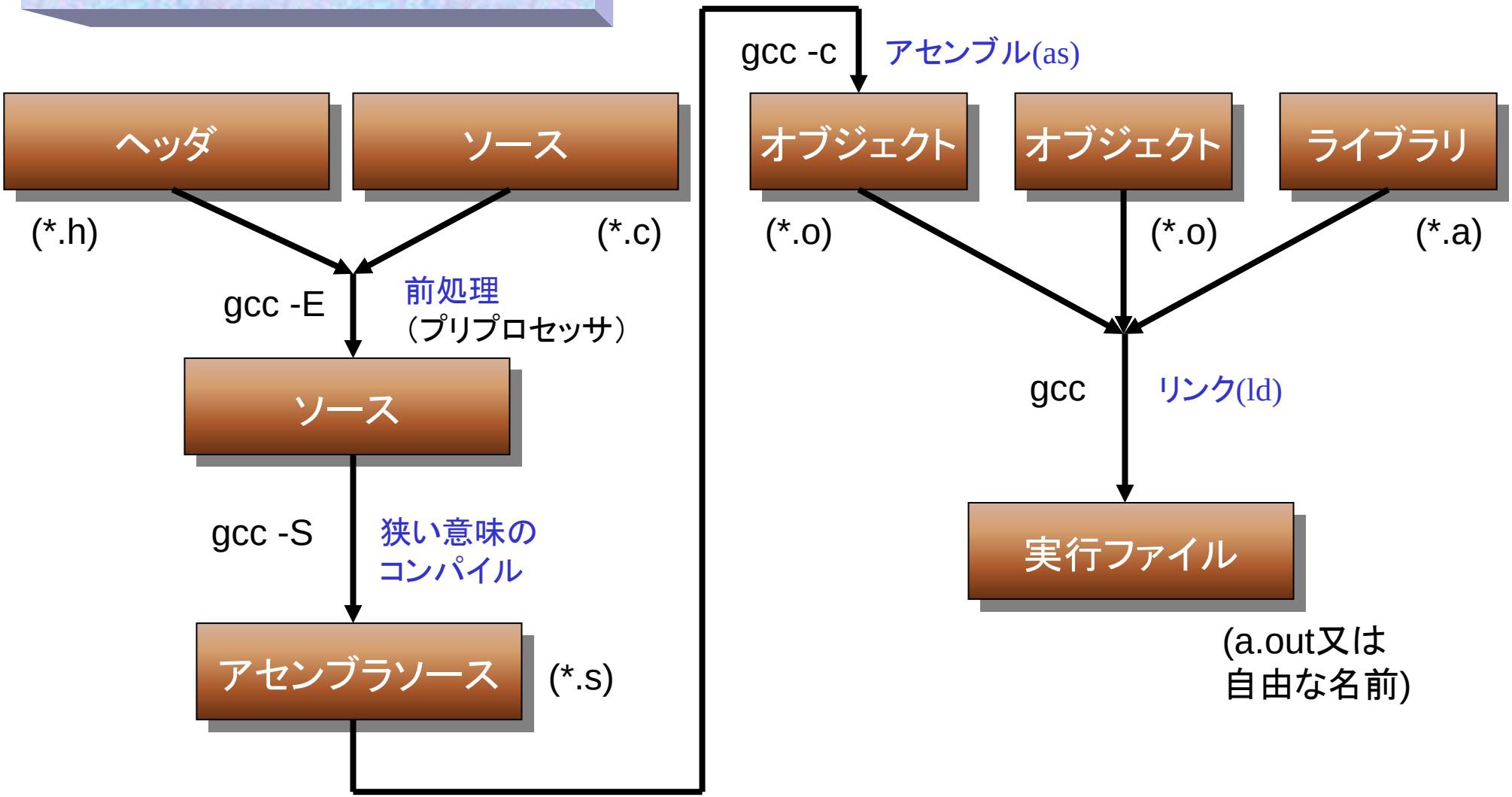
プログラミング1
第12回　コンパイル

- 条件演算子
- プリプロセッサ
- 分割コンパイル

ここにあるサンプルプログラムは
`/home/course/prog1/public_html/2007/HW/lec/sources/`
にあります。各自自分のディレクトリにコピーして、コンパイル・実行してみてください

コンパイルの流れ

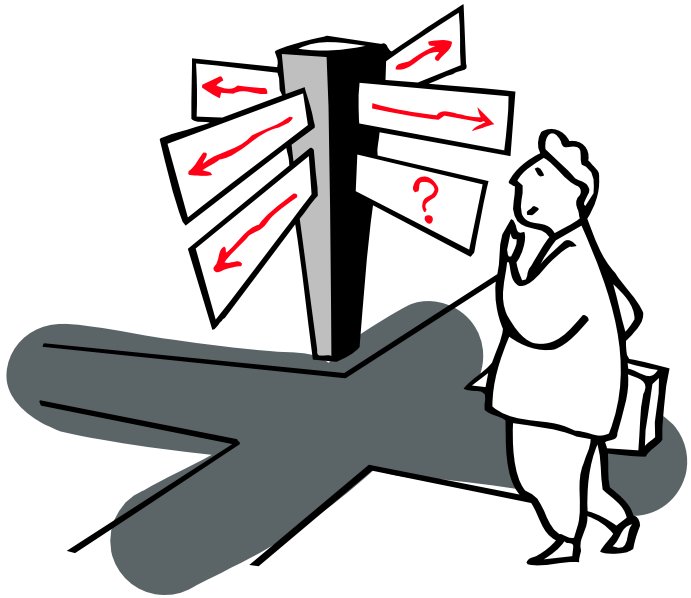

ヘッダ
ソース
(*.h)
(*.c)
gcc -E
前処理
（プリプロセッサ）
ソース
gcc -S
狭い意味の
コンパイル
アセンブラソース
(*.s)
gcc -c
アセンブル(as)
オブジェクト
オブジェクト
ライブラリ
(*.o)
(*.o)
(*.a)
gcc
リンク(ld)
実行ファイル
(a.out又は
自由な名前)

プリプロセッサ

- 本当のコンパイルの前に「前処理」を行う
- 以下のような制御文によって制御される
  - `#include`
  - `#define`
  - `#if / endif / else / elif`
  - `#if defined / if !defined / ifdef / ifndef`
- プリプロセッサの用途は以下のとおり
  - デバッグ時に補助情報を出力させるなど、コンパイル時の条件に応じて違う動作をさせる。
  - 1つのプログラムを複数のC処理系に対応させる(gccとccなど)
  - 複数のオペレーティングシステム(OS)に対応させる (Solarisと Windowsなど)
  - 分割コンパイル時の共通事項をインクルードファイルで共通化させる
  - 簡単な関数をマクロで作成する。

プリプロセッサ

- プリプロセッサに対する指示命令はC言語と異なり、以下のような規則がある
  - 区切りにセミコロンは書かない
  - 一つの命令は一行内に書かなければいけない(「＼」で終わる継続行の場合は複数でも良い)
  - １文字目は ＃ で始まらなければいけない



include

- **#include -- ファイルの展開**
  - **#include <stdio.h> と書くと、その位置にファイル stdio.h を読み込み、展開する**
    - **<>で囲んだ場合は/usr/include/下を探す（会津大の設定）**
    - **“ "で囲んだ場合はカレントディレクトリを探し、ファイルがない場合は/usr/include/下を探す**
    - **展開後のソースを見たい場合は gcc -E Cソースファイル名 を実行する。**
  - **共通の宣言や設定はヘッダファイルに保存し、インクルードすることで毎回書く手間が省ける**

define

- **#define -- マクロの設定**
  - 値の設定
    - **#define 名前 値** のように書く
      - 例 #define MAX 100
      - プリプロセッサがソース中の「MAX」を全て100に置き換える
      - 一括で修正出来るので、修正が楽になる
      - ミスが出にくい（修正し忘れが起こらない）
  - マクロのみ設定（値は設定しない）
    - **#define MSDOS** のようにマクロを設定するが、値は書かない
    - これは後述の#ifdef等で使用される

マクロ関数

- `#define sub(a,b) ((a)-(b))`
  のように、引数のある関数風にマクロを書くことが出来る
  - 呼ぶときは `sub(x,y)` のように書く
  - これは `((x)-(y))` に置き換わる
- もう少し有効な例
  - `#define max(a,b) (((a)>(b)) ? (a) : (b))`
  - これは `z = max(x,y);` のように使用出来る
  - `z = (((x)>(y)) ? (x) : (y));` と展開される
- マクロ関数だと引数の型を気にしなくても良い
  - `max(3,5)`（整数）　`max(1.2,3.4)`（浮動小数点）
    どちらも同じマクロを呼ぶことが出来る。
    （しかし割り算のように整数と浮動小数点で動作が違う物は期待通りの動きにならない可能性もある）

3項演算子(条件演算子) P140

- まだ習っていなかった演算子
- 式1 ? 式2 : 式3 という形式
- これは式の値として
  - 式1が真(0以外)なら 式2 の値を取る
  - 式1が偽(0)なら 式3 の値を取る
- 例えば以下のように使用出来る

```
z = x < y ? x : y;        x,yのうち小さい方を代入
b = a % 2 ? a-1 : a;      a(整数)以下の最大の偶数を代入
c = ('A' <= a) && (a <= 'Z') ? a +('a'-'A') : a;
                小文字に変換、但し記号はそのまま(tolower)
```

3項演算子を利用したマクロ

- 3項演算子は特にマクロ関数に良く用いられる
- 前頁のmax以外にも例えば以下のように使用出来る
  - a(整数)以下の最大の奇数を計算する

    `#define maxodd(a) ((a) % 2 ? (a) : (a)-1 )`

    (`(a) % 2 ?` は `(a) % 2 != 0 ?` と同じ意味である)
  - 小文字を大文字に変換(小文字以外はそのまま(touppperと同じ))

```
#define mytoupper(a) (('a' <= (a)) && ((a) <= 'z') ? (a)-('a'-'A') : (a))
```

マクロ関数の利点と弱点

- マクロ関数は簡単な計算式の場合関数より手軽に使用でき、場合によっては高速に計算を行うことが出来る。
- 反面使い方によっては期待しない結果となることもある
  - `#define func(x,y) x * x + y * y` の場合
    `func(a+1,b)` の結果は `a+1 * a+1 + b * b`
    つまり `a+(1*a)+1+b*b` となって、期待通りの答えとならない
  - `#define func(x,y) (x) * (x) + (y) * (y)` にすると、
    `func(a+1,b)` の結果は `(a+1)*(a+1)+(b)*(b)`で正解を得るが、
    `func(a,b)*c` は `a * a + b * b * c` となって、これでも期待通りの答えとならない
  - `#define func(x,y) ((x) * (x) + (y) * (y))` にすると
    `func(a,b)*c` は `((a)*(a)+(b)*(b))*c` で正解を得るが、
    `func(a,b++)` は `a*a+(b++)*(b++)` となって、結局これでも期待通りの答えとならない（この場合に正解を得るには関数にするしかない）

マクロ関数が正常に動作しない例

```
#include<stdio.h>
#define func1(x,y) x * x + y * y
#define func2(x,y) (x) * (x) + (y) * (y)
#define func3(x,y) ((x) * (x) + (y) * (y))
int funcf(int,int);
main()
{
  char *rgt = "Right!", *wng = "Wrong!", *judge;
  int a = 2, b = 3, c = 4, result;

  result = func1(a+1,b);
  judge = result == funcf(a+1,b) ? rgt: wng;
  printf("%d + 1 * %d + 1 + %d * %d : %d : %s\n",a,a,b,b,result,judge);

  result = func2(a+1,b);
  judge = result == funcf(a+1,b) ? rgt: wng;
  printf("(%d+1) * (%d+1) + (%d) * (%d) : %d : %s\n",a,a,b,b,result,judge);

  result = func2(a,b)*c;
  judge = result == funcf(a,b)*c ? rgt: wng;
  printf("(%d) * (%d) + (%d) * (%d) * %d : %d : %s\n",a,a,b,b,c,result,judge);

  result = func3(a,b)*c;
  judge = result == funcf(a,b)*c ? rgt: wng;
  printf("((%d) * (%d) + (%d) * (%d)) * %d : %d : %s\n",a,a,b,b,c,result,judge);
}
int funcf(int x, int y)
{
  return x * x + y * y;
}
```

マクロ関数群

正誤の判定

比較用関数

実行結果

```
s1000001{std0ss0}1: ./a.out
2 + 1 * 2 + 1 + 3 * 3 : 14 : Wrong!
(2+1) * (2+1) + (3) * (3) : 18 : Right!
(2) * (2) + (3) * (3) * 4 : 40 : Wrong!
((2) * (2) + (3) * (3)) * 4 : 52 : Right!
s1000001{std0ss0}2:
```

条件コンパイル

- **#if / endif / else / elif**
  - 条件が成り立った場所のみコンパイルする
  - ifとendifは必ずペアで使用
  - elifはelseとifが組み合わさった物
  - サンプル:

```
#define MSDOS 1
#if MSDOS
 #include "msdos.h"
#else
 #include "unix.h"
#endif
```

Unixの場合は0に、MSDOSの場合はそれ以外に

マクロMSDOSの値が0以外の場合

条件コンパイル

- **#if defined/ if !defined / ifdef / ifndef**
  - マクロの値とは関係なく、マクロが定義されているか（if defined 又は ifdef）、マクロが定義されていないか（if !defined 又は ifndef）でコンパイル動作が変わる。ifと同様にelseが使用出来る
  - 前ページのサンプルは次のように変更できる:

```
#define MSDOS
#if defined MSDOS
 #include "msdos.h"
#else
 #include "unix.h"
#endif
```

値無しマクロMSDOSが定義されている

マクロMSDOSが定義されている場合

条件コンパイルによるデバッグ

```
#include<stdio.h>
#define PRINT    printf("a = %d, b = %d, c = %d, i = %d¥n",a,b,c,i)
#define SEPARATE printf("-----------------------------¥n")
#define DEBUG 1
main(){
  int a = 0, b, c, i;
  for(i = 0; i <= 10; i += 2) {

#if DEBUG
    PRINT;
#endif

    a += i;
    b = i + 1;
    c += b;

#if DEBUG
    PRINT;
    SEPARATE;
#endif

  }
  printf("0+2+4+6+8+10 = %d¥n", a);
  printf("1+3+5+7+9+11 = %d¥n", c);
}
```

- プログラムの動作がおかしい時、デバッグ用にprintfを追加して、変数を表示させることで、おかしい場所を発見する。
- この場合は変数cの初期化がされていない事が分かる

```
s1000001{std0ss0}1: ./a.out
a = 0, b = 4, c = -268436636, i = 0
a = 0, b = 1, c = -268436635, i = 0
-----------------------------
a = 0, b = 1, c = -268436635, i = 2
a = 2, b = 3, c = -268436632, i = 2
-----------------------------
a = 2, b = 3, c = -268436632, i = 4
a = 6, b = 5, c = -268436627, i = 4
-----------------------------
a = 6, b = 5, c = -268436627, i = 6
a = 12, b = 7, c = -268436620, i = 6
-----------------------------
a = 12, b = 7, c = -268436620, i = 8
a = 20, b = 9, c = -268436611, i = 8
-----------------------------
a = 20, b = 9, c = -268436611, i = 10
a = 30, b = 11, c = -268436600, i = 10
-----------------------------
0+2+4+6+8+10 = 30
1+3+5+7+9+11 = -268436600
s1000001{std0ss0}2:
```

コンパイル時にマクロ値を定義する方法

- **gcc -Dマクロ名=値　ファイル名**
  とするとコンパイル時にマクロ定義と値を与えることが出来る。

```
s1000001{std0ss0}1: cat lec12-3.c
#include <stdio.h>

main()
{
  printf("MAX = %d\n", MAX);
}
s1000001{std0ss0}2: gcc lec12-3.c
lec12-3.c: In function 'main':
lec12-3.c:5: 'MAX' undeclared (first use in this function)
lec12-3.c:5: (Each undeclared identifier is reported only once
lec12-3.c:5: for each function it appears in.)
s1000001{std0ss0}3: gcc -DMAX=100 lec12-3.c
s1000001{std0ss0}4: ./a.out
MAX = 100
s1000001{std0ss0}5:
```

マクロ(又は変数)未定義のコンパイルメッセージ

条件コンパイル/コンパイル時マクロ指定によるデバッグ

```
#include<stdio.h>
#define PRINT    printf("a = %d, b = %d, c = %d, i = %d¥n",a,b,c,i)
#define SEPARATE printf("------------------------------¥n")
main(){
  int a = 0, b, c, i;
  for(i = 0; i <= 10; i += 2) {

#if defined DEBUG
    PRINT;
#endif

    a += i;
    b = i + 1;
    c += b;

#if defined DEBUG
    PRINT;
    SEPARATE;
#endif

  }
  printf("0+2+4+6+8+10 = %d¥n", a);
  printf("1+3+5+7+9+11 = %d¥n", c);
}
```

```
s1000001{std0ss0}1: gcc lec12-2d.c
s1000001{std0ss0}2: ./a.out
0+2+4+6+8+10 = 30
1+3+5+7+9+11 = -268436600
s1000001{std0ss0}3: gcc -DDEBUG lec12-2d.c
s1000001{std0ss0}4: ./a.out
a = 0, b = 4, c = -268436636, i = 0
a = 0, b = 1, c = -268436635, i = 0
------------------------------
a = 0, b = 1, c = -268436635, i = 2
a = 2, b = 3, c = -268436632, i = 2
------------------------------
(中略)
------------------------------
a = 20, b = 9, c = -268436611, i = 10
a = 30, b = 11, c = -268436600, i = 10
------------------------------
0+2+4+6+8+10 = 30
1+3+5+7+9+11 = -268436600
s1000001{std0ss0}5:
```

通常モードのコンパイル

デバッグモードのコンパイル

分割コンパイル

- プログラムのモジュール（部品）化
  - 効率的なプログラム開発の際に重要
  - その組合せで1つの物を作る
  - C言語のプログラムでも、モジュール（関数）の集まりが最終的に1つの実行ファイルを作ることになる。
- 大きなプログラムの場合、1つのソースファイルになっていると非常に効率が悪い。
  - ちょっとした修正でも、修正箇所を捜すのに手間がかかる
  - 1ヵ所だけの修正でも、必要のない関数まで全て再コンパイルする必要がある。
  - グループで大きなプログラムを開発する場合、ソースファイルが一つでは複数の人が同時にプログラムを作ることが出来ない。
- ソースファイルをいくつかの部分に分割する　→　分割コンパイル
  - 無駄なコンパイルも減り、コンパイルにかかる余分な時間を節約することが出来る
  - グループでプログラムを作ることも容易である
  - 分割する最小単位は関数

分割コンパイルの手順

- プログラムに必要な要件を洗い出す。
- 関数に分けることを考えながら、要件を満たすような機能を洗い出す。
- 分けた機能（関数）が量的にバランスがとれているか考える
- 各々の機能間のデータのやり取りを考え、各関数の引数と戻り値の意味、型を決める。データのやり取りの取り決めのことを「インタフェース」と言うこともある。
- 各々の機能の担当者と責任者（担当が複数いる場合）を決める。
- 各々の関数を実際に作成（コーディング）する
- 変更が簡単に確実に行われるようにするためには、以下のような工夫をすると良い。
  - インタフェースの取り決めは関数のプロトタイプとしてヘッダファイルにまとめる
  - コメントを詳しく書く。処理やインタフェースを変更した場合は経緯も書いておくと良い。
  - 古いコメントは混乱の元なので、常にコメントは最新にする
- 単体でコンパイルを行う(**gcc -c** ファイル名**.c** で ファイル名.o を作成)
- 単体での動作確認試験（単体テスト）を行う（他関数を適宜補足）
- 全体でコンパイルを行う(**gcc** ファイル1**.o** ファイル2**.o** …. で a.out作成)
- 全体での動作確認試験（結合テスト）を行う

分割コンパイル例（元プログラム）

```
#include<stdio.h>
#include "lec12-4.h"

main()
{
  int i;
  Triangle t;
  for(i = 0; i < 3; i++){
    scanf("%d %d",&t.p[i].x, &t.p[i].y);
  }
  printf("Area = %f¥n",calcarea(t));
}

double calcarea(Triangle t)
{
  double area;

  area = (double)((t.p[2].x -  t.p[1].x)*(t.p[0].y -  t.p[1].y) -
         (t.p[2].y -  t.p[1].y)*(t.p[0].x -  t.p[1].x))/2.0;
  return myabs(area);
}
```

```
#define myabs(x) ((x) > 0 ? (x): -(x))

typedef struct{
  int x;
  int y;
} Point;

typedef struct{
  Point p[3];
} Triangle;

double calcarea(Triangle);
```

lec12-4.h

３点の座標を与え、その三角形の面積を求めるプログラム

分割コンパイル例

lec12-5a.c

```
#include<stdio.h>
#include "lec12-4.h"

main()
{
  int i;
  Triangle t;
  for(i = 0; i < 3; i++){
    scanf("%d %d",&t.p[i].x, &t.p[i].y);
  }
  printf("Area = %f¥n",calcarea(t));
}
```

同じヘッダファイル
をインクルード

lec12-5b.c

```
#include<stdio.h>
#include "lec12-4.h"

double calcarea(Triangle t)
{
  double area;

  area = (double)((t.p[2].x -  t.p[1].x)*(t.p[0].y -  t.p[1].y) -
         (t.p[2].y -  t.p[1].y)*(t.p[0].x -  t.p[1].x))/2.0;
  return myabs(area);
}
```

単体テスト

- 各コンパイル単位毎に普通のコンパイルが出来るよう、必要な機能を仮に付け足して実行し、テストを行う。
  - メイン部分　→　関数を補う
  - 関数部分　→　メインと必要な関数を補う

関数calcarea試験用　データ固定のmainを追加した

```
#include<stdio.h>
#include "lec12-4.h"

main() /* 単体試験用に追加 */
{
  Triangle t = {1,0,0,1,0,0};
  printf("Area = %f¥n",calcarea(t));
}

double calcarea(Triangle t)
{
  double area;

  area = (double)((t.p[2].x -  t.p[1].x)*(t.p[0].y -  t.p[1].y) -
         (t.p[2].y -  t.p[1].y)*(t.p[0].x -  t.p[1].x))/2.0;
  return myabs(area);
}
```

```
#include<stdio.h>
#include "lec12-4.h"
main()
{
  int i;
  Triangle t;
  for(i = 0; i < 3; i++){
    scanf("%d %d",&t.p[i].x, &t.p[i].y);
  }
  printf("Area = %f¥n",calcarea(t));
}

double calcarea(Triangle t) /* 単体試験用 */
{
  int i;
  for(i = 0; i < 3; i++){
    printf("%d %d¥n",t.p[i].x,t.p[i].y);
  }
  return 0;
}
```

main試験用　データ表示だけの関数calcareaを追加した

結合して実行

```
s1000001{std0ss0}1: gcc lec12-5a.c
/var/tmp/ccecTeQB.o: In function `main':
/var/tmp/ccecTeQB.o(.text+0xac): undefined reference to `calcarea'
collect2: ld returned 1 exit status
s1000001{std0ss0}2: gcc lec12-5b.c
/usr/local/gnu/lib/gcc-lib/sparc-sun-solaris2.8/2.95.2/crt1.o: In function
`_start':
/usr/local/gnu/lib/gcc-lib/sparc-sun-solaris2.8/2.95.2/crt1.o(.text+0x5c):
undefined reference to `main'
collect2: ld returned 1 exit status
s1000001{std0ss0}3: gcc -c lec12-5a.c
s1000001{std0ss0}4: gcc -c lec12-5b.c
s1000001{std0ss0}5: ls
lec12-4.h lec12-5a.c lec12-5a.o lec12-5b.c lec12-5b.o
s1000001{std0ss0}6: gcc lec12-5a.o lec12-5b.o
s1000001{std0ss0}7: ls
a.out lec12-4.h lec12-5a.c lec12-5a.o lec12-5b.c lec12-5b.c
s1000001{std0ss0}8: ./a.out
0 0 0 1 1 0
Area = 0.500000
s1000001{std0ss0}9:
```

そのままコンパイルするとエラーになる

-c オプション付きでコンパイルする

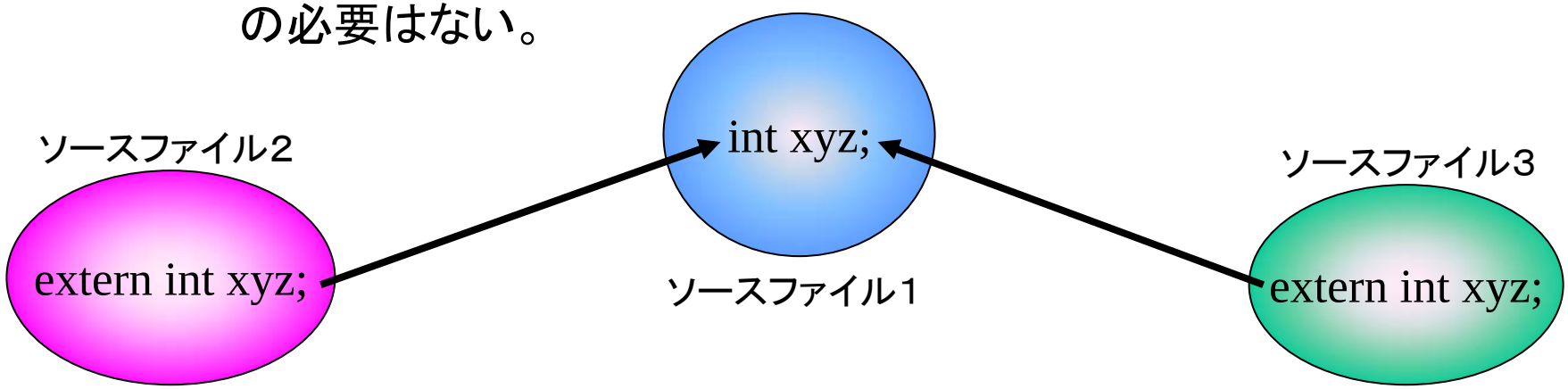
*.o ファイル（オブジェクトファイル）が作成された

更にコンパイル（リンク）すると a.out が作成された

a.outは実行することが出来る

extern と static

- 大規模なプログラムになってくると、各関数が共通に知っておくべき情報が増加する。このような情報は外部（グローバル）変数にもっておくのが一番便利かつスマートである。
  - 分割コンパイルの場合は、使用する側でextern と言うキーワードが必要になる。このキーワードは、別のコンパイル単位にその実体があることを示すものである。
  - なお、メインの外に（つまり外部変数の位置） staticのキーワードをつけた変数を宣言すると、そのコンパイル単位内（つまり同一ファイル）だけで有効な外部変数になる。
- 関数の場合は、常にグローバル扱いとなり、明にexternの宣言の必要はない。



extern と static

なお、関数は省略すると、グローバル・externと同じ（どのコンパイル単位からも見える）となる。他から見えなくさせるためには明に static を付加する。

変数に関するextern/staticを表にまとめる。

	extern	static
通用範囲	全て	コンパイル単位（ファイル）内
宣言	グローバル	static
他コンパイル単位での宣言	extern	他コンパイル単位では見えない

```
****** ソースファイル A *****
extern int a;   /* 参照するための外部変数(メモリには作られない)*/
static int b;   /* 他ファイルからは使えない(別に作られる)*/
main()
{
 ......
}

****** ソースファイル B *****
extern int a;   /* 参照するための外部変数(メモリには作られない)*/
static int b;   /* 他ファイルからは使えない(別に作られる)*/
func1()         /* 関数func1はどのファイルの関数でも使える */
{
 ......
}

****** ソースファイル C ******
int a;          /* externを宣言しない(ここに記憶領域が作られる)*/
static int b;   /* 他ファイルからは見えない(別に作られる)
                   他と同じ名前でもstatic宣言をすると異なる変数となる */
static func2() /* 他ファイルからはこの関数は見えない */
{
 ......
}
```