飛翔ソフトウェア

新人研修

C言語基礎編

|  |  |  |
| --- | --- | --- |
| 日付 | 変更内容 | 変更者 |
| 2021/02/03 | 新規作成 | 庭 |
| 2021/02/09 | C言語基礎の抜粋を記載 | 庭 |
| 2021/03/30 | 間違いの修正と不足知識の追加。 | 庭 |
| 2023/01/13 | VisualStudio依存の部分を変更。その他誤記修正 | 庭 |
| 2023/01/16 | 課題9の追加 | 庭 |

# 研修の目的

飛翔ソフトウェアが受注している仕事にはC言語での開発が多く、必須スキルとなります。また、C言語の基礎を学ぶことで他言語の習得もスムーズになります。

# はじめに

まずは社内の参考書を読み、C言語の学習をしましょう。

とはいっても、膨大な参考書を1から10まで覚えるのは簡単ではないと思いますので、まずは抜粋したものをここに記載します。

## C言語の構造

C言語のプログラムは、関数が集まって作られています。

関数は以下の構造になっています。

|  |
| --- |
| 戻り値の型 関数名（引数の型 引数1, 引数の型 引数2,・・・,引数の型 引数n)  {  処理  return 戻り値; ← 省略されることもある  } |

型とは、変数や関数の戻り値などで使用する、プログラムで実際に扱う「値」の形式のことです。

|  |  |
| --- | --- |
| 型名 | 説明 |
| char | １バイトの符号付整数。ASCIIコードといった文字コードに使用 |
| unsigned char | １バイトの符号なし整数 |
| short | 2バイトの符号付整数 |
| unsigned short | 2バイトの符号なし整数 |
| long | 4バイトの符号付整数 |
| unsigned long | 4バイトの符号なし整数 |
| int | 2または4バイトの符号付整数。(コンパイラに依存) |
| unsigned | 2バイトまた4バイトの符号なし整数。(コンパイラに依存) |
| float | 4バイトの単精度浮動小数点実数 |
| double | 8バイトの倍精度浮動小数点実数 |

関数名とは、文字通り、関数の名前のことです。

引数とは、関数に渡す数値の種類のことです。引数は、複数定義することが可能であり、その場合は、間を,で区切ります。また、省略することも可能です。

関数は、渡された数値を元に計算を行って結果を戻り値として返すことが出来ます。

関数を終了させるには、return(リターン)文を使用します。

引数や戻り値がない場合、情報がないことを表すvoid(ボイド)にします。

C言語では、mainという名前の関数が、1番始めに動作すると決められています。

ですので、以下のコードを作成すると"何もしない"プログラムを作成することができます。

|  |
| --- |
| int main(void)  {  return 0;  } |

## インデント

return文が右にずれているのは、インデントと呼ばれる書き方です。

インデントは、階層的な構造を表現するために使われます。

C言語では、{} で囲まれた中の文は、右に少しずらすことが慣習となっています。

右へのずらしには、タブ(Tab)キーを使用するのが慣習です。

## コメント

C言語には、プログラムに説明を入れておく機能があり、これをコメントと呼びます。

C言語では、/\* \*/ で挟むことで、その間をコメントにすることが出来ます。

このコメントは、プログラムの実行には一切影響を与えません。

## 文字列の表示

C言語で文字列を表示するには、printf(プリントエフ)関数を使います。

printf関数は、次のようにして使います。

|  |
| --- |
| printf("文字列"); |

"" で囲まれていれば、文字列として扱われます。

printf関数は、C言語自体の機能ではありません。

言い換えるならば、C言語のコンパイラは、printfという関数のことを全く知りません。

従って、ただprintf関数を記述しただけでは、動かないのです。

動作させるには、コンパイラにprintf関数の説明書を読ませなければなりません。

C言語には、説明書を渡すための特別な命令が用意されています。

それは、#include(インクルード)疑似命令です。

printf関数の説明書は、stdio.h と言うファイルです。

つまり、次のようなプログラムを追加すれば、printf関数を使えるようになります。

|  |
| --- |
| #include <stdio.h>  int main(void)  {  printf("HelloWorld\n");  return 0;  } |

\nは、改行を表す特殊な文字です。この文字があると、文字列はそこで改行され、続く文字は次の行から改行されます。

この改行記号のように、\マークで始まる文字を、エスケープシーケンスと言います。

|  |  |
| --- | --- |
| 記号 | 意味 |
| \b | バックスペース |
| \n | 改行 |
| \t | タブ |
| \0 | ヌル（null）文字 |

printf関数は、数値を表示する機能も持ち合わせています。

printf関数で数値を表示するには、出力変換指定子を使用します。

これは、文字列の中に埋め込んで使われる、記号の一種です。

|  |  |  |
| --- | --- | --- |
| 書式 | 意味 | 使用例 |
| %d | 整数値を10進数で表示する | 1, 12, 30, -4, 5 + 5 |
| %f | 実数値を10進数で表示する | 0.15, -4.2, 2.21, 1.3 + 3.8 |
| %c | 文字。ASCIIコードで表示された文字列が表示される。 | 'A', 'b', '8' |
| %s | 文字列。文字列をそのまま表示できる。 | "ABC", "あいうえお" |

|  |
| --- |
| #include <stdio.h>  int main(void)  {  printf("%d円\n",100);  return 0;  } |

## 四則演算子

|  |  |  |
| --- | --- | --- |
| C言語での記号 | 数学での記号 | 機能 |
| + | + | 加算(足し算) |
| - | - | 減算(引き算) |
| \* | × | 乗算(掛け算) |
| / | ÷ | 除算(割り算) |
| % | … | 剰余算(割り算の余り) |

|  |
| --- |
| #include <stdio.h>  int main(void)  {  printf("%d\n",10 + 3);  printf("%d\n",10 - 3);  printf("%d\n",10 \* 3);  printf("%d\n",10 / 3);  printf("%d\n",10 % 3);  return 0;  } |

整数計算の結果は四捨五入ではなく切り捨てとなります。

数式での優先順位は、数学と同じです。乗算や除算は、加算や減算よりも先に計算されます。優先順位を変えたい時には、() をつけるのも、数学と全く同じです。

ただし、数学では、2重の () には {} を使いますが、C言語では違います。C言語では、何重の () であっても、() の記号しか使いません。

整数は、3種類の書き方を使うことが出来るようになっています。それは、10進数、8進数、16進数の3種類に分かれています。

|  |  |  |
| --- | --- | --- |
| 数字 | 10進数 | 先頭に0をつけない数は、10進数として扱われます。  例えば、100､25､68､71､19023 などは10進数です。 |
| 0数字 | 8進数 | 先頭に0をつけた数は、8進数として扱われます。  例えば、0152､027､0756､030303 などは8進数です。 |
| 0x数字 | 16進数 | 先頭に 0x をつけて表します。  例えば、0xFF､0xA7､0x912C､0xABCD などは16進数です。 |

## 変数

数値を記憶するメモリに名前をつけて管理する方法です。

|  |
| --- |
| 型名 変数名; |

変数の宣言は、{}ブロックで囲まれた先頭（関数の中）か、関数の外で行います。

関数の中で宣言された変数は、ローカル変数といって、関数の中でしか使用できません。

関数の外で宣言された変数は、グローバル変数といって、どこでも使用できます。

ローカル変数は、定義されている関数が違えば同名のものを定義しても、それぞれ別のものとして扱われます。それに対し、グローバル変数の場合は、プログラム全体でただ一つしかありません。したがって、名前の重複は許されません。

|  |
| --- |
| #include <stdio.h>  /\* グローバル変数宣言 \*/  int global\_value = 1;  int main(void)  {  /\* ローカル変数宣言 \*/  int value;  /\* 代入 \*/  value = 10;  global\_value = 20;  /\* 表示 \*/  printf("%d %d\n", value, global\_value);  return 0;  } |

## 配列

変数を用いて大量のデータを扱う場合、配列を用います。

|  |
| --- |
| 型名 配列名[要素数]; |

このような宣言をした場合、

|  |
| --- |
| int value[5] = {10, 20, 30, 40, 50}; |

以下のような配列ができます。配列の初期化は{}で囲んだ中に順番に,で区切って数値を並べます。配列の要素数よりも少ない場合、残りには全て0が代入されます。

また、要素数を省略すると初期化で指定した数値の個数だけの要素数が確保されます。

ここで[]の中に書いてある数字を、添字（そえじ）と言います。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| value[0] | value[1] | value[2] | value[3] | value[4] |
| 10 | 20 | 30 | 40 | 50 |

例えば、

|  |
| --- |
| value[2] = 10; |

とすると、このように変わります。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| value[0] | value[1] | value[2] | value[3] | value[4] |
| 10 | 20 | 10 | 40 | 50 |

### 文字列配列

文字列の配列の場合、char型の変数の成分には、それぞれ文字列のコードが入っています。そして、文字列の最後には、必ず¥0が入っています。エスケープシーケンスの部分でも説明しましたが、この文字を、NULL(ヌル)文字と言います。値としては0に等しいのですが、文字として使用する場合、とくにこう呼びます。

そのため、配列変数に文字列を作る場合は、最低限、文字数+1の成分が必要になります。また、配列の途中に¥0があれば、そこで文字列は終了になります。

|  |
| --- |
| char str1[4] = { "a", "b", "c", "\0"};  char str2[] = "Hello"; |

|  |  |  |  |
| --- | --- | --- | --- |
| str1[0] | str1[1] | str1[2] | str1[3] |
| "a" | "b" | "c" | \0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| str2[0] | str2[1] | str2[2] | str2[3] | str2[4] | str2[5] |
| "H" | "e" | "l" | "l" | "o" | \0 |

### 二次元配列

配列には複数の添字をつけることができます。2つの添え字をつけることで二次元の配列を作ることが可能です。

|  |
| --- |
| int a[2][3] = {{1,2,3},{4,5,6}}; |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 | 2 |
| 0 | a[0][0]  1 | a[0][1]  2 | a[0][2]  3 |
| 1 | a[1][0]  4 | a[1][1]  5 | a[1][2]  6 |

## 変数のアドレス

変数はコンピュータのメモリ内にあるため、アドレスが存在します。

例えば、aという変数があると、&aとすることにより、変数のアドレスを取得することができます。

これにより、変数の値がメモリ中のどこに存在するのかを知ることができます。

|  |
| --- |
| #include <stdio.h>    int main(void)  {  int a = 100;  double b = 123.4;  float c = 123.4f;  char d = 'a';  printf("aの値は%d、大きさは%dbyte、アドレスは0x%x¥n", a, sizeof(a), &a);  printf("bの値は%f、大きさは%dbyte、アドレスは0x%x¥n", b, sizeof(b), &b);  printf("cの値は%f、大きさは%dbyte、アドレスは0x%x¥n", c, sizeof(c), &c);  printf("dの値は%c、大きさは%dbyte、アドレスは0x%x¥n", d, sizeof(d), &d);  } |

sizeof(サイズオブ)演算子は、変数や型のメモリのサイズを取得する演算子です。

()の中に変数や型を入れれば、 そのサイズをバイト単位で取得することが可能です。

## キーボードからの入力

キーボードから入力する時には、その為の関数を使用します。

Ｃ言語には、scanf(スキャンエフ)関数が用意されています。

scanf関数は、次のようにして使います。

|  |
| --- |
| scanf("入力変換指定子", 変数のアドレス); |

入力変換指定子とは、入力された数字をどのような数値に変換するかを表す文字です。

printf関数で使用した出力変換指定子と、ほぼ同じように使用出来ます。

変数のアドレスには、入力されたデータを記憶しておく変数のアドレスを指定します。

## 条件判断

C言語には、条件による判断を行う文として、if(イフ)文が用意されています。

if文は、2つの数値の値を比較して、その結果を元に処理分けを行います。

|  |
| --- |
| if (条件式) {  文;  } |

条件に一致した時だけでなく、条件に一致しなかった時にも文を実行させたい場合、

if文と一緒に、else(エルス)文を使用することが出来ます。

|  |
| --- |
| if (条件式) {  条件に一致した時の処理;  } else {  条件に一致しなかった時の処理;  } |

C言語には、比較専用の演算子が用意されています。

|  |  |  |
| --- | --- | --- |
| 記号 | 真になる | 偽になる |
| == | 2つの値が等しい | 2つの値が等しくない |
| != | 2つの値が等しくない | 2つの値が等しい |
| < | 左の値が右より小さい | 左の値が右より小さくない |
| > | 左の値が右より大きい | 左の値が右より大きくない |
| <= | 左の値が右以下 | 左の値が右以下ではない |
| >= | 左の値が右以上 | 左の値が右以上ではない |

if文では、1度に1つの条件での判定しか出来ないのですが、

論理演算子を使用することで、複数条件での判定が可能になります。

|  |  |  |  |
| --- | --- | --- | --- |
| 記号 | 意味 | 真になる | 偽になる |
| && | かつ(AND) | 右と左の条件が両方真 | 右と左の条件のどちらか片方でも偽 |
| || | または(OR) | 右と左の条件のどちらか片方でも真 | 右と左の条件が両方偽 |
| ! | 否(NOT) | 条件が偽 | 条件が真 |

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int number = 0;  scanf("%d", &number);  if (number >= 8 && number <= 12) {  printf("8～12の間です。\n");  } else {  printf("8～12の間ではありません。\n");  }  return 0;  } |

## 場合分け

switch文を用いることで場合分けを表現することができます。

switch文は、指定された条件式の値と同じ値のcaseへジャンプします。

ジャンプ先ではcase以下の文を実行し、break(ブレイク)文を見つけたら、

switch文の後で囲っている {} の中から抜け出します。

defaultには、他のcaseに当てはまる数値が無かった場合にジャンプします。

|  |
| --- |
| switch (条件式) {  case 数値:  実行文;  break;  case 数値:  実行文;  break;  default:  実行文;  break;  } |

## 繰り返し動作

C言語では、回数の決まっている繰り返しには、for(フォー)文を使用します。

|  |
| --- |
| for (初期化; 条件式; 更新) {  繰り返す文;  } |

初期化とは、カウント変数の初期化を行うための文です。

ここに書かれた式は、最初に1回だけ実行されます。

条件式とは、ループの終了条件を設定するための文です。

ここに書かれた式の値が真の間は、繰り返す文を実行し続けます。

更新とは、カウント変数の更新を行うための文です。

ここに書かれた式は、繰り返しを行う文を実行した後に実行されます。

for文を途中で終わらせてしまうことも出来ます。それには、break文を使用します。

繰り返し途中にエラーが発生した、など、繰り返し回数が完了する以前に終了させるために使用します。

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int i;  for (i = 1 ;i <= 10; i++) {  printf("%d\n", i);  if (i == 3) {  break;  }  }  return 0;  } |

回数が決まっていない繰り返し文には、while（ホワイル）文を使用します。

while文は、一般に、次のような書き方で使用します。

|  |
| --- |
| while (条件式) {  繰り返す文;  } |

|  |
| --- |
| #include <stdio.h>  int main(void)  {  unsigned long money = 1;  unsigned int month = 1;  while (money < 1000000) {  printf("%02d 月目 : %d 円\n", month, money);  money \*= 2;  month++;  }  printf("%02d 月目 に %d 円となり、100万円を超える。\n", month, money);  return 0;  } |

条件式を後から判定するwhile文もあります。

|  |
| --- |
| do {  繰り返す文;  } while (条件式); |

文を実行してから、条件式の判定が行われるため、do～while文では1回は文が実行されます。do～while文は通常のwhile文で置き換えられることが多いため、繰り返し文としての用途はあまりありません。

do～while文はインデントが深くならないようにする場合に使用されることが多いです。（詳細は後述の「インデントが深くなってしまったら」を参照してください）

### インクリメント・デクリメント

for文では、インクリメントおよび、デクリメントという処理をよく行います。

インクリメントは変数の値を1増加させる処理です。デクリメントとは、インクリメントの反対で、変数の値を1減らす処理です。この処理を表にまとめると、以下のようになります。

|  |  |  |  |
| --- | --- | --- | --- |
| 演算子 | 呼び名 | 意味 | 該当する演算 |
| i++ | インクリメント  （後置：こうち） | 変数の値を1増加させる | i=i+1;  i+=1; |
| ++i | インクリメント  （前置：ぜんち） | 変数の値を1増加させる |
| i-- | デクリメント  （後置：こうち） | 変数の値を1減少させる | i=i-1;  i-=1; |
| --i | デクリメント  （前置：ぜんち） | 変数の値を1減少させる |

前置の場合、演算の修了後、左辺の値に代入されるのに対し、

後置の場合、代入の後に演算が行われます。以下の処理を行うと、

|  |
| --- |
| int a1=1, b1=1;  int a2, b2;  a2 = a1++;  b2 = ++b1; |

a1は2、b1は2、a2は1、b2は2

となります。

## プログラムの部品化

1度作ったプログラムを、もう一度使いたいと思った場合、1番単純なのは、コピーしてmain関数内に貼りつけることですが、プログラムが大量にあった場合には、煩雑になってしまい、わけのわからないプログラムになってしまうことは間違いありません。

以前に作成したプログラムを再び使うことを再利用と呼び、関数として再利用することを部品化と呼びます。

|  |
| --- |
| #include <stdio.h>  /\* プロトタイプ宣言 \*/  int print\_sum(int number1, int number2, int number3);  int main(void)  {  print\_sum(1, 2, 3);  print\_sum(4, 5, 6);  print\_sum(7, 8, 9);  return 0;  }  int print\_sum(int number1, int number2, int number3)  {  printf("合計を表示します：%d\n", number1 + number2 + number3);  return 0;  } |

コンパイラは、新しい関数を発見すれば、自動的に認識してくれますが、新しい関数が使えるのは、その関数よりも後に発見された関数の中だけです。

先の例では、まずmain関数を作り、次にprint\_sum関数を作りました。

この場合、main関数を解析している段階では、print\_sum関数は発見されておらず、従って、main関数の中では、print\_sum関数を使うことは出来なくなってしまいます。

これを解決する方法の1つは、print\_sum関数の概要を先に書いてしまう方法です。

この関数の概要をプロトタイプ宣言と呼んでいます。

## ポインタ

C言語には、アドレスを入れることを前提とした変数が存在します。それをポインタ変数と言います。先頭に\*をつけると、その変数がポインタ変数であることを示すことができます。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 通常の変数 | ポインタ変数 | 解説 |
| 宣言 | int a; | int \*p; | ポインタ変数は、変数の先頭に\*をつける |
| 値 | a | \*p | ポインタ変数で値を示すには、先頭に\*をつける必要がある。 |
| アドレス | &a | p | 通常の変数は、値を入れるが前提だが、ポインタ変数はアドレスを入れる |

例

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int a = 100;  int \*p = NULL;  p = &a;  \*p = 300;  return 0;  } |

(1)変数の宣言

char a = 100;

char \*p = NULL;

メモリ 変数

|  |  |  |
| --- | --- | --- |
| 0xffff0000 | 100 | a |
| 0xffff0001 |  |  |
| 0xffff0002 |  |  |
| 0xffff0003 |  |  |
| 0xffff0004 |  |  |
| 0xffff0005 |  |  |
| 0xffff0006 |  |  |
| 0xffff0007 |  |  |
| 0xffff0008 | 0 | p |
| 0xffff0009 | 0 |  |
| 0xffff000a | 0 |  |
| 0xffff000b | 0 |  |
| 0xffff000c |  |  |
| 0xffff000d |  |  |

ポインタ変数pにNULL(ヌル)を入れて初期化しています。NULLは、C言語で標準的に用いられる定数で、数値でいえば0を意味しますが、通常ポインタ変数はNULLで初期化するという習慣になっています。

※本例では、32bitCPUとします（アドレスは32bit=4byte）

(2)ポインタ変数にアドレスを格納

p = &a;

メモリ 変数

|  |  |  |
| --- | --- | --- |
| 0xffff0000 | 100 | a |
| 0xffff0001 |  |  |
| 0xffff0002 |  |  |
| 0xffff0003 |  |  |
| 0xffff0004 |  |  |
| 0xffff0005 |  |  |
| 0xffff0006 |  |  |
| 0xffff0007 |  |  |
| 0xffff0008 | 0x00 | p |
| 0xffff0009 | 0x00 |  |
| 0xffff000a | 0xff |  |
| 0xffff000b | 0xff |  |
| 0xffff000c |  |  |
| 0xffff000d |  |  |

aのアドレスを格納することにより、pは、aとして振舞うことが可能になります。

そのため、値である\*pは、aと同じ値をとります。

(3)ポインタ変数にアドレスを格納

\*p = 300;

メモリ 変数

|  |  |  |
| --- | --- | --- |
| 0xffff0000 | 300 | a |
| 0xffff0001 |  |  |
| 0xffff0002 |  |  |
| 0xffff0003 |  |  |
| 0xffff0004 |  |  |
| 0xffff0005 |  |  |
| 0xffff0006 |  |  |
| 0xffff0007 |  |  |
| 0xffff0008 | 0x00 | p |
| 0xffff0009 | 0x00 |  |
| 0xffff000a | 0xff |  |
| 0xffff000b | 0xff |  |
| 0xffff000c |  |  |
| 0xffff000d |  |  |

\*pに値を入れると、pには実体は無く、aをさしていることから、その値はaに反映されます。

## 構造体

プログラムがある程度複雑になってくると、一つの概念に対して複数の変数が割り当てられることがあります。例えば学校で学生のデータベースを作る時、その学生の学生番号、名前、年齢 といったデータをひとまとめにして扱う事になります。

そういったプログラムを作る時、関連する変数がバラバラになっていると、取扱が非常に不便です。そこで便利なのが、構造体（こうぞうたい）という概念です。構造体とは、複数の変数をひとまとめにするものです。

たとえば、学生番号を表す整数型の変数id、名前を表す文字列name、年齢を表す整数型変数ageをひとまとめにして構造体にすると、以下のようになります。

|  |
| --- |
| struct student {  int id; /\* 学生番号 \*/  char name[256]; /\* 名前 \*/  int age; /\* 年齢 \*/  }; |

structが、構造体を表すキーワードであり、そのあとのstudentが構造体の名前になります。{}の中に、ひとまとめにする変数を定義します。最後は;（セミコロン）で終了します。

この構造体を実際に使用するには、

|  |
| --- |
| struct student data; |

とすると、dataという名前の構造体変数を定義できます。

構造体の成分の変数のことを、メンバと言います。このメンバにアクセスするには、通常以下のように行います。

(構造体変数名).(メンバ)

構造体のポインタの場合は、"->"（アロー演算子)を用います。

|  |  |  |
| --- | --- | --- |
|  | 通常の構造体 | 構造体のポインタ |
| 定義 | struct student data; | struct student \*pdata; |
| メンバのアクセス | data.id  data.name  data.age | data->id  data->name  data->age |

## プリプロセッサ命令

ソースコードをコンパイルする前に行われる前処理をプリプロセスといいます。

このプリプロセスを行なうプログラムのことをプリプロセッサと呼び、プリプロセッサへの命令をプリプロセッサ命令といいます。

プリプロセッサ命令には主に以下があります。

・マクロの定義(#define)

・ファイルの読込み(#include)

・条件付コンパイル(#if ～ #elif ～ #else ～ #endif)

### #define

#defineはソースファイル内の文字列置き換え指示です。

|  |
| --- |
| #define マクロ名 置き換え後の文字列 |

例)

|  |
| --- |
| #include <stdio.h>  #define INPUT\_NUM\_MAX (100)  #define INPUT\_NUM\_MIN (1)  void main(void)  {  int num = 0;  printf("%dから%dの数字を入力してください。\n", INPUT\_NUM\_MIN, INPUT\_NUM\_MAX);  printf(":");  scanf("%d", &num);  if (num > INPUT\_NUM\_MAX) {  printf("%dより大きい値は入力できません", INPUT\_NUM\_MAX);  } else if (num < INPUT\_NUM\_MIN) {  printf("%dより小さい値は入力できません", INPUT\_NUM\_MIN);  } else {  printf("%d", num);  }  } |

#defineによるマクロは、引数付きにすることができます。これは関数マクロと呼ばれています。

|  |
| --- |
| #define マクロ名(引数) 処理 |

複数行になる場合、改行の前に\(バックスラッシュ)を記述することで次の行に命令の続きを書くことができます。

|  |
| --- |
| #define マクロ名(引数) 処理1 \  処理2 |

例)

|  |
| --- |
| #include <stdio.h>  #define INPUT\_NUM\_MAX (100)  #define INPUT\_NUM\_MIN (1)  #define INPUT\_CHECK(a) ((((a) <= INPUT\_NUM\_MAX) && ((a) >= INPUT\_NUM\_MIN)) ? 1 : 0)  #define TEST\_STR "defineは文字列の置換もできます\n"  #define PRINT\_EVEN\_OR\_ODD(a) \  do { \  if (a % 2) { \  printf("奇数\n"); \  } else { \  printf("偶数\n"); \  } \  } while (0)  void main(void)  {  int num = 0;  printf(TEST\_STR);  printf("%dから%dの数字を入力してください。\n", INPUT\_NUM\_MIN, INPUT\_NUM\_MAX);  printf(":");  scanf("%d", &num);  if (INPUT\_CHECK(num)) {  printf("%d", num);  PRINT\_EVEN\_OR\_ODD(num);  } else {  printf("入力値が不正です");  }  } |

ちなみに…

|  |
| --- |
| #undef マクロ名 |

と記述すると、「#define」で定義した定数やマクロを無効にすることができます。

### #include

#includeはファイル名で示されたファイルの中身をその部分に挿入します。

|  |
| --- |
| #include <ファイル名>  #include "ファイル名" |

システム標準の場所にあるファイルは<>、独自の場所にあるファイルは””で囲みます。

#include 命令によって読み込まれるファイルは一般にヘッダファイルと呼ばれます。ヘッダファイルの拡張子は .h です。

### #if ～ #elif ～ #else ～ #endif

#ifなどを使用することでコンパイルする対象を制御することができます。

|  |
| --- |
| #if 条件  ソースコード;  #endif  #if 条件  ソースコード1;  #else  ソースコード2;  #endif  #if 条件1  ソースコード1;  #elif 条件2  ソースコード2;  #else  ソースコード3;  #endif |

#ifdef、#ifndefで定数またはマクロが定義されている（いない）のときコンパイル対象を制御することができます。

#ifdef 定数またはマクロ

は

#if defined(定数またはマクロ)

でも同じ意味です。

## goto

goto文は指定された場所(ラベル)に無条件にジャンプ（移動）する文です。

ラベルは変数や関数名同様に名前を付けて、名前のあとにコロン 「 : 」をつけます

|  |
| --- |
| goto ラベル名;  処理;  処理;  ラベル名:  処理; |

## 列挙型

列挙型は定数のリストを定義することができます。

列挙型を作成するにはenumキーワードを使用します。

{ } の内側には、列挙子を並べます。列挙子は、列挙型に含まれる定数の名前（列挙定数）のことです。

|  |
| --- |
| enum タグ名 {  列挙子,  列挙子,  :  }; |

列挙定数は int型で表現できる範囲の整数値です。

列挙定数には、自動的に値が割り振られます。先頭に記述した列挙定数には 0 が、後続の列挙定数には 1、2、3、… といった具合に順番に割り当てられます。

列挙定数 = 定数式のような表記で、明示的に任意の値を割り当てることも可能です。

列挙型を定義したら、この型の変数を宣言することもできます。

例)

|  |
| --- |
| #include <stdio.h>  enum ACTION\_TYPE {  ACTION\_TYPE\_ATTACK = 0,  ACTION\_TYPE\_MAGIC,  ACTION\_TYPE\_ESCAPE,  ACTION\_TYPE\_MAX,  };  void main(void)  {  enum ACTION\_TYPE num = ACTION\_TYPE\_MAX;  printf("行動を選択\n");  printf("0:攻撃 1:魔法 2:逃げる\n");  printf(":");  scanf("%d", &num);  switch (num) {  case ACTION\_TYPE\_ATTACK:  printf("攻撃をしたが返り討ちにあった・・・\n");  break;  case ACTION\_TYPE\_MAGIC:  printf("魔法は効かなかった・・・");  break;  case ACTION\_TYPE\_ESCAPE:  printf("逃げれられない！");  break;  default:  printf("不正な入力！");  break;  }  } |

## typedef

キーワードtypedef を用いて、既存のデータ型に新しい名前をつけることができます。

|  |
| --- |
| typedef 既存のデータ型 新しい名前; |

### 例)既存の型に名前を付ける

|  |
| --- |
| #include <stdio.h>  typedef unsigned int Uint;  void main(void)  {  Uint i = 0;  } |

### 例)構造体に名前を付ける

|  |
| --- |
| #include <stdio.h>  typedef struct student {  int id;  char name[256];  int age;  } student\_t;  void main(void)  {  student\_t data = { 0 };  } |

### 例)列挙型に名前を付ける

|  |
| --- |
| #include <stdio.h>  typedef enum {  ACTION\_TYPE\_ATTACK = 0,  ACTION\_TYPE\_MAGIC,  ACTION\_TYPE\_ESCAPE,  ACTION\_TYPE\_MAX,  } ACTION\_TYPE;  void main(void)  {  ACTION\_TYPE num = ACTION\_TYPE\_MAX;  } |

### 例)関数プロトタイプに名前を付ける

|  |
| --- |
| #include <stdio.h>  typedef struct student {  int id;  char name[256];  int age;  } student\_t;  typedef int (func\_t)(student\_t \*);  int test\_func1(student\_t \*data)  {  data->id = 10;  return 0;  }  int test\_func2(student\_t \*data)  {  data->id = 20;  return 2;  }  void main(void)  {  student\_t student = { 0 };  func\_t \*func\_p = test\_func1;  int ret;  ret = func\_p(&student);  func\_p = test\_func2;  ret = func\_p(&student);  } |

## 動的メモリ（ヒープメモリ）の確保・解放

動的メモリ（ヒープメモリ）は、プログラム実行中にメモリの確保／解放を自由にプログラマーがコントロールできるメモリです。

ヒープメモリの確保は標準ライブラリ関数を呼び出すことで行います。

mallocの引数には確保したいメモリのバイトサイズを指定します。

確保に成功すれば、戻り値に確保したメモリを指すポインタが返却されます。

失敗時はNULLが返却されます。

|  |
| --- |
| #include <stdlib.h>  void \* malloc(size\_t size); |

確保されたヒープメモリは、使い終わったタイミングで解放する必要があります。

freeの引数に解放するメモリのポインタを指定します。

|  |
| --- |
| #include <stdlib.h>  void free(void \* mem); |

# コーディングルール

本研修でのコーディングルールを規定します。

職場毎にコーディングルールは変わるため、研修後は配置された職場のコーディングルールに従いましょう。

## インデント

半角スペースを使用する方法や、4文字ではなく8文字など、しばしば論争になりますが、本研修ではタブで4文字分とします。

プログラムが右に行き過ぎて読みにくくなってしまう場合、プログラムそのものが良くない可能性があります。ネストが深くならないよう、修正を検討してください。

switch 文は、「switch」とそれに含まれる「case」ラベルを同じ列にそろえます。

|  |
| --- |
| switch (suffix) {  case 'G':  case 'g':  mem <<= 30;  break;  case 'M':  case 'm':  mem <<= 20;  break;  case 'K':  case 'k':  mem <<= 10;  /\* fall through \*/  default:  break;  } |

## 括弧の位置と空白

括弧の位置は、ブロックの開始行の行末に開始括弧を置いて、ブロックの最終行の始まりに終了括弧を置き、次のようになります。

これは、関数以外のブロック全て（if、switch、for、while、do）にあてはまります。

|  |
| --- |
| do {  body of do-loop  } while (condition);  if (x == y) {  ..  } else if (x > y) {  ...  } else {  ....  } |

関数定義の括弧だけは例外で、開始括弧は次の行の始まりに置きます。

|  |
| --- |
| int function(int x)  {  body of function  } |

以下のキーワードの後には、空白を使ってください。

if, switch, case, for, do, while

しかし、sizeofの後には空白を使わないでください。

括弧内の式の前後に、空白を使わないでください。

|  |
| --- |
| s = sizeof( struct file );  s = sizeof(struct file); |

ポインタデータ、またはポインタ型を返す関数を定義するとき、

「\*」はデータ名または関数名にはつけて、型名につけません。

|  |
| --- |
| char \*linux\_banner;  unsigned long long memparse(char \*ptr, char \*\*retptr);  char \*match\_strdup(substring\_t \*s); |

以下のように、たいていの二項演算子や三項演算子の演算子の前後に、空白を１文字使ってください。

= + - < > \* / % | & ^ <= >= == != ? :

|  |
| --- |
| c=a+b;  if (c<=MAX) {  c=0;  }  c = a + b;  if (c <= MAX) {  c = 0;  } |

しかし単項演算子の後には、空白を使わないでください。

& \* + - ~ ! sizeof defined

|  |
| --- |
| a = sizeof (long);  a = - a;  a = sizeof(long);  a = -a; |

後置インクリメント単項演算子や後置デクリメント単項演算子の前には、空白を使わないでください。

++ --

前置インクリメント単項演算子や前置デクリメント単項演算子の後に空白を使わないでください。

++ --

|  |
| --- |
| c = 0;  c ++;  ++ c;  c = 0;  c++;  ++c; |

構造体演算子「.」と「->」の前後に、空白を使わないでください。

行末に無駄な空白を残さないでください。

## 名前の選択

グローバル変数（本当に必要な時にだけ使うこと）には意味がよく分かる名前を付けてください。グローバル関数も同じです。

ローカル変数の名前は短くて、変数の特徴を表したものにします。

## 関数

関数は短くて親切、かつ単純に一つのことを行いましょう。

一つの関数は一つの処理に専念して、それだけをキッチリと行いましょう。

ソースファイル内では、空行１行で関数を区切ってください。

関数のプロトタイプ宣言には、データの型と一緒に、パラメータ名を含めてください。

## 関数終了コードの共通化

goto 構文は、関数がいくつかの場所で処理を終了してから共通的な動作（例えばクリーンアップ動作）を行う場合に使用します。それ以外で安易に使用しないでください。

|  |
| --- |
| int fun(int a)  {  int result = 0;  char \*buffer = malloc(SIZE);  if (buffer == NULL)  return -ENOMEM;  if (condition1) {  while (loop1) {  ...  }  result = 1;  goto out;  }  ...  out:  kfree(buffer);  return result;  } |

## コメント

コメントスタイルは、C89 の "/\* ... \*/" スタイルです。

C99 スタイルの "// ..." のコメントは使用してはいけません。

長い（複数行にわたる）コメントにとって推奨されるスタイルを次に示します。

|  |
| --- |
| /\*  \* この形式が、複数行のコメントにおいて  \* 推奨されるスタイルです。  \* 一貫して、それを使用してください。  \*  \* 説明 - 左側にアスタリスクの列を置き、開始行と終了行には、  \* なにもない行を置いてください。  \*/ |

## マクロ、列挙型

定数を定義するマクロの名前と列挙定数は大文字で記述します。

|  |
| --- |
| #define CONSTANT 0x12345 |

マクロは大文字が好ましいですが、関数形式マクロは小文字でも構いません。

また、複数の文から構成されるマクロは do - while ブロックで囲むべきです。

|  |
| --- |
| #define macrofun(a, b, c) \  do { \  if (a == 5) \  do\_this(b, c); \  } while (0) |

マクロ使用時に避けるべきこと

1. 制御の流れに影響を与えるマクロ

|  |
| --- |
| #define FOO(x) \  do { \  if (blah(x) < 0) \  return -EBUGGERED; \  } while(0) |

関数呼び出しのように見えながら、「呼んだ」関数から抜けてしまいます。

1. 特定の名前のローカル変数に依存しているマクロ

|  |
| --- |
| #define FOO(val) bar(index, val) |

コードを読む際に混乱させるし、表面上、問題の無い変更が処理を壊す傾向にあります。

1. 左辺値として使われる引数付マクロ

|  |
| --- |
| FOO(x) = y |

1. 演算子の優先度が不定

式で定数を定義するマクロは式を括弧で囲まなければなりません。引数を利用するマクロの同じような問題に用心しましょう。

|  |
| --- |
| #define COST 100  #define PRICE **(**COST + 10**)**  void func(void)  {  int num = 20;  int total;  total = PRICE \* num;  printf("%d¥n", total);  } |

括弧で囲まないと、予想と違う結果を生むことになるからです。

# コーディング作法

IPA 独立行政法人 情報処理推進機構が発行しているコーディング作法ガイドが参考になりますので参考にしてください。

<https://www.ipa.go.jp/files/000064005.pdf>

## インデントが深くなってしまったら

|  |
| --- |
| int example\_function(void)  {  int result = 0;  int error = 0;    result = function1();  if (result < 0) {  error = -EACCES;  } else {  result = function2();  if (result < 0) {  error = -EBUSY;  } else {  result = function3();  if (result < 0) {  error = -ENOMEM;  }  }  }  return error;  }  int example\_function(void)  {  int result = 0;  int error = 0;    do {  do while文で囲み、条件が成立したらbreakすることでネストを浅くできます。  result = function1();  if (result < 0) {  error = -EACCES;  break;  }  result = function2();  if (result < 0) {  error = -EBUSY;  break;  }  result = function3();  if (result < 0) {  error = -ENOMEM;  break;  }  } while (0);  return error;  } |

## 途中return

処理の途中でreturnする場合は気を付けて使用しましょう。

|  |
| --- |
| int example\_function(void)  関数の途中でreturnすると実行すべき処理が抜けてしまう可能性があります。バグを引き起こしやすいため避けましょう。  {  int result = 0;  char \*buff = NULL;  buff = malloc();  if (buff == NULL) {  return -ENOMEM;  }  result = function2(buff);  if (result < 0) {  return -EBUSY; **/\* ここでreturnするとfreeが呼ばれなくなってしまう \*/**  }  free(buff);  return 0;  }  int example\_function(void)  {  int result = 0;  int error = 0;  char \*buff = NULL;  do {  buff = malloc();  if (buff == NULL) {  error = -ENOMEM;  break;  }  result = function2(buff);  if (result < 0) {  error = -EBUSY;  break;  }  } while (0);  if (buff != NULL) {  free(buff);  }  return 0;  } |

|  |
| --- |
| int example\_function(int type)  {  switch (type) {  switch文の中にreturnを入れるのも避けましょう。  case TYPE\_A:  return 1;  break;  case TYPE\_B:  return 2;  break;  default:  return 3;  break;  }  return 0;  }  int example\_function(void)  {  int result = 0;  int error = 0;  char \*buff = NULL;  do {  もちろんこのようなreturnもダメです。  buff = malloc();  if (buff == NULL) {  return -ENOMEM;  break;  }  result = function2(buff);  if (result < 0) {  return -EBUSY;  break;  }  } while (0);  if (buff != NULL) {  free(buff);  }  return 0;  } |

|  |
| --- |
| int example\_function(int type)  {  if (type >= TYPE\_MAX) {  return -1;  処理前のパラメータチェックでのエラー時に使用すると、後述の処理がスッキリしたりするのでよいです。  }  switch (type) {  case TYPE\_A:  break;  case TYPE\_B:  break;  default:  break;  }  return 0;  } |

## マジックナンバーを避ける

wikipediaより引用

|  |
| --- |
| プログラムにおけるマジックナンバー（英: magic number、魔法の数字）とは、何らかの識別子もしくは定数として用いられる、プログラムのソースコード中に書かれた具体的な数値である。そのプログラムを書いた時点では製作者は数値の意図を把握しているが、他のプログラマーまたは製作者本人がマジックナンバーの意図を忘れたときに閲覧すると「この数字の意味はわからないが、とにかくプログラムは正しく動く。まるで魔法の数字だ」という皮肉を含む。 |

|  |
| --- |
| int example\_function(int type)  {  int result = 0;  switch(type) {  case 0:  break;  case 1:  break;  default:  break;  }  return result;  }  typedef enum {  EXAMPLE\_TYPE\_A = 0,  EXAMPLE\_TYPE\_B,  enumやdefineを使って読み手に分かりやすくします。  また、値の変更が容易になり、保守性も上がります。  EXAMPLE\_TYPE\_MAX,  } EXAMPLE\_TYPE;  int example\_function(EXAMPLE\_TYPE type)  {  int result = 0;  switch(type) {  case EXAMPLE\_TYPE\_A:  break;  case EXAMPLE\_TYPE\_B:  break;  default:  break;  }  return result;  } |

## 型の選択

変数を作成する際、何の型にすればよいだろう？

と迷うことがあるかもしれません。

一例をあげておきます。

|  |  |
| --- | --- |
| int  unsigned int | 最も使われる型かもしれません。例えば、関数の戻り値として使用し、成功なら0/失敗なら負値として使用することが多いです。その他にもさまざまな変数に使用されます。  unsignedを付けると負値が使用できなくなる代わりに扱える整数の範囲が増えます。負値が使用されないことが明らかであればこちらを使用します。 |
| char  unsigned char | 文字列を扱う場合に使用されます。  1byteのデータを扱う際にも使用します。例えば、バイナリデータを1byteずつ解析する際にunsigned charを使用します。 |
| short  unsigned short | 2byteのデータを扱う場合に使用されます。それ以外にはあまり使われません。（メモリが限られる組み込みシステムでは使用されることもあります） |
| long  unsigned long | intの代わりによく使われますが、32bitCPUでは4byte、64bitCPUでは8byteにサイズが変わります。多用すると痛い目にあいます。 |
| long long  unsigned long long | 8byteのサイズを使用したいのであれば、longを使用せずにlong longを使用しましょう。  大きな値を扱う場合に使用します。 |
| float/double | 複雑な算術を使用する場合以外には使いません。今回の研修で使用することはないでしょう。 |

## フォールスルー

switch文などのbreakをあえて入れない場合は、それがわかるように/\* fall through \*/とコメントをいれて、break抜けのバグなのか意図的なのかがわかるようにしましょう。

|  |
| --- |
| switch (suffix) {  case 'G':  case 'g':  mem <<= 30;  break;  case 'M':  case 'm':  mem <<= 20;  break;  case 'K':  case 'k':  mem <<= 10;  /\* fall through \*/  default:  break;  } |

## 多重インクルード防止

ヘッダファイルには「自分はすでにインクルードされています」を意味するシンボルを定義(#define)して、ヘッダ全体を次のようにインクルードガードで囲みましょう。

下記例のHEADER\_Hは各自のヘッダファイル名に置き換えてください。

|  |
| --- |
| #ifndef HEADER\_H  #define HEADER\_H  /\* ... <header.h> の内容がここに入る \*/  #endif /\* HEADER\_H \*/ |

これがないと、ヘッダファイル内の定義が多重定義されてしまう可能性があります。

## 大きなローカル変数を作らない

ローカル変数はスタックメモリを消費します。大きなサイズのローカル変数を作ると、スタックオーバーフローの原因となります。

Windowsアプリではスタックサイズが1MB程度用意されるため、本研修では問題になる事は稀かと思います。

が、組み込みソフトウェア開発の世界ではシビアになってきますので、今のうちに意識しておきましょう。

・ローカル変数のサイズが大きい場合（基準が難しいですが、128byte以上あれば多いと考えてください）はmallocを使用してヒープメモリを確保する

※スタックメモリ…プログラムを実行しているタスクやプロセスが使用するメモリで、実行した関数のアドレスやローカル変数などを格納しています。サイズはシステムによってあらかじめ決められています。

## TBD

適宜追加予定

# 実習

実際にC言語を使用してプログラムを作成しましょう。

開発環境にはVisual Studio等を使用します。

本実習ではscanfを使用します。Visual Studioを使用する場合、コンパイルエラーになりますので、ソリューションエクスプローラー→プロパティ→C/C++→SDLチェック

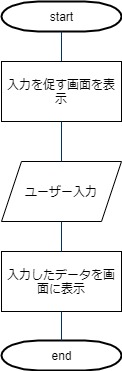
を「いいえ」に設定してください。

## 課題1

以下のフローチャートとプログラムを改造してください。

改造内容：ユーザー入力関数の戻り値も画面に出力してください。

フローチャート



プログラム

|  |
| --- |
| #include <stdio.h>  #define TEST\_BUFF\_SIZE (50 + 1)  int main(void)  {  char buff[TEST\_BUFF\_SIZE] = {"\0"};  /\* 入力を促す画面を表示 \*/  printf("scanfの動作確認です。文字を入力してください。\n");  printf(":");  /\* ユーザー入力 \*/  scanf("%50s", buff);  /\* 結果を表示 \*/  printf("出力バッファ:%s", buff);  return 0;  } |

プログラムが出来たら、動作確認してみましょう。

(1)50byte未満の文字を入力したときの結果は？

(2)50byteの文字を入力したときの結果は？

12345678901234567890123456789012345678901234567890

(3)51byteの文字を入力したときの結果は？

123456789012345678901234567890123456789012345678901

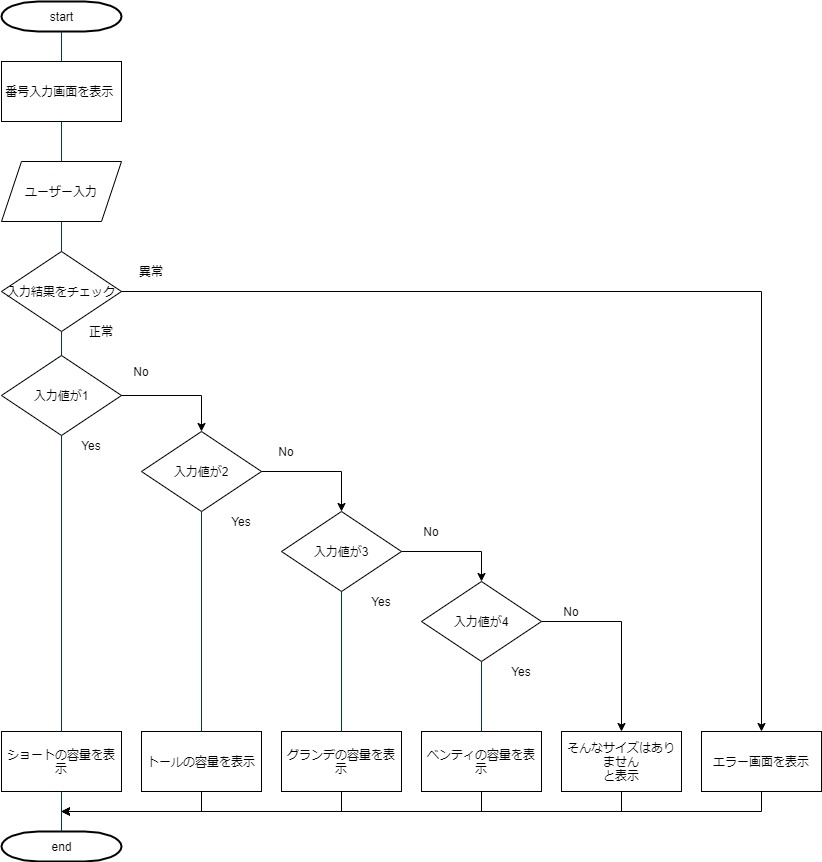
(4)ctrl + zキー、enterキーの順に3回押したときの結果は？

※(4)はVisual Studioを使用する場合のみ確認可能です

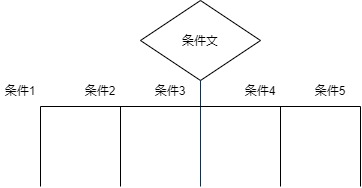
## 課題2

以下のフローチャートとプログラムを改造してください。

改造内容：if文をswitch文に変えてください。



switch文は以下のように表現してください

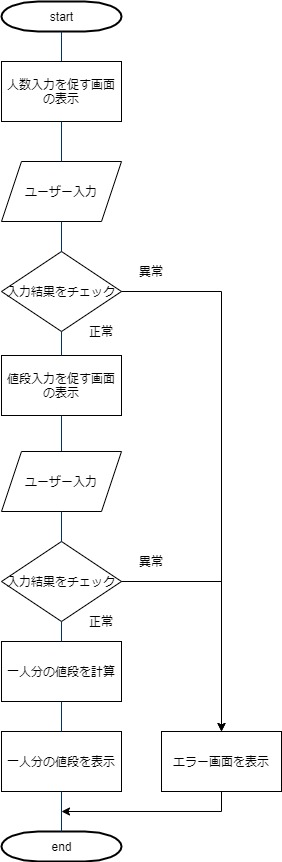


|  |
| --- |
| #include <stdio.h>  enum {  SIZE\_TYPE\_SHORT = 1,  SIZE\_TYPE\_TALL,  SIZE\_TYPE\_GRANDE,  SIZE\_TYPE\_VENTI,  SIZE\_TYPE\_MAX,  };  int main(void)  {  int size\_index = 0;  int scan\_ret = 0;  /\* 番号入力画面を表示 \*/  printf("[1]ショート（Short）\n");  printf("[2]トール（Tall）\n");  printf("[3]グランデ（Grande）\n");  printf("[4]ベンティ（Venti）\n");  printf("容量を知りたいサイズを番号で入力して下さい。\n");  printf(":");  /\* ユーザー入力 \*/  scan\_ret = scanf("%d", &size\_index);  if ((scan\_ret == EOF) || (scan\_ret == 0) || (size\_index < 0)) {  printf("入力値が異常です\n");  } else {  /\* 結果表示 \*/  if (size\_index == SIZE\_TYPE\_SHORT) {  printf("ホット240ml/アイス300mlです\n");  } else if (size\_index == SIZE\_TYPE\_TALL) {  printf("350mlです\n");  } else if (size\_index == SIZE\_TYPE\_GRANDE) {  printf("470mlです\n");  } else if (size\_index == SIZE\_TYPE\_VENTI) {  printf("590mlです\n");  } else {  printf("そんなサイズはありません\n");  }  }  return 0;  } |

## 課題3

以下のフローチャートとプログラムを改造してください。

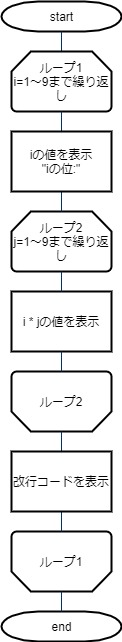
改造内容：余りを切り上げにして、おつりの値段も表示するようにしてください。



|  |
| --- |
| #include <stdio.h>  int main(void)  {  long price = 0;  long count = 0;  long div\_ret = 0;  int scan\_ret = 0;  int result = -1;  do {  /\* 人数入力を促す画面の表示 \*/  printf("割り勘の計算をします。人数を入れてください\n");  printf(":");  /\* ユーザー入力 \*/  scan\_ret = scanf("%ld", &count);  if ((scan\_ret == EOF) || (scan\_ret == 0) || (count <= 0)) {  break;  }  /\* 値段入力を促す画面の表示 \*/  printf("値段を入れてください\n");  printf(":");  /\* ユーザー入力 \*/  scan\_ret = scanf("%ld", &price);  if ((scan\_ret == EOF) || (scan\_ret == 0) || (price <= 0)) {  break;  }  /\* 一人分の値段を計算 \*/  div\_ret = price / count;  result = 0;  } while (0);  /\* 結果の表示 \*/  if (result == 0) {  printf("一人分は%ld円です\n", div\_ret);  } else {  printf("エラーが発生しました\n");  }  return result;  } |

## 課題4

フローチャートの通りにプログラムを作成してください。



## 課題5

課題3のプログラムに出てきた「1人分の値段を計算」する処理を関数化しましょう。

関数のインターフェースは、

引数に人数と値段、戻り値に一人分の値段を返却する関数にしてください。

## 課題6

ポインタの練習。まずは飛ばしてください。課題７，８に取り組む際、ポインタが理解できていない場合は相談しましょう。

## 課題7

課題3のプログラムに出てきた「ユーザー入力」の処理を関数化しましょう。

## 課題8

以下のメンバを持つ構造体を作成し、ユーザー入力によって名前、テストの点数を10人分入力させて、名前・点数・評価を一覧表示するプログラムを作成しましょう。

|  |  |
| --- | --- |
| 名前 | 最大２０文字（全角20文字/半角40文字） |
| テストの点数 | 0～100の数字 |
| 評価 | A：テストの点数が90～100点  B：テストの点数が60～89点  C：テストの点数が30～59点  D：テストの点数が29点以下 |

## 課題9

下記の仕様で、じゃんけんをするプログラムとフローチャートを作成しましょう。

・プレイヤーとコンピュータで対戦を行う。

・プレイヤーは、scanf関数を使って、じゃんけんの手を入力する。

・コンピュータは、rand関数を使って、じゃんけんの手を決定する。

・結果が「勝ち」「負け」となった場合は、プログラムを終了する。

・結果が「あいこ」となった場合は、プログラムを継続する。