : DESKTOP\_SESSION=gnome-classic  25 : QT\_IM\_MODULE=ibus  26 : PWD=/home/tomoki/src/ctests  27 : XMODIFIERS=@im=ibus  28 : LANG=ja\_JP.UTF-8  29 : GDM\_LANG=ja\_JP.UTF-8  30 : GDMSESSION=gnome-classic  31 : HISTCONTROL=ignoredups  32 : XDG\_SEAT=seat0  33 : HOME=/home/tomoki  34 : SHLVL=2  35 : GNOME\_DESKTOP\_SESSION\_ID=this-is-deprecated  36 : XDG\_SESSION\_DESKTOP=gnome-classic  37 : LOGNAME=tomoki  38 : QTLIB=/usr/lib64/qt-3.3/lib  39 : DBUS\_SESSION\_BUS\_ADDRESS=unix:abstract=/tmp/dbus-Hf3yrLCayp,guid=b5e0bfe60e384ceae4587cdd57f83256  40 : LESSOPEN=||/usr/bin/lesspipe.sh %s  41 : WINDOWPATH=1  42 : XDG\_RUNTIME\_DIR=/run/user/1000  43 : DISPLAY=:0  44 : XDG\_CURRENT\_DESKTOP=GNOME-Classic:GNOME  45 : XAUTHORITY=/run/gdm/auth-for-tomoki-gBzlQY/database  46 : \_=./list06  47 : OLDPWD=/home/tomoki/src |

　プライベートが覗かれているみたいで恥ずかしい．

4.3　オプション解析

　Unixでは慣習的に，コマンドライン・オプションには-（ハイフン）で始まる文字列を使う．-hや-?，あるいは-help，--help，--usageといったオプションは，簡単なヘルプやコマンドの使い方を表示るうために使われる．また--versionというオプションを指定すると，そのコマンドのバージョン番号を表示するような実装が一般的な約束事とされている．

　コマンドライン・オプションの解析には，Linuxの標準ライブラリに含まれているgetopt()とgetopt\_long()を使う．

　ハイフンにつづけた英数記号1文字で表す形式のオプションを短いオプションという．gccでは，-cや-oが短いオプションに相当する．短いオプションには引数をとるものととらないものがあり，引数をとる場合には，オプション文字に続けて指定，ないしは　で区切って続けて並べることで指定する．

　--helpのように，ハイフン2個に続けて単語で表されるオプションを長いオプションという．長いオプションにも引数をとるものととらないものがあり，引数をとる場合には，--longopt=argumentのように，イコールで続けて指定することが慣習となっている．

　具体的な例として，短いオプション-b，-h，-nを解釈するプログラムをリスト4.5に示す．

リスト4.5　list07.c

|  |
| --- |
| #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  // usage関数  // 引数：ファイル名（実行コマンド）  // 返り値：なし  // 処理：ターミナルに，オプションの使い方を表示する  void usage (char\* cmd) {  fprintf (stderr, "usage: \n");  fprintf (stderr, " %s -h : show help\n", cmd);  fprintf (stderr, " %s -b : say good-bye\n", cmd);  fprintf (stderr, " %s -n <name> : greeting to <name>\n", cmd);  }  int main (int argc, char\* argv[]) {  int opt;  char\* greeting = "Hello, ";  char\* target = "world";  // 引数が取得される間  while ((opt = getopt (argc, argv, "bhn:")) != -1) {  switch (opt) {  // -bオプション  case 'b' :  greeting = "Good-bye, ";  break;  // -nオプション  case 'n' :  target = optarg;  break;  // -hオプションとそれ以外  case 'h' : /\* fall through \*/  default :  usage (argv[0]);  return EXIT\_FAILURE;  }  }  printf ("%s%s!\n", greeting, target);  return EXIT\_SUCCESS;  } |

　getopt()を使うために，ヘッダファイルunistd.hをインクルードした．ヘッダファイルstdlib.hは，EXIT\_FAILUREとEXIT\_SUCCESSを使うためにインクルードした．

　getopt()は，配列argv[]を順番に解析し，最後まで解析し終えたときに-1を返す．すなわち，オプションをすべて受け取ってから-1を返し，while文を終了する．-b，-h，-nの3種類のオプションをとるので，文字列”bhn:”を渡している．nの後のコロン:は，オプション-nが引数をとることを表す．

　ヘッダファイルstdlib.hでは，EXIT\_FAILUREとEXIT\_SUCCESSのマクロは「#define EXIT\_FAILURE 1」，「#define EXIT\_SUCCESS 0」として定義されている．

　list07の実行例をリスト4.6に示す．

リスト4.6　短いオプション解析

|  |
| --- |
| [tomoki@localhost ctests]$ gcc list07.c -o list07  [tomoki@localhost ctests]$ ./list07  Hello, world!  [tomoki@localhost ctests]$ ./list07 -h  usage:  ./list07 -h : show help  ./list07 -b : say good-bye  ./list07 -n <name> : greeting to <name>  [tomoki@localhost ctests]$ ./list07 -b  Good-bye, world!  [tomoki@localhost ctests]$ ./list07 -n  ./list07: option requires an argument -- 'n'  usage:  ./list07 -h : show help  ./list07 -b : say good-bye  ./list07 -n <name> : greeting to <name>  [tomoki@localhost ctests]$ ./list07 -n foo  Hello, foo!  [tomoki@localhost ctests]$ ./list07 -x  ./list07: invalid option -- 'x'  usage:  ./list07 -h : show help  ./list07 -b : say good-bye  ./list07 -n <name> : greeting to <name> |

　list07.cを改造して，同様の処理をするが，長いオプションを解析するプログラムを作成する．作成したプログラムをリスト4.7に示す．viエディタでのファイルの名前を付けて保存は，コマンドモードでの「:w list08.c」である．

リスト4.7　list08.c

|  |
| --- |
| #include <getopt.h>  #include <stdio.h>  #include <stdlib.h>  // usage関数  // 引数：ファイル名（実行コマンド）  // 返り値：なし  // 処理：ターミナルに，オプションの使い方を表示する  void usage (char\* cmd) {  fprintf (stderr, "usage: \n");  fprintf (stderr, " %s --help : show help\n", cmd);  fprintf (stderr, " %s --bye : say good-bye\n", cmd);  fprintf (stderr, " %s --name=<name> : greeting to <name>\n", cmd);  }  int main (int argc, char\* argv[]) {  int opt;  char\* greeting = "Hello, ";  char\* target = "world";  int option\_index = 0;  // option構造体  // getopt.hにおいて，以下のように定義されている  // struct option {  // const char \*name;  // int has\_arg;  // int \*flag;  // int val;  // };  // nameはオプション名  // has\_argは引数をもつかどうか  // flagはオプション引数の解析をした結果を代入する変数へのポインタ  // NULLを指定しておけば，その結果を返り値として返す  // valはそのオプションが指定されたときに返す値  struct option long\_options[] = {  {"bye", 0, NULL, 'b'},  {"help", 0, NULL, 'h'},  {"name", 1, NULL, 'n'},  {0, 0, 0, 0}  };  // 引数が取得される間  while ((opt = getopt\_long (argc, argv, "bhn:", long\_options, &option\_index)) != -1) {  switch (opt) {  // --byeオプション  case 'b' :  greeting = "Good-bye, ";  break;  // --nameオプション  case 'n' :  target = optarg;  break;  // --helpオプション  case 'h' : /\* fall through \*/  default :  usage (argv[0]);  return EXIT\_FAILURE;  }  }  printf ("%s%s!\n", greeting, target);  return EXIT\_SUCCESS;  } |

リスト4.8　長いオプション解析

|  |
| --- |
| [tomoki@localhost ctests]$ gcc list08.c -o list08  [tomoki@localhost ctests]$ ./list08  Hello, world!  [tomoki@localhost ctests]$ ./list08 --bye  Good-bye, world!  [tomoki@localhost ctests]$ ./list08 --name=bar  Hello, bar!  [tomoki@localhost ctests]$ ./list08 --name bar  Hello, bar!  [tomoki@localhost ctests]$ ./list08 --help  usage:  ./list08 --help : show help  ./list08 --bye : say good-bye  ./list08 --name=<name> : greeting to <name>  [tomoki@localhost ctests]$ ./list08 -b  Good-bye, world! |

5．テスト手法の紹介

・makeを活用したテストの自動実行

・リグレッションテスト

・CUnitの利用

6．GUIプログラミング

6.1　GUIについて

　現代のコンピュータ環境においては，グラフィクスを多用したGUI（Graphical User Interface）が一般的である．GTK+は，Unix環境でCによるGUIプログラムを構築する際のデファクトスタンダードとなっているライブラリであり，さまざまな種類のウィジットと操作手段を備えている．

　マルチタスクで複数のアプリケーションを同時に起動して処理を並行させることが当たり前となった現在，ときにはドラッグ＆ドロップといった操作によってアプリケーション間でデータをやり取りすることもある．その際，各アプリケーションでインタフェースの外観，ルック＆フィールが統一されていることが望ましい．そこで用意されている仕組みがGUI部品である．GUI部品のことをウィジットといい，それぞれのOSプラットフォームがプログラミングライブラリとして提供されることでアプリケーションのルック＆フィールに関する統一感を保つように工夫されている．一連のGUI部品を集めたライブラリを，ウィジットセット・ライブラリと呼ぶ．

　Linuxを含むUnix系のOSではGNOME（GNU Network Object Model Environment）およびKDE（K Desktop Environment）と呼ばれるデスクトップが2大勢力となっており，それぞれの環境でGTK+，QtというGUI部品が用意されている．なお，GKTK+はCのんプログラミングライブラリとして用意されている一方で，QtはC++のクラスライブラリとして提供されているという特徴がある．

6.2　GTK+

　もともと画像レタッチプログラムであるGIMP（GNU Image Manipulation Program）向けに作成されたパーツを，汎用的なGUI部品のライブラリとして再構成したウィジットセット・ライブラリがGTK+の起源である．GTK+の関連プロジェクトとして，表6.1に示すようなものがある．

表6.1　GTK+を指されるアーキテクチャ

|  |  |
| --- | --- |
| 名称 | 説明 |
| Glib | GTK+の基礎をなす低レベルのコアライブラリ |
| Pango | 国際化機能を備えたテキスト描画・配置ライブラリ |
| Cairo | 複数の出力形式をサポートする2次元グラフィクスライブラリ |
| ATK | アクセシビリティを配慮したインタフェースを提供するライブラリ |

　GTK+自体はCで書かれたライブラリだが，C++のクラスライブラリとして用意されたGFC（GTK+ Foundation Classes）も存在する．また，C#，Java，Python，JavaScript，Perl，Vala，R，Lua，Guile，Ruby，PHP，Ada，OCaml，Haskell，FreeBASIC，DなどからGTK+の機能を利用できるように，各種の言語バインディングも開発されている．

　GTK+は，Cによるライブラリでありながらも，徹底したオブジェクト指向で設計されている．

6.3　イベント駆動型プログラミング

　なんらかのイベントの発生に応じて，適宜処理を行うようなスタイルのプログラムを「イベント駆動型プログラミング」「イベントドリプン・プログラミング」という．イベント駆動型プログラミングの一般的な処理の手順は

　　1 さまざまな準備を行う

　　2 コールバック関数を登録する

　　3 メインループを起動する

　　4 イベントに応じて登録したコールバック関数が呼ばれる

という流れで行われる．

6.4　ウィジットのレイアウトとパッキング

　自由度の高いGUIアプリケーションにおいても，一般的なインタフェース構築の作法やガイドラインが定められている．メニューがウィンドウの最下行に用意されていたり，ドロップダウンリストをクリックするとアイテムが横に並んだりといった，通常ありえないような挙動を示すアプリケーションはユーザに受け入れられない．ウィジットの配置についても，プログラマの理屈ではなく，ユーザの利便性や常識を優先すべきである．

　現代的なGUIでは，ウィンドウの大きさを変えたときに，それに合わせてウィジットが適切にリサイズされ，配置されるような仕組みが用意されている．レイアウトマネージャやコンテナへのパッキングと呼ばれる機構を使い，ウィジットの配置に意味を与えてレイアウトするのである．設定すべきパラメータは増え，ウィジットを論理的に配置しなければならないため難しくはなるが，ユーザ操作の利便性を考えればレイアウトマネージャを使った「正しい」レイアウトを用意すべきである．