応用プログラミング

第5回　プロトタイプ宣言、分割コンパイル

メディア情報コース
平居　悠（ひらい　ゆたか）

到達目標

- C言語を通じてコンピュータの根幹部分を理解する。
- 目的に応じたプログラムを自由に設計できる。

前回

第１回　C言語の基本文法・言語処理系
第２回　数学的計算、配列、文字列
第３回　型変換、入出力処理
第４回　デバッグ、関数定義、変数の分離
第５回　プロトタイプ宣言、分割コンパイル
第６回　配列の受け渡し、ポインタ
第７回　まとめ、期末試験

前回の目標

- プログラムをデバッグして正しく動くプログラムに修正できる。
- 関数を定義して自由にプログラムを設計できるようになる。

前回学んだこと

1. デバッグ
2. 関数定義
3. 変数の分類

バグ

プログラムに含まれる論理的誤り

デバッグ

プログラムのバグを取り除く作業

printfデバッグ

printfを要所要所で挟んでその時の変数状態を表示してデバッグする方法。

ソースレベルデバッガ

プログラムのソースとの対応を取りながら、プログラムを少しずつ動かしたり止めたりしつつデバッグを支援するツール。

GDB (GNU Debugger)

GNUプロジェクトで開発されたデバッガ。多くのUnix系システムで利用できる。

gdbによるデバッグ手順

1. コンパイル時に-gオプションを付ける。

```
gcc average01.c –g –o average01
```

gdbによるデバッグ手順

2. gdbの起動。gdbコマンドの後に実行プログラムを指定する。

```
gdb average01
```

gdbによるデバッグ手順

3. プログラムリストを見て行番号を調べる。
listコマンドを打つとプログラムが10行ずつ表示される。l(エル)のみでも可。

```
(gdb) list
```

list 3のように打つと、表示し始める行番号も指定できる。

gdbによるデバッグ手順

4. ブレークポイントを設定する。breakコマンドでプログラムの動作を止めたい行番号を指定する。例えば7行目で止めたい場合は以下のようにする。

(gdb) **break 7**
(または)
(gdb) **b 7**

gdbによるデバッグ手順

5. プログラムを走らせる。runコマンドでデバッグ実行する。

(gdb) **run**
(または)
(gdb) **r**

gdbによるデバッグ手順

6. 設定したブレークポイントの行でプログラムが止まるのでそのあと「ステップ実行」、「変数の値表示」などで調査する。

ステップ実行

(gdb) **s** (または **step**)

変数の値表示

(gdb) **p 変数** (または **print 変数**)

gdbによるデバッグ手順

7. gdbを終了する。

(gdb) **q** (または **quit**)
The program is running. Exit anyway? (y or n) **y**

関数定義

```
戻り値の型　関数名(引数)
{
　やりたい処理;

　return 戻り値;
}
```

引数を持たない関数

```
戻り値の型　関数名(void)
{
   やりたい処理;

   return 戻り値;
}
```

括弧内のvoidは省略できる。

関数の再帰呼び出し

関数内で自分自身を呼び出すこと。

変数の寿命

関数内で変数を宣言すると、その変数のためのメモリが確保される。return すると、確保されたメモリは破棄される。確保されてから破棄されるまでをその変数の**寿命**という。

寿命とスコープ

```
int func(int x, int y)
{
      int kotae;
      kotae = x * y;
      return kotae;
}

int main()
{
   int c;
   c = func(10, 20);
   printf("10 x 20 = %d\n", c);

   return 0;
}
```

kotaeの寿命

変数kotaeはこの範囲（スコープ）内でのみ可視。main関数からは見えない。

cの寿命

変数cはこの範囲（スコープ）内でのみ可視。func関数からは見えない。

変数の分類

- **ローカル変数**：関数内やブロック内のみで有効な変数
- **グローバル変数**：静的領域というメモリ上に記憶され、プログラムが終了するまで値を保持する変数

記憶クラス指定子

変数の前に以下の指定子を付けると変数の種類を指定して宣言できる。

- **static:** 関数がreturnしても値を保持し続ける。
- **auto:** その変数が関数内もしくはブロック内で宣言された変数（**自動変数**）であることを示す。
- **extern:** どこか別の箇所（他のソースファイルでも可）で定義されたグローバル変数を参照する変数を宣言する。

前回の問い

- プログラムをデバッグはどのように行うのか？
  - printfやgdbを利用する。
- C言語で関数はどのように定義すれば良いのか？
  - 「戻り値の型　関数名(引数)」と定義する。

今回

第１回　C言語の基本文法・言語処理系
第２回　数学的計算、配列、文字列
第３回　型変換、入出力処理
第４回　デバッグ、関数定義、変数の分離
第５回　プロトタイプ宣言、分割コンパイル
第６回　配列の受け渡し、ポインタ
第７回　まとめ、期末試験

今回の目標

- 関数をプロトタイプ宣言して読みやすいコードを書けるようになる。
- 複数のソースファイルに書かれたプログラムを分割コンパイルできるようになる。

今回学ぶこと

1. プロトタイプ宣言
2. 分割コンパイル

今回の問い

- プロトタイプ宣言とは何か？
- 分割コンパイルをするにはどのようにすれば良いのか？

今回学ぶこと

1. プロトタイプ宣言
2. 分割コンパイル

プロトタイプ宣言

後で利用する関数について、その引数の個数と型、返す結果の型を先に宣言すること。

これは前回の授業で扱った階乗を求めるプログラムであるが、コンパイルするとエラーになった。どこが間違っているか？

```
#include <stdio.h>

int main()
{
   int i;
   for (i = 0; i < 11; i++)
      printf(“%d! = %d\n”, i, factorial(i));

   return 0;
}

int factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

factorial_p.cというファイルを作成し、右のプログラムを書いて実行してみよう。

```
#include <stdio.h>

int factorial(int);

int main()
{
   int i;
   for (i = 0; i < 11; i++)
      printf(“%d! = %d\n”, i, factorial(i));

   return 0;
}

int factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

プロトタイプ宣言

```
#include <stdio.h>

int factorial(int);

int main()
{
   int i;
   for (i = 0; i < 11; i++)
      printf("%d! = %d\n", i, factorial(i));

   return 0;
}

int factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

今回の問い

- プロトタイプ宣言とは何か？
  - 後で利用する関数について、その引数の個数と型、返す結果の型を先に宣言すること。
- 分割コンパイルをするにはどのようにすれば良いのか？

今回学ぶこと

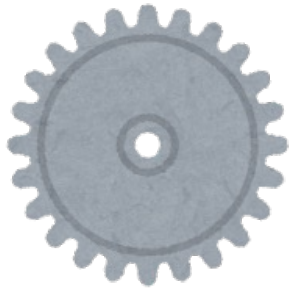
1. プロトタイプ宣言
2. 分割コンパイル

機械語への変換処理：これまでの説明

プログラミング言語をコンピュータが読める機械語に翻訳

ソースプログラム

(C言語)

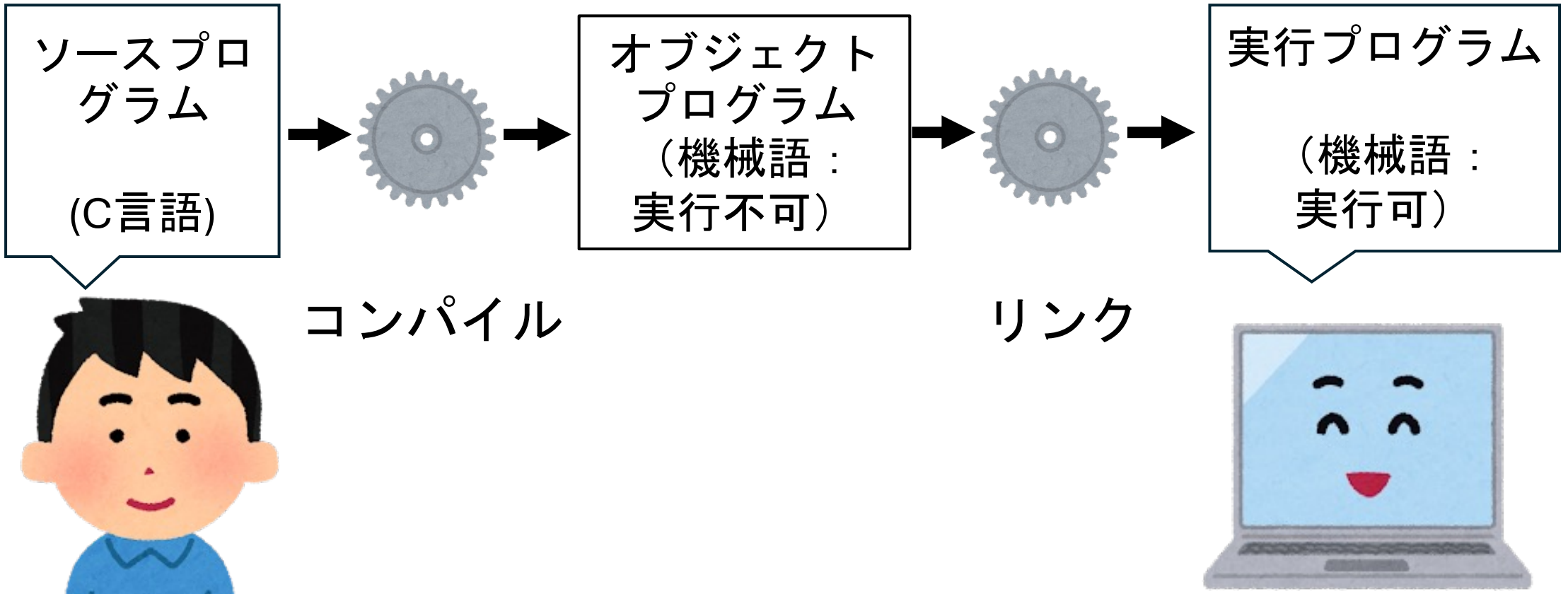


コンパイル

実行プログラム

（機械語）

機械語への変換処理：実際の行程

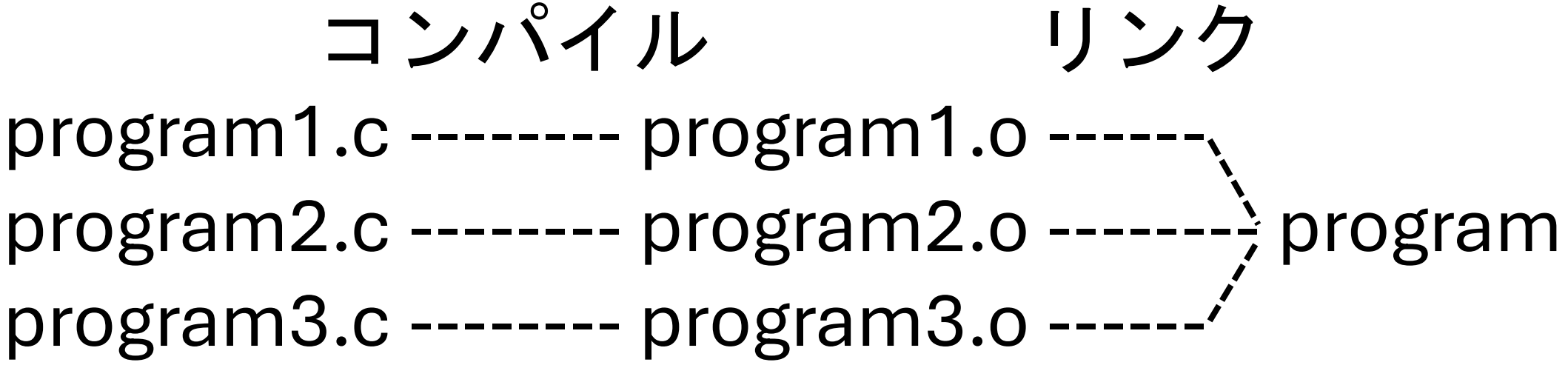


リンク

プログラムの実行に必要な処理やprintfやscanfなどのシステムで用意された関数の実体をオブジェクトプログラムと結合すること

コンパイルとリンクを分ける意味

実用的なプログラムは非常に大きなものになるので、通常複数のファイルに分けて作る。このような場合、個々のソースファイルをコンパイルして最後にリンクする。



分割コンパイル

addというディレクトリ(mkdirコマンドで作成できる)を作成し、その中に右のように2つのソースファイルmain.cとsub.cを作成しよう。

main.c

```
#include <stdio.h>

int main()
{
   int add(int, int);   /* プロトタイプ宣言 */
   int x = 5, y = 4;
   printf("%d + %d = %d\n", x, y, add(x, y));
   return 0;
}
```

sub.c

```
int add(int v1, int v2)
{
   return v1 + v2;
}
```

分割コンパイル

以下のようにターミナルに入力する。

```
gcc -c main.c
gcc -c sub.c
gcc -o main main.o sub.o
```

実行

以下のようにターミナルに入力する。

```
./main
```

makeユーティリティ

ソースファイルが多くなると、コンパイル＆リンク作業を行うのが複雑になってくる。makeユーティリティはこの負担を軽減してくれる。

ファイルの依存関係

２つのソースファイルmain.cとsub.cから、１つの実行ファイルmainを作るときの依存関係：

材料となるファイル	その成果物
main.c	main.o
sub.c	sub.o
main.oとsub.o	main

Makefile

makeにファイル依存関係を教えておくと、makeを起動したときに適切な処理を行ってくれる。makeに依存関係を教えるには、作業ディレクトリに**Makefile**というファイルを置き、そこに依存関係を記述する。

Makefileの書き方

ターゲット：<TAB空白>依存ファイル（群）
<TAB空白＞それを作るためのコマンドライン

- コロン(:)の後とコマンドラインを各行の先頭はスペースではなく**TAB**コードを入れる。
- #で始まる行はコメント行として解釈対象外となる。
- 慣習として最初にallというターゲットを置き、最終ターゲットをその右辺に書く。

Makefileの書き方

以下のMakefileを作成してみよう

```
#
# Makefile for main program
#
all:	main

main:	main.o sub.o
	gcc -o main main.o sub.o

main.o:		main.c
	gcc -c main.c
sub.o:	sub.c
	gcc -c sub.c
```

makeの起動

make ターゲット

makeは指定されたターゲットとその右辺に書かれた依存ファイル(群)のどちらが新しいファイルか調べ、依存ファイルの方が新しい場合、ターゲットも更新が必要だと判断し、次の行から始まるコマンドラインを起動する。

makeの起動

以下のようにターミナルに入力する。

```
make
```

実行プログラムを新たに生成したい場合は、main, sub.o, main.oを削除 (rmコマンドで削除できる）してからmakeと入力する。

makeの動作

ターゲットとしてmainファイルを指定

```
all:    main

main:         main.o sub.o
        gcc -o main main.o sub.o

main.o:      main.c
        gcc -c main.c
sub.o:        sub.c
        gcc -c sub.c
```

makeの動作

左辺に書かれているターゲットのmainファイルよりも右辺に書かれたmain.o, sub.oの方が新しければ、次の行のgcc –o main main.o sub.oを実行する。

```
all:    main

main:       main.o sub.o
        gcc -o main main.o sub.o

main.o:     main.c
        gcc -c main.c
sub.o:      sub.c
        gcc -c sub.c
```

makeの動作

main.o、sub.oの依存関係を調べる。これは前ページの操作の前に行われる。

```
all:    main

main:       main.o sub.o
        gcc -o main main.o sub.o

main.o:     main.c
        gcc -c main.c
sub.o:      sub.c
        gcc -c sub.c
```

Makefileを作る上での注意

makeはデフォルトでは必ずMakefileというファイルを依存関係登録ファイルとして参照するので、makeを利用してコンパイルを行う場合には作りたいものに対して一つのディレクトリを作成するようにする。

作業の流れ：

mkdir newProject
cd newProject
(Makefileとソースファイルを作成）
make

より進んだMakefileの書き方

```
clean:
	rm -f *.o main
```

とMakefileに書くと、

```
make clean
```

で必ずrm –f *.o mainを起動する。rmはファイルを消すコマンドで、-fオプションをつけるとそのファイルがなくても文句を言わず消す。

より進んだMakefileの書き方

```
backup:
	gtar zcf backup.tar.gz *.c
```

カレントディレクトリの*.cファイルを全て圧縮してbackup.tar.gzファイルにバックアップする。

分割コンパイル演習

次の３ページに渡るプログラムは変数のスコープを確かめるプログラムである。新たに作成したscopeというディレクトリにmain.c, source1.c, source2.cというファイルを作成し、makeでコンパイルして実行せよ。

```
/* main.c */

#include <stdio.h>

int func1();
int func2();

int a = 100;

int main()
{
   extern char *str;

   printf(“ここはmain関数です。\na = %d\n”, a);
   printf(“strも見えます---%s\n”, str);
   func1();
   func2();
   return 0;
}
```

```
/* source1.c */

#include <stdio.h>

extern int a;

int func1()
{
   extern char *str;

   printf(“ここはsource1.cのfunc1関数です。\na = %d\n”, a);
   printf(“strも見えます---%s\n”, str);
   return 0;
}
```

```
/* source2.c */

#include <stdio.h>

int func2()
{
   extern int a;
   extern char *str;

   printf(“ここはsource2.cのfunc2関数です。\na = %d\n”, a);
   printf(“strも見えます---%s\n”, str);
   return 0;
}

char *str = “abc”;
```

今回の問い

- プロトタイプ宣言とは何か？
  - 後で利用する関数について、その引数の個数と型、返す結果の型を先に宣言すること。
- **分割コンパイルをするにはどのようにすれば良いのか？**
  - **“gcc –c”でオブジェクトファイルを作成し、リンクする**

今回学んだこと

1. プロトタイプ宣言
2. 分割コンパイル

今回の目標

- 関数をプロトタイプ宣言して読みやすいコードを書けるようになる。
- 複数のソースファイルに書かれたプログラムを分割コンパイルできるようになる。

課題

前回作成した円の面積を表示するプログラムで定義した関数を別のファイルで定義して実行できるプログラムを作成せよ。コンパイルはmakeを用いて行うこと。生成AIでのコード生成は不可。

提出先：yutaka.hirai@koeki-u.ac.jp

件名：応用プログラミング第5回課題 [学籍番号]

提出物：**ソースファイル一式とMakefile**

締め切り：**11月5日 (木)**

次回

第１回　C言語の基本文法・言語処理系
第２回　数学的計算、配列、文字列
第３回　型変換、入出力処理
第４回　デバッグ、関数定義、変数の分離
第５回　プロトタイプ宣言、分割コンパイル
第６回　配列の受け渡し、ポインタ
第７回　まとめ、期末試験