

情報基礎 B
第 11 回 アカデミック・スキル II
C 言語プログラミング (3)

長江 剛志

(nagae@m.tohoku.ac.jp)

東北大学大学院工学研究科
技術社会システム専攻

2015 年 6 月 19 日 (金)

今日やること

if 文による条件分岐

while 文による繰返し

for 文による回数指定繰返し

ループの無駄を省いて高速化する

レポート課題 III-3: 素数の個数と和

ディレクトリとサンプル・コードの準備 (1)

ディレクトリの準備

Terminal 上から `mkdir` コマンドを使って
~/Documents/prog/03_ctrl というディレクトリを作る.

サンプル・コードのダウンロード

- ▶ ISTU (<http://www.istu.jp>) にアクセスし, 右下の 受講授業科目 から, 金曜の 3 時限 情報基礎 B を選択
- ▶ 科目共通教材の中から 配布資料 第 11 回 C 言語サンプルプログラム (2) を選択
- ▶ `ctrl_if.c`, `ctrl_if_else.c`, `fizzbuzz.c`, `ctrl_while.c`, `ctrl_do_while.c`, `ctrl_for.c`, `is_prime.c`, `is_prime_v2.c` の 8 つのファイルをダウンロード

ディレクトリとサンプル・コードの準備 (2)

サンプル・コードの移動

- ▶ Home (メニューバーの「場所」でもよい) を開いて「ダウンロード」内にある上記のファイルを
~/Documents/prog/03_ctrl1 へ移動.

現在の作業ディレクトリの変更

Terminal 上の `cd` コマンドを使って、現在の作業ディレクトリを
~/Documents/prog/03_ctrl1 に変更する.

if 文による条件分岐 (ctrl_if.c)

やりたいこと

入力された整数が **奇数** ならば「奇数」と出力

サンプル・コード (ctrl_if.c)


```
1 #include <stdio.h>           /* 標準入出力ライブラリ */
2 int main(void)               /* main 関数の引数と戻り値の定義 */
3 {                             /* main 関数の始まり */
4     int a;                   /* 整数型の変数 a を宣言 */
5     scanf("%d", &a);         /* キーボードから a に値を読み込む */
6
7     if (a % 2 == 1)          /* 奇数かどうかを判定 */
8     {                         /* 奇数の時に実行されるブロックの始まり */
9         printf("%dは奇数です\n", a);
10    }                         /* 奇数の時に実行されるブロックの終わり */
11    /* aが奇数でなかったときには何もしない */
12
13    return 0;                 /* 戻り値として 0 を返す */
14 }
```

ctrl_if.c のコンパイルと実行




1. Terminal 上で ls コマンドを使い、現在の作業ディレクトリに ctrl_if.c があるか確認する.

```
$ ls   
ctrl_if.c      # ctrl_if.c が表示されることを確認する.
```

2. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて、実行ファイル名を ctrl_if とする.

```
$ gcc -o ctrl_if ctrl_if.c 
```

3. 生成された実行ファイル (./ctrl_if) を呼び出す.

```
$ ./ctrl_if   
7  # 入力待ち状態になるので適当な数値を入れて   
7は奇数です
```

4. 色々な数値を入れて正しく動くか確認してみよう

ctrl_if.c の解説 (1)

- ▶ 1～3 行目: 省略
- ▶ 4～5 行目:

```
int a;                                /* 整数型の変数 a を宣言 */
scanf("%d", &a);                      /* キーボードから a に値を読み込む */
```

整数型変数 a を定義し, scanf 関数を使ってキーボードから読み込んだ値を a に代入する.

- ▶ 7～10 行目:

```
if (a % 2 == 1)                       /* 奇数かどうかを判定 */
{                                       /* 奇数の時に実行されるブロックの始まり */
    printf("%dは奇数です\n", a);
}                                       /* 奇数の時に実行されるブロックの終わり */
```

if 文を使って a が奇数か否かを判定し, 奇数ならそれを表示させる.

- ▶ 7 行目の if 文は, 括弧の中の **条件式** を評価し, それが **真** (0 でない) ならば, その直後の **ブロック** を実行する.

ctrl_if.c の解説 (2)

- ▶ この場合, まず, 括弧内の $a \% 2 == 1$ が評価される. ここで, $==$ は比較演算子 (後述) の 1 つで, 左辺と右辺を比較し, それが等しいなら 真, 等しくないなら 偽 を返す.
- ▶ $a \% 2$ は「 a を 2 で割った余り」だから, これが 1 に等しい (i.e. a が奇数) の時に 8 行目の { から 10 行目の } で囲まれたブロック を実行する.
- ▶ このブロックでは, 9 行目で `printf` 関数を用いて入力された値 ($\%d$ を置換) と, それが奇数であることを表示している.

比較演算子

左辺と右辺を比較し、それがあある条件を満足すれば **真**, そうでないなら **偽** を返す **演算子**. C 言語で用いられる比較演算子は以下の通り:

演算	演算子	a が 3, b が 2 のときの結果
等しい	==	$a==b \rightarrow$ 偽
等しくない	!=	$a!=b \rightarrow$ 真
より小さい	<	$a<b \rightarrow$ 偽
より大きい	>	$a>b \rightarrow$ 真
以上	<=	$a<=b \rightarrow$ 偽
以下	>=	$a>=b \rightarrow$ 真

注意点:

- ▶ 「等しい」が **2 個の連続した等号** == で表される
- ▶ 「以上」「以下」は **不等号の後に等号** が入る (<=, >=)

if...else 文による条件分岐 (ctrl_if_else.c)

やりたいこと

ctrl_if.c を修正し、入力された整数が **偶数** だった場合には「偶数です」と表示させる。

サンプル・コード (ctrl_if_else.c)

```
1 #include <stdio.h>          /* 標準入出力ライブラリ */
2 int main(void)              /* main 関数の引数と戻り値の定義 */
3 {                            /* main 関数の始まり */
4     int a;                  /* 整数型の変数 a を宣言 */
5     scanf("%d", &a);        /* キーボードから a に値を読み込む */
6
7     if (a % 2 == 1)          /* 奇数かどうかを判定 */
8     {                        /* 奇数の時に実行されるブロックの始まり */
9         printf("%dは奇数です\n", a);
10    }                        /* 奇数の時に実行されるブロックの終わり */
11    else                      /* 奇数でない時に実行するブロックがあることを指示 */
12    {                        /* 奇数でない時に実行されるブロックの始まり */
13        printf("%dは偶数です\n", a);
14    }                        /* 奇数でない時に実行されるブロックの終わり */
15
16    return 0;                /* 戻り値として 0 を返す */
17 }
```

ctrl_if_else.c のコンパイルと実行

1. Terminal 上で ls コマンドを使い、現在の作業ディレクトリに ctrl_if_else.c があるか確認する.

```
$ ls  
ctrl_if_else.c # ctrl_if_else.c が表示されることを確認する.
```

2. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて、実行ファイル名を ctrl_if_else とする.

```
$ gcc -o ctrl_if_else ctrl_if_else.c
```

3. 生成された実行ファイル (./ctrl_if_else) を起動.

```
$ ./ctrl_if_else  
7 # 入力待ち状態になるので適当な数値を入れて  
7は奇数です  
$ ./ctrl_if_else  
10  
10は偶数です
```

4. 色々な数値を入れて正しく動くか確認してみよう

ctrl_if_else.c の解説 (1)

▶ 7～14 行目:

```
if (a % 2 == 1)          /* 奇数かどうかを判定 */
{                        /* 奇数の時に実行されるブロックの始まり */
    printf("%dは奇数です\n", a);
}                        /* 奇数の時に実行されるブロックの終わり */
else                    /* 奇数でない時に実行するブロックがあることを指示 */
{                        /* 奇数でない時に実行されるブロックの始まり */
    printf("%dは偶数です\n", a);
}                        /* 奇数でない時に実行されるブロックの終わり */
```

if...else... 文を使って a が奇数でない場合 (i.e. 偶数) の場合には、それを表示させる。

- ▶ 7～10 行目は ctrl_if.c と同じ
- ▶ 11 行目の else は、if で判定した条件が **偽** の場合に、その直後の **ブロック** (12 行目の { から 14 行目の }=の間) を実行することを指示している。
- ▶ このブロックでは、13 行目で printf 関数を用いて入力された値 (%d を置換) と、それが偶数であることを表示している。

else if を使った多分岐 (fizzbuzz.c) (1)

やりたいこと

整数を入力し、それが 3 と 5 で割り切れるか否かに応じて以下のように表示する:

- ▶ 3 で割り切れるなら **Fizz** と表示
- ▶ 5 で割り切れるなら **Buzz** と表示
- ▶ 3 でも 5 でも割り切れるなら **Fizz Buzz** と表示
- ▶ 3 でも 5 でも割り切れないなら **入力された数値** をそのまま表示

(サンプル・コードは次のページ)

else if を使った多分岐 (fizzbuzz.c) (2)

サンプル・コード (fizzbuzz.c)

```
1 #include <stdio.h>           /* 標準入出力ライブラリ */
2 int main(void)               /* main 関数の引数と戻り値の定義 */
3 {                             /* main 関数の始まり */
4     int a;                   /* 整数型の変数 a を宣言 */
5     scanf("%d", &a);         /* キーボードから a に値を読み込む */
6
7     if ( (a % 3 == 0) && (a % 5 == 0) ) /* 3でも5でも割り切れる時の処理 */
8     { printf("Fizz Buzz\n"); }
9     else if (a % 3 == 0)      /* 3で割り切れる時の処理 */
10    { printf("Fizz\n"); }
11    else if (a % 5 == 0)      /* 5で割り切れる時の処理 */
12    { printf("Buzz\n"); }
13    else                      /* 3でも5でも割り切れないときの処理 */
14    { printf("%d\n", a); }
15
16    return 0;                 /* 戻り値として 0 を返す */
17 }
```

fizzbuzz.c のコンパイルと実行

1. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて, 実行ファイル名を fizzbuzz とする.

```
$ gcc -o fizzbuzz fizzbuzz.c ↩
```

2. 生成された実行ファイル (./fizzbuzz) を起動.

```
$ ./fizzbuzz ↩
3 ↩ # 入力待ち状態になるので適当な数値を入れて ↩
Fizz
$ ./fizzbuzz ↩
15 ↩
Fizz Buzz
$ ./fizzbuzz ↩
11 ↩
11
```

3. 色々な数値を入れて正しく動くか確認してみよう

fizzbuzz.c の解説 (1)

▶ 7～14 行目:

```
if ( (a % 3 == 0) && (a % 5 == 0) ) /* 3でも5でも割り切れる時の処理 */
{ printf("Fizz Buzz\n"); }
else if (a % 3 == 0) /* 3で割り切れる時の処理 */
{ printf("Fizz\n"); }
else if (a % 5 == 0) /* 5で割り切れる時の処理 */
{ printf("Buzz\n"); }
else /* 3でも5でも割り切れないときの処理 */
{ printf("%d\n", a); }
```

if...else if...else... 文による多分岐を使って a が 3 や 5 で割り切れるか否かに応じて異なる処理を行う。

- ▶ 7 行目: if の条件式において, $(a \% 3 == 0)$ (a が 3 で割り切れる) と $(a \% 5 == 0)$ (a が 5 で割り切れる) を繋ぐ **&&** は論理積を表す **論理演算子** (後述) であり, 左辺と右辺の評価式がいずれも **真** である時にのみ **真** となる。つまり, $(a \% 3 == 0) \&\& (a \% 5 == 0)$ は, 「a が 3 でも 5 でも割り切れる時」に **真** となる。

fizzbuzz.c の解説 (2)

- ▶ 8 行目: a が 3 でも 5 でも割り切れる時に実行される ブロック を 1 行にまとめて記述している. a が「3 か 5 の少なくとも一方では割り切れない」場合は 9 行目以降が実行される.
- ▶ 9~10 行目: else if を使って「a が 3 で割り切れるか」を判定し, それが 真 であるなら Fizz と表示する. 7 行目の if の条件が満足されていないため, この条件は「a が 3 では割り切れるが 5 では割り切れない」時に 真 となる.
- ▶ 11~12 行目: 再び else if を使って「a が 5 で割り切れるか」を判定し, それが 真 であるなら Buzz と表示する. 直前の条件を満足していないため, この条件は「a が 5 では割り切れるが 3 では割り切れない」時に 真 となる.
- ▶ 13~14 行目: else を遣って, ここまでの条件がどれ 1 つ満足されない場合の処理を記述している.

論理演算子

複雑な条件式を表現する場合には **論理演算子** を用いる。論理演算子は、その左辺と右辺に論理式を持ち、それぞれの真偽の組み合わせに応じた結果を返す。**論理積** (&&) と **論理和** (||) がよく使われる。

論理積

左辺と右辺の **両方が真** の時にのみ真となる。論理表現の「および (and)」に相当する。

a	b	a && b
真	真	真
真	偽	偽
偽	真	偽
偽	偽	偽

論理和

左辺と右辺の **少なくとも一方が真** の時に真となる。論理表現の「または (or)」に相当する。

a	b	a b
真	真	真
真	偽	真
偽	真	真
偽	偽	偽

while 文による繰返し (ctrl_while.c)

やりたいこと

入力された整数を $2^n * m$ の形に分解して表示する. $^$ はべき乗を表す記号で, 2^n は「2 の n 乗」を意味する. 例えば $96 = 2^5 * 3$, $128 = 2^7 * 1$, $7 = 2^0 * 7$.

```
1 #include <stdio.h>           /* 標準入出力ライブラリ */
2 int main(void)               /* main 関数の引数と戻り値の定義 */
3 {                             /* main 関数の始まり */
4     int a;                   /* 整数型の変数 a を宣言 */
5     scanf("%d", &a);         /* キーボードから a に値を読み込む */
6     printf("%d", a);         /* a の値を表示 */
7
8     int n = 0;               /* 2で割り切れる回数を記録する整数 */
9     while (a % 2 == 0)       /* a が2で割り切れるなら次のブロックを実行 */
10    {                         /* a が2で割り切れる時のブロックの始まり */
11        ++ n;                 /* n の値を1つ増やす */
12        a = a / 2;           /* a を2で割った商をaに代入 */
13    }                         /* a が2で割り切れる時のブロックの終わり */
14    printf(" = 2^%d * %d\n", n, a); /* 2^n * a という形で2で割り切れる数を出力 */
15
16    return 0;                 /* 戻り値として 0 を返す */
17 }
```

ctrl_while.c のコンパイルと実行

1. Terminal 上で ls コマンドを使い、現在の作業ディレクトリに ctrl_while.c があるか確認する.

```
$ ls  
ctrl_while.c # ctrl_while.c が表示されることを確認する.
```

2. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて、実行ファイル名を ctrl_while とする.

```
$ gcc -o ctrl_while ctrl_while.c
```

3. 生成された実行ファイル (./ctrl_while) を起動.

```
$ ./ctrl_while  
128 # 入力待ち状態になるので適当な数値を入れて  
128 = 2^7 * 1  
$ ./ctrl_while  
96  
96 = 2^5 * 3
```

4. 色々な数値を入れて正しく動くか確認してみよう

ctrl_while.c の解説 (1)

▶ 6 行目

```
printf("%d", a);          /* a の値を表示 */
```

このプログラムでは実行中に a の値を書き換えるため、入力された値を出力しておく。

▶ 8 行目

```
int n = 0;                /* 2で割り切れる回数を記録する整数 */
```

2 で割り切れる回数をカウントするために新しい整数型の変数 n を定義し、0 で **初期化** する。

▶ 9～13 行目

```
while (a % 2 == 0)        /* a が2で割り切れるなら次のブロックを実行 */
{                          /* a が2で割り切れる時のブロックの始まり */
    ++ n;                 /* n の値を1つ増やす */
    a = a / 2;            /* a を2で割った商をaに代入 */
}                          /* a が2で割り切れる時のブロックの終わり */
```

while 文を使って a が 2 で何回割り切れるかを数えている。

ctrl_while.c の解説 (2)

- ▶ 9 行目の while は、括弧の中の **条件式** を評価し、それが **真** である間、その直後の **ブロック** を **繰り返し** 実行する。
- ▶ ここでは a が 2 で割り切れる間、10 行目の { から 13 行目の } までのブロックが実行される。
- ▶ 11 行目では、**インクリメント演算子** (++) を用いて、n の値を 1 つ増やしている。C 言語では「n の値を 1 つ増やす」という操作が多用されるため、わざわざ $n = n + 1$ と書かなくても済むように ++n という演算子が用意されている。
- ▶ 11 行目では、a を 2 で割ったものを再び a に代入している。ブロックはここで終わっているため、プログラムは **再び 9 行目に戻って** a が 2 で割り切れるかどうかを判定する (2 で割り切れるなら再びこのブロックを実行する)。
- ▶ **ブロックが実行されるたび** に 11 行目で a の値が **半分になる** ことに注意せよ。このため、a は **いつかは** 2 で割り切れない値になり、while ループは終了する。

ctrl_while.c の解説 (3)

- ▶ while ループが終了した時, 変数 n には 入力された整数が 2 で割り切れる回数 (while ブロックが実行された回数) が入り, 変数 a には 入力された変数を 2 で n 回割った時に残った数が入っている.
- ▶ 14 行目

```
printf(" = 2^%d * %d\n", n, a); /* 2^n * a という形で2で割り切れる数を出力 */
```

printf 関数を使って, 入力された変数が 2 で n 回割り切れ, その残りが a となる ことを $= 2^n * a$ という形式で出力する.

do...while 文による繰り返し (ctrl_do_while.c)

やりたいこと

任意の桁の整数を入力された時, その順序を反転させた整数を表示する. ただし, 1200 などを反転させた場合は 0021 ではなく, 21 と表示する.

サンプル・コード (ctrl_do_while.c)

```
1 #include <stdio.h>          /* 標準入出力ライブラリ */
2 int main(void)              /* main 関数の引数と戻り値の定義 */
3 {                            /* main 関数の始まり */
4     int a;                  /* 整数型の変数 a を宣言 */
5     scanf("%d", &a);        /* キーボードから a に値を読み込む */
6     printf("%d", a);        /* a の値を表示 */
7
8     int b = 0;              /* 出力用の値を宣言 */
9     do {                    /* 繰り返しの始まり */
10         b = b * 10 + a % 10; /* b を左に1桁シフトし, a の最小桁の値を加える */
11         a = a / 10;         /* a を右に1桁シフトさせる */
12     } while (a > 0);        /* a が 0 になるまで繰り返す */
13
14     printf("<->%d\n", b);    /* b の値を表示 */
15
16     return 0;              /* 戻り値として 0 を返す */
17 }
```


ctrl_do_while.c のコンパイルと実行

1. Terminal 上で ls コマンドを使い、現在の作業ディレクトリに ctrl_do_while.c があるか確認する.
2. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて、実行ファイル名を ctrl_do_while とする.

```
$ gcc -o ctrl_do_while ctrl_do_while.c
```

3. 生成された実行ファイル (./ctrl_do_while) を起動.

```
$ ./ctrl_do_while
128 # 入力待ち状態になるので適当な数値を入れて
128 <-> 821
$ ./ctrl_do_while
96
9600 <-> 69
$ ./ctrl_do_while
1
1 <-> 1
```

4. 色々な数値を入れて正しく動くか確認してみよう

ctrl_do_while.c の解説 (1)

▶ 8 行目

```
int b = 0; /* 出力用の値を宣言 */
```

順序を反転させた値を格納する新たな整数型変数 `b` を定義し, `0` で **初期化** しておく.

▶ 9~12 行目

```
do { /* 繰り返しの始まり */  
    b = b * 10 + a % 10; /* b を左に1桁シフトし, a の最小桁の値を加える */  
    a = a / 10; /* a を右に1桁シフトさせる */  
} while (a > 0); /* a が 0 になるまで繰り返す */
```

`do...while` 文を使って, 以下の手続きを繰り返している:

Step 1 `b` の全体を左に 1 桁シフトし (ずらし), 空いた 1 桁目に `a` の最小桁の値を入れる

Step 2 `a` の最小桁を削除し, 全体を右に 1 桁シフトする.

ctrl_do_while.c の解説 (2)

Step 3 a の全ての桁を b に移し終わってれば終了.
そうでなければ Step 1 に戻る.

各行での処理は以下の通り：

- ▶ 9 行目の do は「この後の **ブロック** を実行し、その後の while で指定された条件が満足されるなら再び **ブロック** の先頭から処理を繰り返す」ことを指示している.
- ▶ ここでの **ブロック** は 9 行目の { から 12 行目の } までの間.
- ▶ 10 行目は Step 1 に対応する処理を行なう. $b * 10$ は b の全体を左に 1 桁シフトしたもので、それに a の最小桁の値 $a \% 10$ を加えたものを、新たな b として上書きしている.
- ▶ 11 行目は Step 2 に対応する処理を行なう. $a / 10$ は a の最小桁を削除し、全体を右に 1 桁シフトしたものであり、これを新たな a として上書きしている.
- ▶ 12 行目の while は **括弧内の条件式が満足されるなら、再び do 以下のブロックを実行する** ことを指示している.

前判定繰り返しと後判定繰り返し

while ループと do...while ループは、それぞれ、**前判定繰り返し**と**後判定繰り返し**と呼ばれる。

前判定繰り返し (while ループ)

繰り返し処理を行なう **ブロック** の **前** に条件判定がある。このため、処理ブロックが **一度も実行されないことがある**。

ctrl_while.c では、**入力された整数が 2 で割り切れない時** は n の値を増やしたり a を 2 で割る必要がない。このような時には while ループを使う。

後判定繰り返し (do...while ループ)

繰り返し処理を行なう **ブロック** の **後** に条件判定がある。このため、処理ブロックは **少なくとも一度は実行される**。

ctrl_do_while.c では、入力された整数は必ず **1 桁以上** であるため、少なくとも一度は **Step 1～Step 3** を実行する必要がある。このような場合には do...while ループを使う。

for 文による繰返し (ctrl_for.c)

やりたいこと

ある整数を入力されたとき、それより小さい の整数の中で 3 または 5 の倍数 の和を求めたい。

サンプル・コード (ctrl_for.c)

```
1 #include <stdio.h> /* 標準入出力ライブラリ */
2 int main(void) /* main 関数の引数と戻り値の定義 */
3 { /* main 関数の始まり */
4     int a; /* 整数型の変数 a を宣言 */
5     scanf("%d", &a); /* キーボードから a に値を読み込む */
6
7     int sum = 0; /* 合計値を格納する変数を宣言し、0で初期化 */
8     int i; /* 和の候補を格納する変数を宣言 */
9     for (i = 3; i < a; ++i) /* i=3からi<aである間、iを1つづ増やしながら処理を繰り返す */
10     {
11         if ((i % 3 == 0) || /* i が 3もしくは5の倍数なら、その値を sum に加える */
12             (i % 5 == 0))
13             { sum = sum + i; }
14     }
15     printf("sum=%d\n", sum); /* sum を出力する */
16
17     return 0; /* 戻り値として 0 を返す */
18 } /* main 関数の終わり */
```

ctrl_for.c のコンパイルと実行

1. Terminal 上で ls コマンドを使い、現在の作業ディレクトリに ctrl_for.c があるか確認する.
2. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて、実行ファイル名を ctrl_for とする.

```
$ gcc -o ctrl_for ctrl_for.c
```

3. 生成された実行ファイル (./ctrl_for) を起動.

```
$ ./ctrl_for
10          # 10と入力して
sum=23      # 3 + 5 + 6 + 9 = 23 が表示される
$ ./ctrl_for
20          # 20と入力して
sum=78
$ ./ctrl_for
10000000
sum=1404932684 # 大きい数でも一瞬で計算できる
```

4. 色々な数値を入れて正しく動くか確認してみよう

ctrl_for.c の解説 (1)

▶ 7 行目

```
int sum = 0; /* 合計値を格納する変数を宣言し, 0で初期化 */
```

「和」を格納する変数 `sum` を宣言し, 0 で **初期化** する.

▶ 8 行目

```
int i; /* 和の候補を格納する変数を宣言 */
```

和に加える候補 (`a` 未満の整数) を格納する変数 `i` を宣言する.

▶ 9~14 行目

```
for (i = 3; i < a; ++i) /* i=3からi<aである間, iを1つづ増やしながら処理を繰り返す */
{
    if ((i % 3 == 0) || /* i が 3もしくは5の倍数なら, その値を sum に加える */
        (i % 5 == 0))
        { sum = sum + i; }
}
```

ctrl_for.c の解説 (2)

for 文を使った繰り返しによって i の値を増やしながら, a 未満の 3 か 5 の倍数 の和を求める. 各行での処理は以下の通り:

- ▶ 9 行目の for の括弧の中は, 次の 3 つの 式 を ; (セミコロン) で区切って for(式 1; 式 2; 式 3) と記述している.

式 1 (初期化) ここに書かれた 式 は, 最初の繰り返し処理が始まる 前 に一度だけ実行される. ここでの式 ($i=3$) は, 和の候補を格納する変数 i に, 候補となる最小の整数 3 を代入している.

式 2 (継続条件) 各繰り返し処理の 前 に, ここに書かれた 条件式 が評価され, それが 真 の時にのみ ブロック が実行される. ここでの条件式 ($i < a$) は「候補 i が a より小さい間」を意味する.

式 3 (繰り返し毎の処理) 各繰り返し処理が 終わる度 に, ここに書かれた 式 が実行される. ここでの処理 ($++i$) は i の値を 1 だけ増やしている.

ctrl_for.c の解説 (3)

なお, この for 文の繰り返し対象となるブロックは 10 行目の { から 14 行目の } まで.

- ▶ 11~13 行目では, if 文を用いて「i が 3 または 5 で割り切れる」ならば i を sum に加えている. ここで「i が 3 または 5 で割り切れる」という条件は「i が 3 で割り切れる $((i \% 3 == 0))$ 」という論理式と「i が 5 で割り切れる $((i \% 5 == 0))$ 」という論理式を論理和 (\parallel) を使って繋ぐことで表現している.

ループの無駄を省く—素数の判定 (is_prime.c) (1)

やりたいこと

入力された整数が **素数か否か** を判定する。素数でないならその **因数分解** の例を表示する。

(サンプル・コードは次のページに)

ループの無駄を省く—素数の判定 (is_prime.c) (2)

サンプル・コード (is_prime.c)

```
1  #include <stdio.h>          /* 標準入出力ライブラリ */
2  int main(void)              /* main 関数の引数と戻り値の定義 */
3  {                            /* main 関数の始まり */
4      int a;                  /* 整数型の変数 a を宣言 */
5      scanf("%d", &a);        /* キーボードから a に値を読み込む */
6
7      int n;                  /* 候補の数を順に格納するための変数 */
8      int divisor = 1;         /* 1より大きい約数を格納する変数 */
9      for (n = 2; n < a; ++n)  /* 約数の候補を2から順に1ずつ増やしながら繰り返す */
10     {                       /* 繰り返しの始まり */
11         if (a % n == 0)       /* aが n で割り切れるかを判定 */
12         {
13             divisor = n;      /* aが n で割り切れるならそれを divisor として記憶 */
14         }
15     }                       /* 繰り返しの終了 */
16
17     if (divisor == 1 && a != divisor) /* 1より大きくaより小さい約数が無ければ素数 */
18     { printf("%dは素数です\n", a); }
19     else /* 1より大きい約数があったら合成数と表示 */
20     { printf("%d=%d*d\n", a, divisor, a/divisor); }
21
22     return 0;                /* 戻り値として 0 を返す */
23 }
```

is_prime.c のコンパイルと実行

1. Terminal 上で ls コマンドを使い、現在の作業ディレクトリに is_prime.c があるか確認する.
2. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて、実行ファイル名を is_prime とする.

```
$ gcc -o is_prime is_prime.c
```

3. 生成された実行ファイル (./is_prime) を起動.

```
$ ./is_prime
17
17は素数です
$ ./is_prime
9922331
9922331=3163*3137
$ ./is_prime
9922399
9922399は素数です
```

4. 色々な数値を入れて正しく動くか確認してみよう

is_prime.c の解説 (1)

▶ 7～8 行目

```
int n; /* 候補の数を順に格納するための変数 */
int divisor = 1; /* 1より大きい約数を格納する変数 */
```

約数の候補を格納する変数 `n` と、`a` の約数が見つかった時に、それを格納する変数 `divisor` を宣言. `divisor` は 1 で **初期化**.

▶ 9～15 行目

```
for (n = 2; n < a; ++n) /* 約数の候補を2から順に1ずつ増やしながら繰り返す */
{ /* 繰り返しの始まり */
    if (a % n == 0) /* aが n で割り切れるかを判定 */
    {
        divisor = n; /* aが n で割り切れるならそれを divisor として記憶 */
    }
} /* 繰り返しの終了 */
```

`for` ループを使って、1 より大きく `a` より小さい自然数で `a` が割り切れるかどうかを確認する.

is_prime.c の解説 (2)

- ▶ 9 行目の for では, 10 行目の { から 13 行目の } の **ブロック** に対する **繰り返しの条件** を, 以下のように指定している.
 - 式 1 (初期化) 約数の候補 n の初期値を 2 とする
 - 式 2 (継続条件) 約数の候補 n が a より小さい間, 繰り返す
 - 式 3 (繰り返し毎の処理) 約数の候補 n を 1 つ増やす.
- ▶ 11~14 行目では, n が a の約数である ($a \% n == 0$) 時, これを記憶するため, divisor に代入している.
- ▶ a がいずれの n でも割り切れなかった場合に限り, divisor は初期値 1 のままであることに注意されたい.

▶ 17~20 行目

```
if (divisor == 1 && a != divisor) /* 1より大きくaより小さい約数が無ければ素数 */
{ printf("%dは素数です\n", a); }
else /* 1より大きい約数があったら合成数と表示 */
{ printf("%d=%d%d\n", a, divisor, a/divisor); }
```

a が素数であれば「素数です」と表示し, そうでなければ「 $a=333*331$ 」のように因数分解して表示する.

- ▶ 17 行目の条件式は, 以下の 2 つ:

is_prime.c の解説 (3)

- ▶ divisor が 1 のままである (`divisor == 1`)
- ▶ divisor が a とは一致しない (`divisor != a`)

を論理積 (&&) で結びつけることで「a が 1 より大きく自分より小さい約数を持たない」という条件を表している.

is_prime.c の無駄を省く (1)

is_prime.c には 2 つの無駄がある.

【無駄 1】 n を $a-1$ まで調べる必要はない

- ▶ n が a の約数なら, a がもう一つの因数 $m=a/n$ で割り切れることも **同時に判る**.
- ▶ つまり, $n \leq a/n$ となる範囲, すなわち $n*n \leq a$ なる範囲についてののみ調べればよい.
- ▶ これを実装するには, is_prime.c の 9 行目を以下のように書き換える.

```
for (n = 2; n*n <= a; ++n)    /* 約数の候補を2から順に1つつ増やしながら繰り返す */
```


is_prime.c の無駄を省く (2)

【無駄 2】 1 つでも約数が見つかったら、残りの候補について調べる必要はない

- ▶ 例えば, a が偶数 (2 の倍数) なら, 3 以上の約数を持つか否かを調べることなく, a が **素数で無い** と直ちに判断できる.
- ▶ つまり, a が n で割り切れることが判り, その値を `divisor` に **保存した後** は, それ以上 `for` ループを繰り返す必要がない.
- ▶ これを実装するには, `is_prime.c` の 11~14 行目の `if` ブロックを以下のように書き換える.

```
if (a % n == 0)           /* a が n で割り切れるかを判定 */
{
    divisor = n;          /* a が n で割り切れるならそれを divisor として記憶 */
    break;               /* 1 つでも約数が見つかったら for ループを抜け出す */
}
```

具体的には, 14 行目に `break;` という行を付け加える.

is_prime.c の無駄を省く (3)

- ▶ break は ループを抜け出す 命令. 複数のループが入れ子になっている場合は, 最も内側のループを抜け出す. なお, 文として成立させるには文末に ; (セミコロン) が必要 な点に注意.

無駄を削ったサンプル・コード (is_prime_v2.c)

```
1  #include <stdio.h>          /* 標準入出力ライブラリ */
2  int main(void)              /* main 関数の引数と戻り値の定義 */
3  {                            /* main 関数の始まり */
4      int a;                  /* 整数型の変数 a を宣言 */
5      scanf("%d", &a);        /* キーボードから a に値を読み込む */
6
7      int n;                  /* 候補の数を順に格納するための変数 */
8      int divisor = 1;        /* 1より大きい約数を格納する変数 */
9      for (n = 2; n*n <= a; ++n) /* 約数の候補を2から順に1ずつ増やしながらか繰り返す */
10     {                       /* 繰り返しの始まり */
11         if (a % n == 0)      /* a が n で割り切れるかを判定 */
12         {
13             divisor = n;     /* aが n で割り切れるならそれを divisor として記憶 */
14             break;           /* 1つでも約数が見つかったら for ループを抜け出す */
15         }
16     }                       /* 繰り返しの終了 */
17
18     if (divisor == 1 && a != divisor) /* 1より大きくaより小さい約数が無ければ素数 */
19     { printf("%dは素数です\n", a); }
20     else /* 1より大きい約数があったら合成数と表示 */
21     { printf("%d=%d*d\n", a, divisor, a/divisor); }
22
23     return 0;                /* 戻り値として 0 を返す */
24 }
```

is_prime_v2_v2.c のコンパイルと実行

1. Terminal 上で ls コマンドを使い、現在の作業ディレクトリに is_prime_v2.c があるか確認する.
2. gcc コマンドを用いてソースファイルから **実行ファイル** を生成する. -o オプションを用いて、実行ファイル名を is_prime_v2 とする.

```
$ gcc -o is_prime_v2 is_prime_v2.c
```

3. 生成された実行ファイル (./is_prime_v2) を起動.

```
$ ./is_prime_v2
999999733 # 9が6桁続いた後733
999999733は素数です
$ ./is_prime # 素数と判定されるまでの時間を is_prime と is_prime_v2
999999733
999999733は素数です
```

4. 999999937, 999999929, 999999893, 999999883, 999999797 などの大きな素数に対して is_prime と is_prime_v2 の実行時間を比較してみよう.

レポート課題 III-3 (1)

レポート課題 III-3 (素数の個数と和)

入力された整数 **より小さい** 素数の個数 と **その和** を求めるプログラムを作り, その **ソースファイル** と, 以下の 4 つの整数

10, 100, 1000, 10000

に対する **計算結果** を提出せよ. ただし, 下記を満足すること.

- ▶ 提出するファイル名は B5TB9999_prime_sum.c (ソースファイル) および B5TB9999_prime_sum.txt (実行結果) とせよ.
- ▶ ソースファイルには適宜 **コメント** を記入せよ.

(続く)

レポート課題 III-3 (2)

レポート課題 III-3 の仕様 (続き)

- ▶ 計算結果は下記のような形式で記載せよ:

10 未満の素数	個数:x,	和:xx
100 未満の素数	個数:xx,	和:xxx
1000 未満の素数	個数:xxx,	和:xxxx
10000 未満の素数	個数:xxxx,	和:xxxxx

- ▶ 今回までの講義で紹介されていない C 言語の機能 (倍長整数 (long int 型, 関数, 配列など) を使ってもよい。ただし, 該当する部分でどのような処理が行なわれるのかをコメントとして記載すること

提出期限: 2015 年 7 月 2 日 (木)

参考までに,

レポート課題 III-3 (3)

20, 200, 2000, 20000

より小さい素数の個数と和は, それぞれ以下の通り:

20 未満の素数	個数: 8,	和: 77
200 未満の素数	個数: 46,	和: 4227
2000 未満の素数	個数: 303,	和: 277050
20000 未満の素数	個数: 2262,	和: 21171191

レポート III-3 の評価基準 (1)

必須要素

守られていない場合は減点

- ▶ 提出ファイル名 は適切か
- ▶ C 言語ソースファイル と 出力結果 を提出しているか
- ▶ gcc でコンパイルでき、生成したファイルを実行できるか
- ▶ 10, 100, 1000, 10000 の入力に対して 適切な出力 がされるか
- ▶ 式や処理について 十分なコメント が記載されているか

加点要素 (1)：技術の習得

- ▶ 講義で使っていない機能 の利用
 - ▶ 十分なコメント が付されている場合に限る
 - ▶ 関数, 配列, ファイル入出力などの利用

加点要素 (2)：創意工夫

(次ページに記載)

レポート III-3 の評価基準 (2)

加点要素 (2): 創意工夫

- ▶ 指定されていない数値 についての実行結果 (ただし多くても 10 個まで)
- ▶ 大きな数 (例えば 1,000,000) に対して適切に結果を表示させる
 - ▶ 何も工夫しないと 1,000,000 未満の素数の個数は 78,498 と正しく求まるが, その和が負になってしまう
- ▶ 計算時間を短縮するための工夫
 - ▶ 偶数が素数でないことは判っているので奇数に対してしか素数判定しない
 - ▶ 配列 が使えるなら Eratosthenes のふるいを使ってもよい

レポート III-3 の進め方 (1)

1. `mkdir` コマンドを用いて
~/Documents/report/Report-III/Report-III-3/ というディレクトリを作る (`-p` オプションを使うと便利).
2. Home でのマウス操作や `cp` コマンドなどで `is_prime.c` をコピーし, 1. で作ったディレクトリの下に
`B5TB9999_prime_sum.c` という名前で保存する.
3. `cd` コマンドを用いて, 現在の作業ディレクトリを
~/Documents/report/Report-III/Report-III-3/ に変更する.
4. `gedit` を用いて, `B5TB9999_prime_sum.c` を「入力された整数より小さい素数の個数と和」を求めるように変更する. その方法の一例を以下に示す:
 - ▶ まず, 「候補とする最大の整数」を格納する新たな変数 `a_max` と, 素数の個数と和を格納するための新たな変数 `num` と `sum` を導入する.

レポート III-3 の進め方 (2)

- ▶ 入力された `a_max` より小さい全ての自然数を `a` に順に代入し、「`a` が素数か否かを判定する」ループを繰り返すようにする
- ▶ そのループの中で、ある `a` が素数と判定されたら「`num` の値を 1 増やし、`sum` の値を `a` だけ増やす」処理を行なう
- ▶ `a_max` より小さい全ての整数について調べ終わったら、`num` に格納された素数の個数と `sum` に格納された素数の和を表示する。