制御構造(control structure)

いままでの program はすべて上から順番に実行されました. これでは例えば 1000 回 1 を足すという計算をさせるためには 1000 回+するよう書かないといけません. 制御構造は選択, 繰り返しを program にさせることができ, 上から順番にしか実行できなかった program の順番を自在に『制御』することができます.

if

```
#include <stdio.h>
int main(void) {
  int i = 10;
  if (i > 0) {
    // i が 0 より大きければ, {}の中に入る.
    printf("%s¥n", "i > 10 == true");
  }
  return 0;
}
```

if 文は
if (式) 文
という構造を持ち, 式が true, 真であるときに文を実行します.
さて, true / false の概念を説明します.

真偽値(boolean)及び関係演算子(relational operator)

true / false はそのとおり真偽を表す値です. if の式が true のとき実行されるといいました. これには, 関係を表す演算子が必要です.
たとえば,
10 > 100
は関係演算子>を使った式で, 数学の>と同様, a > b で b より a が大きいときに true になります. 上の式は 10 が 100 より大きいはずがないので false です.

演算子	説明	数学
>	a > b で a が b より大きいとき true	>
<	a < b で a が b より小さいとき true	<
>=	a >= b で a が b と同じもしくは b より大きいとき true	≧

<=　　a <= b で a が b と同じもしくは b より小さいとき true　　≦
==　　a == b で a と b が同じとき true　　=
!=　　a != b で a と b が違うとき true　　≠
数学の演算子で見れば結構そのままですね.

また, C 言語の世界では, 真偽値は数値です. これが 0 なら false, 0 以外なら true というふうになっています.

つまり

```
int i = 0;
if (i) {
  printf("%s¥n", "実行されない");
}
```

となります.

```
#include <stdio.h>
int main(void) {
  int i;
  scanf("%d", &i);
  if (i > 0) {
    // i が 0 より大きければ, {}の中に入る.
    printf("%s¥n", "i > 10 == true");
  }
  return 0;
}
```

scanf を使えば, 入力に応じて if 文の中を実行するかどうかを選択することができます. これで場合分けを行うことができるようになりました.

else

else を使うと, if 文をもれたものをひろうことができます.

```
#include <stdio.h>
int main(void) {
  int i;
  scanf("%d", &i);
```

```
  if (i > 0) {
    // i > 0 が true なら，この{}の中に入る.
    printf("%s¥n", "i > 10 == true");
  } else {
    // i > 0 が false なら，この{}の中に入る.
    printf("%s¥n", "i > 10 == false");
  }
  return 0;
}
```

また，何回も条件を調べたいときには else if が使えます.

```
#include <stdio.h>
int main(void) {
  int i;
  scanf("%d", &i);
  if (i > 10) {
    // i > 10 が true なら，この{}の中に入る.
    printf("%s¥n", "i > 10 == true");
  } else if (i > 0){
    // i > 0 が true なら，この{}の中に入る.
    printf("%s¥n", "i > 0 == true");
  } else {
    //  どちらも false なら.
    printf("%s¥n", "false");
  }
  return 0;
}
```

if/else 文について豆知識

ぶら下がり Block

非常に大切な if 文．実際の program では if 文を使わないということはまずありません.

いままではこう書いていました.

```
if (i > 0) {
```

```
   printf("%s¥n", "i > 10 == true");
}
```
じつは if の構文は
if (式) 文
なので,
```
if (i > 0)
    printf("%s¥n", "i > 10 == true");
```
こうかいても大丈夫です. {}のことを複文, もしくは Block といい,
```
{
   文;
   文;
   文;
}
```
という風に複数の文をまとめて一つの文(Block Statement)とします.

ただし, 書けるからといって{}を抜かすのは, 実は C 言語においては特有の問題を持ってくるので, よほど短い文でない限りは{}をつけて書くようにすべきです.

また, else の後も文なので 1 文であれば{}を省略できますが, 次のような問題を生みます.

```
if (i > 0)
   if ( i > 10)
      printf("in¥n");
else
   printf("else¥n");
```
この else は if (i > 0)と if (i > 10)のどちらにかかるのでしょうか? 実際には二番目の if 文にかかるのですが, このように書かれると最初の if 文にかかってそうに見えます. 非常に読みにくく, わかりにくい bug を生み出す温床になります.

代入演算子との取り違い

あるある bug というものが program 関連にはあって,この一つに代入演算子と関係演算子の equal を取り違えるというものがあります.

```
int i = 0;
if ( i = 20) {
```

```
   printf("実行されないとおもいきや...¥n");
}
```

上の例の場合 i==20 を=と書き間違いました. i は 0 のつもりなので 0==20 は false で printf は実行されないつもりで書いていますが, これは実行されてしまいます. なぜなら=で i に 20 が代入されて, i = 20 の部分の値は 20 だからです. 0 以外は true という C 言語の世界ではこれは true です.

これはいろいろ言われていて, 一部では,

```
int i = 0;
if ( 20 == i ) {
   printf("実行されないとおもいきや...¥n");
}
```

とかくといいといわれた時期もありました. これなら, もし programmer が間違って,

```
int i = 0;
if ( 20 = i ) {
   printf("実行されないとおもいきや...¥n");
}
```

と書いても, 20 に i を代入することはできないので, compile でエラーになってわかるといわれたからです.

しかし現在, これはすでに時代遅れなやり方で, C 言語で const が使える今使うべきではありません. const については後々解説します. とりあえず, 上のようなやり方は, 心の隅にとどめて, 実用しないようにしましょう.

論理演算子(logical operator)

a	b	a&&b	a||b	!a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	false
false	false	false	false	true

演算子	説明	俗にいう
&&	a && b で a と b が true のとき true. それ以外で false	and
||	a || b で a と b がどちらか true のとき true, どちらも false で false	or
!	!a で true と false がひっくり返る	not

logical and / logical or / logical not といいます.

論理積とか論理和とかいいます.

計算機基礎とか, 情報数学概論とかやったことある方は, あれかと思うかもしれません.

これを使うと.

```
int i = 12;
if ( i > 0 && i < 20 ) {
   printf("i は 0 より上 20 より下¥n");
}
```

とできます.

論理演算子の短絡評価

論理演算子の logical and と logical or は短絡評価という性質があります.
例えば,

```
true || (?);
```

or は片方が true ならもう true 確定なので, (?)のほうは見なくてもこの条件式は true 確定なわけです. また,

```
false && (?);
```

and は両方 true で true なので, 片方が false ならもう false 確定です. これも(?)のほうはみなくても確定してます.

このような場合, 高速化のため program は(?)を実行しません. これを短絡評価といいます.

while/switch/for

if 文で分岐ができるようになりました. この後同じく分岐の switch, 繰り返しの while と for をやります.

switch

switch 文で広範囲な分岐を行うことができます.

```
#include <stdio.h>

int main(void) {
```

```
    int ch;
    ch = getchar();
    switch (ch) {
        case 'a':
            printf("input: a¥n");
            break;

        case 'b':
            printf("input: b¥n");
            break;

        case 'c':
            printf("input: c¥n");
            break;

        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            printf("input: decimal %c¥n", ch);
            break;
    }
    return 0;
}
```

getchar で文字を 1 字読み込み, switch で分岐しています.
int 型に代入しているので注意.

case 'a'と書いたとき,

```
if (ch == 'a') {
```

```
  // case の中身
}
```
が行われ, ch == 'a'が true なら case 'a':の中に入ります.

break は switch 文を抜けるための文で, case 'a':の最後に書いています. これを書いていないと, case 'a':の最後で switch 文を抜けずに, case 'b'のほうへ突入します.(switch の fall through. bug の温床). このため, 特別な理由がないときはちゃんと break をつけるよう気をつけないといけません.
ただし, この面倒なことが役立つ場合があって, 上の case '0':には break がなく, case '1':に突入しています. これは, case '0':から case '9':まで, どれと true になってもみんな
printf("input: decimal %c", ch);
break;
に辿り着くようになっています. break していないからです.
```
default:
        文
        break;
```
とすれば case に無い値の場合にこれが実行されます.

while

繰り替えし処理, while です. これを使えば, 今まではずっと逐一書いてきた処理が, 繰り返しを見つけることによって簡潔かつ永続的に実行ができます.

```
while (exp)
  statement;
```
exp が true である間, while 文は statement を実行します.

単純な話,
```
while (true) {
  printf("loop¥n");
}
```
はみんな大好き無限 loop です.終わりません.

```
#include <stdio.h>
```

```
int main(void) {
   int ch;
   while ((ch = getchar()) != EOF) {
      if (ch == 'Q' || ch == 'q') {
            printf("EXIT¥n");
            break;
      } else if (ch != '¥n') {
          printf("%c¥n", ch);
      }
   }
   return 0;
}
```

getchar で読み込み, ch に代入しています. getchar は読み込みが終了すると(終端文字に出会うと)EOF を返します. これは End Of File です.

if 文の中で, ch が'Q'か'q'であったときに printf("EXIT¥n")して break しています. この break は一番近い loop を抜けることができて, この場合は while 文を抜け出し, 終了します.
それ以外だったら文字の表示です.
文字を入力するときに Enter を押していますね. これで実は入力文字に「改行」が入っています. ¥n です. ここで ch != '¥n'とすることで, 改行だけの入力を無視しています.

カウンタ

while で繰り返しを行う場合, カウンタという概念が重要になります.
n 回繰り返すというのはどうすればいいのかということに対する一般的な解答です.

increment / decrement
インクリメントとデクリメントというものがあります.

```
#include <stdio.h>
int main(void) {
   int a = 20, b, c;
   b = --a;
   c = a--;
   printf("a = %d¥nb = %d¥nc = %d¥n", a, b, c);
```

```
    return 0;
}
```
4,5 行目がデクリメントです. a = 18, b = 19, c = 19 と表示されたら大丈夫です.
前置/後置デクリメントの 2 つがあります.

int a = 20, b, c;
この時点では a = 20, b, c は不定です.

b = --a;
--a が評価されて, まず a の値が−1 されます. 19 ですね. で, その 19 が b に代入されます.

c = a--;
a--が評価されて, まず a の値そのままの 19 が返ります. で, その 19 が c に代入されます.
そしてそれが終わった後, a の値は−1 されて 18 になります.

このように後置の場合は評価時ではなくその後に−1 されます.
インクリメントはこれらのプラス版です.

以下の program の意味がわかれば大丈夫です.
```
#include <stdio.h>

int main(void) {
    int n = 10;
    while (n--) {
        printf("%d¥n", n);
    }
    return 0;
}
```
while の中身が true かどうか, 確認するたびに n の数が−1 されていきますね. すると正の数ならいずれは 0 になるはずです. 0 は false 評価されますので, 結果 while 文がとまります.

この場合, 9 から 0 までの数, つまり loop 自体は 10 回ったのを確認できると思います.

while (n--) {
で, n==10 のとき n--は 10 を返すので true です. 後置なので, それが終わったら n の値は 9 になっています. なのではじめに表示される printf("%d¥n", n); は 9 でした.

これの n が 1 になったときを考えます.

while (n--) {
n−−でまず 1 が返るので loop は続行ですね. で loop 内で n は 0 になります.

で, 次の loop のときに n−−が 0 なので while 文が終了します.

これをもし前置でやるとどうなるでしょうか.

```
#include <stdio.h>

int main(void) {
   int n = 10;
   while (--n) {
      printf("%d¥n", n);
   }
   return 0;
}
```

9 から 1 までしか表示されません. これは n が 1 のとき, −−n では 0 になってしまい, n = 1 の時点で打ち切られるからです.
ですので, while で一般に n 回 loop するというときには,

```
#include <stdio.h>

int main(void) {
   int n = 10;
   while (n--) {
      // 内容
   }
   return 0;
}
```

のようにします. for 文を使うとよりわかりやすい構文になります.

繰り返し処理を使う

ようやく分岐/繰り返しを覚えることで, 最低限の program を書く準備が整いました.

とりあえず, Fibonacci 数を求めましょう. (こと programmer がはじめに何か書くとき, Fibonacci 数を求める/前にやった Hello World を表示する/この後出てくる FizzBuzz を解くのどれかをやる傾向があります...)

Fibonacci 数とは,
n 番目のフィボナッチ数を F_nで表わすと $F_o = 0, F_1 = 1, F_{n+2} = F_n + F_{n+1}$ (n $\geq$ 0) となります.
フィボナッチ数列
0, 1, 1, 2, 3, 5, 8, 13, 21,34,55, ...

前二つの数字を足したものが次の数字になるという数列です.
n=0 番目の値は 0 として 0 開始にします.

この n 番目の Fibonacci 数を求めるというものです. 是非やってみてください.

For
for

繰り返し処理を行うには少し while は使いにくくないでしょうか?
while は一般に決まった数の繰り返しを行うというより, 条件が一致するまで実行し続けるといった無限ループ前提のような面があります.
決まった数字の繰り返しには, for を使うことでより簡潔に記述することができます.

Ex.
まず 1 から 10 までの積の合計, つまり 10!の計算をしてみます.

```
#include <stdio.h>

int main(void) {
   int i, n = 10, res = 1;
   for (i = 1; i <= n; i++) {
      res *= i;
```

```
    }
    printf("%d¥n", res);
    return 0;
}
```
for は次のような syntax を持っています.

for(初期化; 条件; インクリメント) 文;
example の場合, i = 1 にして初期化, i <= n の間 for の文が実行され, 最後に i++されてまた条件を調べます. 実はこれは while でそのまま書くことができます.

for の
```
int i, n = 10, res = 1;
for (i = 1; i <= n; i++) {
    res *= i;
}
```
は while でかくと,
```
int i, n = 10, res = 1;
i = 1;
while (i <= n) {
    res *= i;
    i++;
}
```
とまったく同じ挙動をします. じゃあ何で for があるんやって話ですが, for 文では i=1 がカウンターの初期化, i <= n が条件, i++が条件を変化させる部分というのがすっぱり分かれてわかりやすく書かれています.

また, for では n 回実行というのが非常に直感的に書くことができます. 10 回 printf をしてカウントを出してみましょう.

```
#include <stdio.h>

int main(void) {
    int i, n = 10;
    for (i = 0; i < n; ++i) {
        printf("%d¥n", i);
    }
```

```
    return 0;
}
```

0 から 9 まで, 10 回 printf が実行され数値が表示されました. while (n--)と比べてわかりやすく, かつ i の数字を使って今何回目かを 0 から始める形でカウントできます.

例題

前のほうで話題に出た, FizzBuzz というものをやってみます.

FizzBuzz とは?

programmer が大好き
fibonacci くらい大好き
なものです. 具体的には,

1 から 100 までカウントしていくときに,
3 の倍数なら Fizz
5 の倍数なら Buzz
3 の倍数かつ 5 の倍数なら FizzBuzz
それ以外ならその数字を表示
というものです. 出力例としては,

1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz

となります.是非やってみてください.

do-while

while の変形として, do while というものがあります.

```
#include <stdio.h>

int main(void) {
   int i;
   do {
      printf("input integer (more than 0): ");
      scanf("%d", &i);
   } while (i < 0);
   printf("received value is %d¥n", i);
   return 0;
}
```

これは正の数が入力されるまで入力を繰り返すという場合の program です.

do 文 while(条件式);
となっており, まず文を実行, その後条件式を調べ, true ならまた文を実行するという風に loop します.一回目は必ず実行されます.

使用例としては上のように, 一度値を受け取ってそれが条件にあっていなければ繰り返すといったような何かしら一度実行してみてからじゃないと loop するかどうかわからないものに関して使います.

http://www.soft-forum.net/2010/programming/ により詳しく載っています.