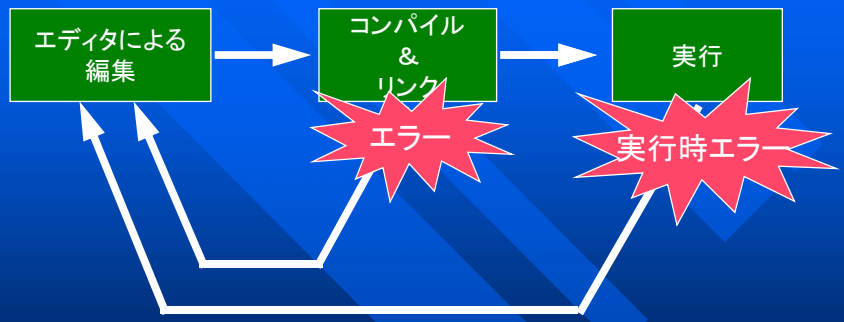
makeおよびデバッガの活用

上級プログラミング　講義資料

成蹊大学理工学部情報科学科

プログラムの作成から実行まで



分割コンパイル

1つのソースファイルで、それが大規模化していくと・・・・・・

デメリット

×正しい動作をする部分とそうでない部分が混合してしまい、
バグのある部分が特定しづらい。

×ごく一部のバグの修正にともなって、その他の正しい部分まで合わせて何度もコンパイルし直さなければならず、時間が無駄になる。

関連のあるモジュールごとに個別のファイルに分けて作成する。

例　：　ある範囲の角度における、sin,cos,tanの値を表示するプログラム

ファイル構成（/share/material/advpro/maketry/にある）

◎ myheader.h　：　関数のプロトタイプ宣言（ヘッダファイル）

◎ src1.cpp　：　main関数を含むソースファイル

◎ calsin.cpp　：　sinの値を表示する

◎ calcos.cpp　：　cosの値を表示する

◎ caltan.cpp　：　tanの値を表示する

myheader.h の内容

```
// myheader.h の記述

void sinlist(int, int, int);
void coslist(int, int, int);
void tanlist(int, int, int);
```


src1.cpp の内容

```
#include <iostream>
#include "myheader.h"
int main()
{
    int min, max, inc;

    cout << "角度の範囲(始値と終値)を入力-->";
    cin >> min >> max;
    cout << "角度の増分を入力-->";
    cin >> inc;

    sinlist(min, max, inc);
    coslist(min, max, inc);
    tanlist(min, max, inc);

    return 0;
}
```


calsin.cpp の内容

```
#include <iostream>
#include <iomanip>
#include <cmath>

void sinlist(int min, int max, int inc)
{
    int degree;

    cout << endl << "sin関数の値を一覧します.....¥n";
    for(degree=min; degree<=max; degree+=inc){
        cout << "sin(" << setw(3) << degree << ")="
             << sin(M_PI*degree/180) << endl;
    }
}
```

calcos.cpp, caltan.cppも、calsin.cppと同様なので省略

分割コンパイルの方法(手動)

1．それぞれのソースファイルをコンパイルする。

```
g++ src1.cpp   -c   -o src1.o
g++ calsin.cpp -c   -o calsin.o
g++ calcos.cpp -c   -o calcos.o
g++ caltan.cpp -c   -o caltan.o
```

ここは、省略可能

-c というオプションはソースコードのコンパイルのみで、リンクはしない。コンパイル済みのオブジェクトファイルはそれぞれ .o という拡張子のファイルに出力される。

エラーの出たファイルだけ編集し直せば良い。

２．すべてのオブジェクトファイルと必要なライブラリをリンクして実行ファイルを構築する。

```
g++ src1.o calsin.o calcos.o caltan.o -o src1
```

実行ファイルの出力は、-o オプションを用いて src1 と指定している。

もし実行時エラーが出たら、１．のうち、該当するソースファイルのみを修正・コンパイルし直し、上記の実行ファイル構築の作業を行う。

分割されたファイル数が多くなると、上記の手動方法では大変。

これを自動化するのがmakeの活用である。

makeコマンドの利用

分割コンパイルを自動化する。

ただし、makeにどのように実行ファイルを構築すれば良いかを指示するために Makefile という名前の指示ファイルを作成する必要がある。ちなみに　makeコマンドは、

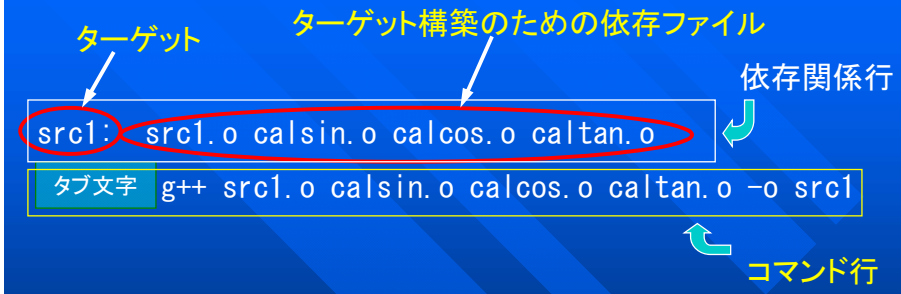
makefile
Makefile
s.makefile
s.Makefile
SCCS/s.makefile
SCCS/s.Makefile

の順で指示ファイルを探し、最初に見つかったファイルに従って、実行ファイルを構築してくれる。

Makefile　の記述例（テキストファイルとして作成）

```
src1:  src1.o calsin.o calcos.o caltan.o
タブ文字 g++ src1.o calsin.o calcos.o caltan.o -o src1
src1.o: src1.cpp myheader.h
タブ文字 g++ src1.cpp -c
calsin.o: calsin.cpp
タブ文字 g++ calsin.cpp -c
calcos.o: calcos.cpp
タブ文字 g++ calcos.cpp -c
caltan.o: caltan.cpp
タブ文字 g++ caltan.cpp -c
```


Makefile　の構造



ターゲットは、:の右側の項目に依存して作成される。

その作成方法は、次のコマンド行に示された通りに行う。このコマンド行は2行以上にわたることができるが、行頭はかならずタブ文字にする。

依存関係行とコマンド行のペアを**エントリ**という。

makeコマンドの実行

makeを実行するには単に

```
make
```

と入力するか、作成したMakefileのファイル名を指定して、

```
make -f Makefile
```
Makefile以外の名前のときに便利

のようにしても良い。実行すると各エントリのターゲットが依存関係により適切な順序で以下のように作成されていく。

```
g++ src1.cpp -c
g++ calsin.cpp -c
g++ calcos.cpp -c
g++ caltan.cpp -c
g++ src1.o calsin.o calcos.o caltan.o -o src1
```


ここで、例えばcalcos.cppのファイルを少し変更して保存し直してみよ。その後、makeコマンドを実行すると、以下のように実行される。

```
g++ calsin.cpp -c
g++ src1.o calsin.o calcos.o caltan.o -o src1
```

すなわち、makeコマンドは依存関係にあるファイルを調べ、**修正されて新しくなったファイルに依存するターゲットのみを再構築**してくれたのである。

もし、修正したソースファイル（例えばcalcos.cpp）のみをコンパイルし、エラーチェックしたければ、

```
make calcos.o
```

のようにそのターゲットをmakeコマンドに指定すればよい。

makeにより作成された .o ファイルや、最終的な実行ファイルを消すには

```
rm *.o src1
```

のようなコマンドで可能であるが、Makefileに次のようなエントリを加え、

```
clean:
タブ文字 rm  *.o  src1
```

そして

```
make clean
```

と、ターゲットにcleanを指定すれば、実行される。

コマンド行が2行以上連続する場合、各コマンド行ごとに新しいシェルでそのコマンドが実行されることに注意が必要である。

2つ以上のコマンドを;で区切って、1行以内に納めるか、行末にバックスラッシュ（＼）をつけて次行もその行の続き手あることを明示すると、それらのコマンドは同じシェル上で実行されることになる。

また、Makefile内では、#文字の後ろは行末までがコメント扱いとなる。

GNUデバッガ(gdb)の利用

- コンパイルではエラーは無いが、実行結果が間違っているようだ.....
- エラー箇所を調べるために、あちこちに出力文を挿入したが、どこが悪いのかはっきりと把握しづらい.....

デバッガを利用すると、あちこちに出力文を入れることなく、現在プログラム中のどの変数がどんな値になっているか、プログラムの流れは思った通りに動いているかなどが観察できる。

デバッガを利用するには.....

ソースコードのコンパイル時に、**-g** というオプションをつける。このオプションにより、デバッガによるプログラムの実行を行うと、ソースコードレベルでプログラムの実行を追跡することができる(ここでは、emacsとともに使用する)。

```
g++  src1.cpp -c -g
```

あるいは、Makefile中に -g オプションを使用するように手を加えることもできる。そのように修正したMakefileを次に示す。

デバッグ可能にするためのMakefileの例

```
# 簡単なMakefileの例

# 次の行はマクロ定義
CXXFLAGS = -g

# 以降はエントリ
src1:  src1.o calsin.o calcos.o caltan.o
        g++ src1.o calsin.o calcos.o caltan.o -o src1
src1.o: src1.cpp myheader.h
        g++ src1.cpp -c $(CXXFLAGS)
#                          ここでマクロ変数を利用している
calsin.o: calsin.cpp
        g++ calsin.cpp -c $(CXXFLAGS)
calcos.o: calcos.cpp
        g++ calcos.cpp -c $(CXXFLAGS)
caltan.o: caltan.cpp
        g++ caltan.cpp -c $(CXXFLAGS)
```

例えば上記を Makefile2 のファイル名で前のMakefileと区別して保存した場合、
`make -f Makefile2`　とすることでMakefile2を使ってmakeすることができる。

emacs上でのgdbの利用

① emacsを起動しておく。

② gdbを起動
ミニバッファで　**M-x gdb**　(M-xはESCキーを押してx)

③ 実行ファイル名を指定
Run gdb (like this): **gdb src1**　(先ほどの例の場合)
これで上部バッファに (gdb) というプロンプトが出る。

④ ソースファイルを確認
(gdb) **list**
ソースの続きを見るには list を再実行していく。単にリターンキーをたたくと、直前のgdbコマンドを繰り返す機能があるのでそれを使っても良い。

⑤ ブレークポイントを設定する。
ブレークポイントとは、プログラムを途中で止める位置のこと。
(gdb) **break 10**
これでソースコードの10行目を実行する直前で停止するように設定したことになる。必要に応じてブレークポイントをいくつか設定する。
設定したブレークポイント情報を確認するには
（gdb）**info break**
各ブレークポイントに行番号とは別に設定した順に番号がつくことに注意。
ブレークポイントを削除するには、info breakで確認した番号（行番号ではない）を使って、
（gdb）**delete 番号**
すべてのブレークポイントを消すには
（gdb）**clear**

⑥ 実行開始
（gdb）**run**
最初に遭遇するブレークポイントで止まる。ブレークポイントが1つも設定されていない場合は、通常通り、連続実行される。

⑦ 実行停止時の各種操作
1行ずつ実行　（gdb）**step**
行内に関数があれば、そこへジャンプし、step実行が続けられる。
1行ずつ実行　（gdb）**next**
次の行へ移るところまで現在行内は連続実行
現在の関数終了時まで連続実行　（gdb）**finish**
変数や式の値を確認　（gdb）**print 式**
その他の操作については、オンラインヘルプ参照
（gdb）**help コマンド名**
あるいは
M-x info を実行し、GDBの項目を参照する。

⑧ ウォッチポイントの設定
指定した変数または式の値が変更したときにプログラムを一時停止させる機能
（gdb）**watch 式**

⑨ 実行の再開
（gdb）**continue**

⑩ gdbの終了
（gdb）**quit**
gdbバッファも閉じる場合は、M-x kill を行う。