C 言語独習用

資料

目次

1 本書の目的

本書は C 言語を利用して独習でプログラミングを行える様になる事を目的としています。

対象者 IT スキルは下記を行える事を目安にしています。

- PC の基本操作
- ソフトウェアのインストール、アンインストール
- インターネットサービスのアカウント新規登録等

2 開発環境構

2.1 Visual Studio 2019 Community インストール手順

1. Visual Studio Community 2019 - Free IDE and Developer Tools (microsoft.com)を下記URL からダウンロードする。
   https://visualstudio.microsoft.com/ja/vs/community/

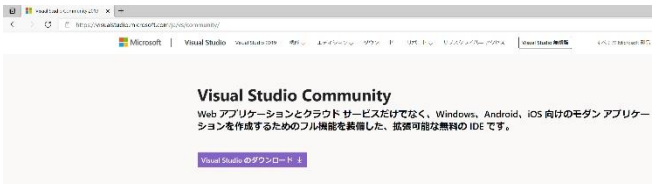

図 1 ダウンロードリンク

2. ダウンロードしたインストーラーを起動する
   任意のフォルダにダウンロードした Visual Studio 2019 Community のインストーラーを起動する。

図 2　ダウンロードしたインストーラーファイル
（※インストーラーの数字の部分は同一の必要はありません）

3. インストールウィザードからインストールを行う
   「このアプリがデバイスに変更を加えることを許可しますか」
   上記のメッセージウィンドウが表示されたら、「はい」をクリックする。

   「作業を開始する前に、インストールを構成するためにいくつかの点を設定する必要があります。」
   上記のメッセージウィンドウが表示されたら、「続行」をクリックする。

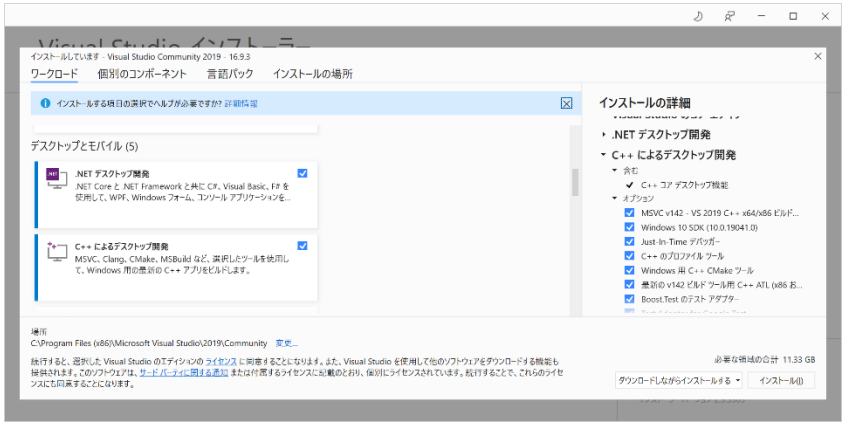

図 3 インストール内容の選択

上図、図 3 と同じく「NET デスクトップ開発」、「C++によるデスクトップ開発」にチェックを入れる。「インストール(I)」をクリックする。

インストール処理が始まり、進捗が表示されるので、インストール終了まで待つ。

インストールウィザードが閉じ、下図の画面が表示され、インストール終了

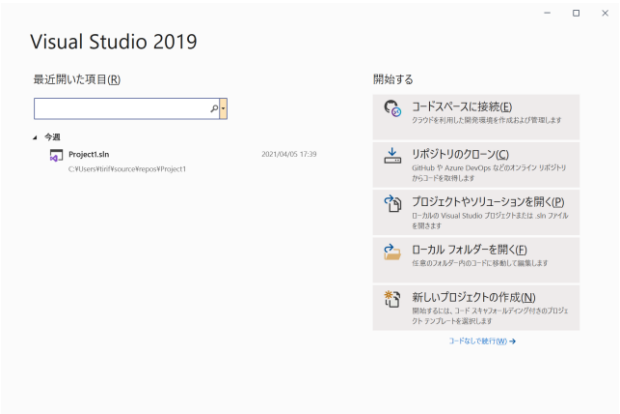

図 4　インストール修了後に自動的に起動した Visual Studio の画面

3 C 言語解説とサンプルプログラム

3.1 変数

変数とはプログラムの中で数値や文字などのデータを格納する入れ物になります。
プログラミングでは変数への数値の格納を代入と言います。
C 言語の基本的な変数を下記に示します。
＊から始まる文章は備考の為、読み飛ばしても問題ありません。

- char 型
  文字型と言われる方で 8bit=1byte のサイズの変数になります。
  扱える数値範囲は 8 ビット符号付(通常の char 型、または signed char 型)では
  下記になります。
  -128～127
  符号なし(unsigned char 型)では下記の範囲を扱えます。
  0～255
  ＊符号付変数とは最上位ビット(MSB)を変数が負の数直、正の数値どちらを
  格納しているか判断するフラグとして利用している変数になります。
  1 ビット分をフラグとして利用するためビットで数値を格納する範囲が
  7bit となり表現可能範囲の 2 の 7 乗になるため
  符号なしの 8bit、2 の 8 乗の表現範囲の最大値の半分となり、符号なしでは
  表現できない負の数値と正の数値を扱えるようになります。
  その他の方も後述する浮動小数点系の型と真偽のみ扱える bool 型以外は
  共通になります。
  ＊ビット数の増減による表現可能範囲の変化は進数関係の為
  下記リンク参照願います。
  進数 - Wikipedia 二進法 - Wikipedia 八進法 - Wikipedia
  十進法 - Wikipedia 十六進法 - Wikipedia

char 型は文字コードと呼ばれる特定の数値を格納することにより
数値を文字として扱うことが可能です。
文字コード例　A の文字コード 0x41(16 進数表記)(ASCII)
A の文字コード 65(10 進数表記) (ASCII)
A の文字コード 0101(8 進数表記) (ASCII)
＊先頭に 0 をつけると 8 進数として扱われます
A の文字コード 0101(8 進数表記) (ASCII)

これらの数値を直接 char 型に代入又は「’（シングルクォーテーション)」
という記号で文字を囲み代入すると後述のサンプルプログラムで使用する
printf 関数等で文字として画面上に表示可能になります。

例　シングルクォーテーションを利用した char 型への代入例
char chr = ‘A’;
＊文字コードは様々なものがありますが C 言語では基本的に
ASCII という文字コードが利用されます。
その他の文字コードについては下記のリンク参照願います。
文字コード - Wikipedia

- int 型
整数型の基本で一般的な PC の開発環境では 32bit=4Byte のサイズの
データ型として扱われます。

一般的な PC 等での開発環境で扱える数値範囲は
32 ビット符号付(通常の int 型、または signed int 型)では
下記になります。
-2147483648～2147483647
符号なし(unsigned int 型)では下記の範囲を扱えます。
0～4294967295

＊int 型(特に unsigned int 型)は別名 word 型と呼ばれることもあり
マイコンなどの場合、処理系(マイコンの一度に扱える bit 数)に依存するため
必ずしも変数サイズが 32bit=4Byte とはなりません。下記に例を示します。
8bit マイコン 8bit=1Byte 16bit マイコン 16bit=2Byte
32bit マイコン 32bit=4Byte

- long int 型
  基本的に int 型と変わらない 32bit=4Byte の整数型になります。
  開発環境によって int 型が 16bit=2Byte に成ってしまう事がある為
  意図的に 4Byte の変数として宣言する為などに用いられます。

  扱える数値の範囲は signed,unsigned ともに 4Byte の int 型と同じになります。

- short int 型
  4Byte の int 型の半分、16bit=2Byte の整数型になります。
  long int 型とは逆に意図的に 2Byte の整数型として宣言する際などに
  用いられます。主にメモリの節約などの理由で使用されることが多いです。

- float 型
  単精度浮動小数点と呼ばれる少数を扱える型になります。
  変数のサイズは 4Byte になります。
  int 型や char 型とは内部的に違った方式で
  数値管理を行っている型になります。
  int 型 char 型とは異なり最上位ビット(MSB)の符号ビットなし
  unsigned は使用不可の変数型になります。
  扱える数値範囲は下記になります。
  3.4E-38 ～ 3.4E+38
  ＊参考　浮動小数点数 - Wikipedia
  物理演算等の少数が必要な場面で用いられます。

- double 型
  倍精度浮動小数点と呼ばれる少数を扱える型になります。
  変数のサイズは 8byte になります。
  基本的には単精度浮動小数点型の float と変わりませんが
  扱える数値の範囲の大きさと扱える少数の細かさ(精度)が
  float 型より高くなります。
  扱える数値範囲は下記になります。
  1.7E-308 ～ 1.7E+308

- long double 型
  拡張精度浮動小数点と呼ばれる少数を扱える型になります。
  Windows 上での開発では double 型と同一になりますが、
  処理系によっては double 型より精度が高くなる場合があります。

3.2 配列

- 1 次元配列
  1 次元配列とは単に配列とも呼ばれ下記のように宣言し、要素数とインデックスと呼ばれるもので管理する連続した変数のまとまりの事を言います。
  要素数の番号は 0 から始まり宣言時の要素数-1 で終わります。

  例 要素数 10 の int 型変数の配列　i の宣言
  インデックスの範囲は 0~9 配列 i のインデックス 3 に 10 を代入する

```
int　i[10];
i[2]=10;
```

- 多次元配列
  多次元配列とは 2 次元以上の配列の事を言います。

  例　2 次元目の要素数 2、1 次元目の要素数 10 の char 型 2 次元配列 chr の宣言
  2 次元目のインデックス 2、1 次元目のインデックス 5 に文字 A を代入

```
char　chr[2][10];
chr[1][4]='A';
```

  例　全次元要素数 5 の float 型 3 次元配列 f の宣言
  3 次元目インデックス 3、2 次元目インデックス 1、1 次元目インデックス 5
  上記に 0.01 を代入

```
flaot　f[5][5][5];
f[2][0][4]=0.01;
```

  上記の例のように配列の変数名[要素数]の[]1 つにつき 1 次元と数えていきます。
  3 次元以上の多次元配列も宣言可能です。

3.3 列挙型

列挙型とある一定の変数名を定数(固定値)として連続して宣言する型になります。

例　rgbColor という列挙型の rgb と言う変数の宣言
enum rgbColor {red,green,blue} rgb;

上記のように宣言すると rgbColor という型の rgb という変数が宣言され
それぞれ red=0,green=1,blue=2 として使用可能になります。

例　宣言した列挙型の使用例
rgb=green; この例は列挙型 rgbColor の rgb という変数に
1 を代入したことになります。

printf(“%d”,red); この例はコマンドプロンプトなどのコンソール画面に
0 を表示したことになります。
＊printf についての詳細は後述します。

int　i=blue;この例は int 型変数 i に 2 を代入したことになります。

列挙型はこのように 0 から番号を定数に割り振っていく為
rgb のように色の管理などを数値で直接行わず定数によって可読性を
良くする目的などで使用されます。

3.4 構造体

構造体とは複数の変数型を一つのまとまりとして、宣言する方法になります。
下記のように宣言します。

例　前項の列挙体を含む、構造体の宣言例

```
enum gender {male,female};//性別用列挙体 gender の宣言

struct human //人物の管理用構造体 human の宣言
{
        char name[60];   //名前用 char 型配列  name
        int age;         //年齢用 int 型変数
        gender gend;     //性別用列挙型 gender の変数 gend
        folat height;    //身長用変数 height
        folat weight;     //
};
```

例　宣言した列挙体と構造体に値を実際に代入。

```
struct human human1; //human 型構造体の変数  human1 の宣言

strcpy_s(human1.name,"テストネーム");
//上記、構造体の名前用配列 name に名前を入力　(注)要  include <string.h>

human1.age = 20; //構造体の年齢用変数 age に 20 を代入
human1.gend = male; //構造体の性別用、列挙体変数 gend に male を代入
human1.height = 170.5; //構造体の身長用変数 height に 170.5 を代入
human1.weight = 80.2; //構造体の体重用変数 weight に 80.2 を代入
```

上記のように human1 という 1 つの変数に複数の型の変数を
持たせることによりデータ管理を行い易くするため等に用いられます。

3.5 共用体

共用体とは構造体と同じように複数の型の変数をまとめて宣言したものですが
構造体と違い内部で宣言された変数のメモリ領域を共有する型になります。

例　構造体と共用体の型宣言時、メモリ確保の違い

構造体の場合

```
struct test
    {
        int i;
        char c[4];
    };
```

上記の宣言の場合、構造体 test 型のサイズは int 型 1 つの 4 バイト
char 型の配列 4 バイトの計 8 バイトになります。

*宣言の仕方などで 6 バイトサイズ等の構造体を宣言した場合、処理系にも
よりますが、ワードにサイズが合わされて(パディング、アライメント)
8 バイトになることなどがあります。
パディング、アライメントのされかたは処理系によって異なるため
8 バイト以上になることもあります。

共用体の場合

```
union test2
{
    int i;
    char c[2];
};
```

上記の宣言の場合、共用体 test2 型のサイズは
処理系によりますが、基本的に内部の型で最も大きい方のサイズ
(この場合 int 型の 4 バイト)になります。
＊次項へ続きます。

また、メモリの扱い方も構造体と違い変数ごとにメモリ上に保存されず
一つのメモリ領域を共有して利用されます。
＊下図参照

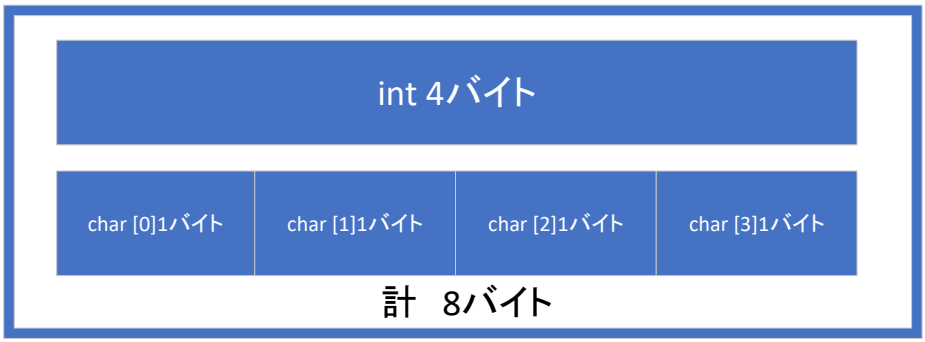

図 5　構造体　メモリ確保イメージ

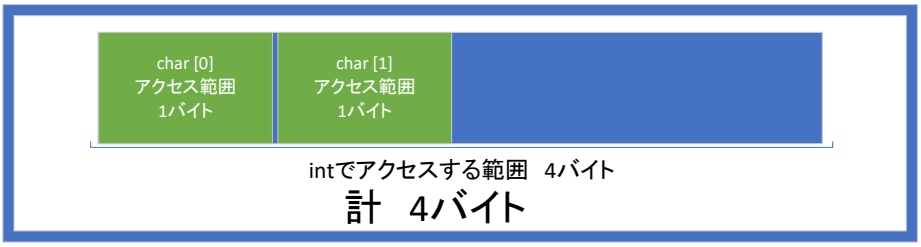

図 6 共用体　メモリ確保イメージ

3.6 ポインタ

- 通常のポインタの利用

  ポインタとはメモリ空間のアドレスを扱うための変数になります。
  通常の値型の変数や、関数等は宣言されるとメモリ上に配置され
  メモリのアドレスが割り振られます。

  メモリ上に割り振られた変数や関数のアドレスを格納し間接参照演算子(*)を用いてアドレスの中に格納されている値の変更などを行うのに用いらます。

  ＊メモリ上のアドレスを扱う変数なので OS 等が管理しているアドレスに
  不意にアクセスしてしまい OS の動作などに致命的な損害を
  与えてしまう事などがあるので、使用する際には注意が必要になります。

  例　int 型のポインタ変数の宣言と int 型の変数のアドレスの代入例

```
int*   ip1;    //int 型ポインタ宣言例 1
int   *ip2;    //int 型ポインタ宣言例 2
```

  ＊その他の型のポインタも上記のように
  　　型名* 又は 型名　*変数名とするとポインタ変数として宣言されます。

```
int   i = 0;  //通常の int 型の宣言
ip1 = &i ;    //変数の頭に(&)アドレス参照演算子をつけてポインタに代入すると
              //変数 i のメモリ上でのアドレスが ip1 に代入されます
ip2 = ip1;  //ポインタはアドレスを格納している変数なので ip2 に ip1 を
              //代入すると ip1 に格納されているアドレスが
              //ip2 に代入されます。
```

  　　＊上記の代入の際に&ip1 としてしまうと ip1 に格納されている
  　　アドレスではなく ip1 自身のアドレスが代入されてしまうので
  　　注意が必要です。

```
*ip1=10；  //左記のようにポインタ変数の頭に間接参照演算子(*)をつけて
           //代入すると ip1 に格納されているアドレスの値を
           //格納する領域に値を代入できます。
           //(この場合 int 型 i に 10 を代入したことになります。)

 *ip2=5;    //また ip1 に格納しているアドレスを代入された ip2 も
           //(*)を利用して ip2 に代入すると同じアドレスを格納している為
           //int 型 i に 5 を代入したことになります。
```

例　構造体、共用体のポインタの宣言と内部変数へのアクセス方法

```
strcut   test //テストという構造体の宣言
{
     int i;      //int 型 i の宣言
     char c;   //char 型 c の宣言
     int *ip;   //int 型ポインタ ip の宣言
};

int j = 0;                  //int 型変数 j の宣言
struct   test   test1;    //構造体 test 型変数  test1 の宣言
struct   test   *ptest;   //構造体 test 型変数ポインタ  ptest の宣言

ptest = &test1;          //ptset に test1 のアドレスを代入

ptest->i = 20;          //構造体のポインタから内部変数にアクセスする場合
                            //基本的に｛->(アロー型演算子)｝と呼ばれる演算子を
                            //利用しアクセスします。

(*ptest).ip = &j;       //また、左記の様に間接参照演算子で構造体のポインタを
                        //括弧で括り、間接参照演算子(*)でアクセスした場合
                        //間接参照演算子(*)の効果で通常の構造体のように(.)で
                        //内部の変数にアクセスできます。
                        ＊(*ptest).ip ではなく*ptest.ip としてしまうと演算子の
                          優先順位の関係で(.)演算子が優先され
                          *(ptest.ip)と解釈されてしまい
                          内部変数の ip に間接参照演算子が、
                          適用されてしまいますので、注意が必要になります。
```

- ポインタの応用

ポインタはアドレスを扱う変数の為、格納しているものがアドレスということもあり
プログラム実行時にメモリ上に展開されてアドレスが割り振られているもは基本的に
扱うことが可能です。

例　ポインタの関数への利用

#include <stdio.h> //#include コマンドで stdio.h という標準入出力ライブラリファイルの読み込み

#include <stdlib.h> //同コマンドで stdlib.h という標準ライブラリファイルの読み込み

＊ファイル名.h なっているものはヘッダファイルと言います。

詳細は後記のリンクを参照願います。　ヘッダファイル - Wikipedia

void　testFunc(); //void 型戻り値(戻り値なし)の関数 testFunc のプロトタイプ宣言

void　tsetFunc2();//同様に testFunc2 のプロトタイプ宣言

＊プロトタイプ宣言の詳細は後記参照願います。　関数プロトタイプ - Wikipedia

```
//メイン関数の宣言　プログラムを実行するとこの関数から呼び出され
//{}で囲まれた範囲のプログラムを実行して行きます。
  int main(void)
    {
        void　(*pFunc)();　// void 型戻り値関数のポインタ変数 pFunc の宣言;

        testFunc();          //testFunc 関数、通常実行方法
        testFunc2();         //testFunc2 関数、通常実行方法

        pFunc = testFunc;　//pFunc に void 型関数 testFunc のアドレスを代入
        (*pFunc)();          //pFunc に格納されているアドレスの関数実行
        //testFunc のアドレスが代入されている為、testFunc が実行される。

        pFunc = testFunc2; // pFunc に void 型関数 testFunc2 のアドレスを代入
        (*pFunc)();          //pFunc に格納されているアドレスの関数実行
        //testFunc のアドレスが代入されている為、testFunc2 が実行される。

        return 0; //int 型メイン関数の戻り値、正常終了時は 0 を返す。
                  //その他の場合、エラーコードを返す。
    }
```

次項へ続きます

```
void    testFunc(); //前述の testFunc プロトタイプ宣言に対応する関数の実行部分
{
        printf(“Run testFunc”);
}

void    testFunc2(); //前述の testFunc2 プロトタイプ宣言に対応する関数の実行部分
{
        printf(“Run testFunc2”);
}
```

上記のようにポインタはアドレス上に割り振られたものを格納し実行する事なども、可能になります。

3.7 制御文

- if 文

  if(条件式)で条件式が true の場合{}で囲った範囲、または if 文直下の 1 行を実行する制御文になります、
  条件式は bool 型の値である必要があり bool 型の変数を()中に入れる場合や
  関係演算子(<,<=,>,>=)や等価演算子{!=(等しくない)、==(等しい)}などで bool 型の値にする必要があります。また論理演算子{&&(AND)、||(OR)}で二つ以上の条件判定式を組み合わせることも可能です。

  例　int i が 0 より大きい、かつ 10 以上の場合

```
int i=11;
if(i>0　&&　i>=10)　//i が 10 なので true と判定される
{
        実行処理
}
```

  又は

```
if(i>0　&&　i>=10)
        実行処理 1 行;
```

- else if 文

  直前の if 文の条件判定が false の際に次の条件として判定される if 文になります。

  例　int　i が 10 以上の判定で false の場合

```
int i=9;
if(i>=10)　//i が 9 なので false と判定され次の else if 文が判定される。
{
    実行処理
}

else if(i<10)// i が 10 より小さいので true 判定になる。
{
    実行処理
}
```

  ＊if 文同様処理が 1 行の場合 {} は不要です。

- else 文

  if 文、else if 文など他の条件判定すべてが false の際に実行される判定式になります。

  例　int i が 0 より大きいかつ 10 以上の判定で false の場合

```
int i=0;
if(i>0 && i>=10) //i が 0 のため i>0 の判定が false となり else 文を実行する
{
        実行処理
}

else
{
        実行処理
}
```

＊if 文同様処理が 1 行の場合 {} は不要です。

- switch 文、break 文
  switch(入力値)で下記の例で示す case 文　定数:の記述と一致した個所の処理を実行する文になります。break 文は自身が囲まれている {} の範囲から抜けるという制御文になります。

例 switch(i)の値が 1~5 までのそれぞれの記述方法

```
switch(i)
{
  case 1://i が 1 だった場合
  実行処理
  break; //i が 1 だった場合の処理を実行し switch 文から抜ける。
  case 2://i が 2 だった場合
  実行処理
  break; //i が 2 だった場合の処理を実行し switch 文から抜ける。
   case 3://i が 3 だった場合
  実行処理
  break; // i が 3 だった場合の処理を実行し switch 文から抜ける。
   case 4://i が 4 だった場合
  実行処理
  break; // i が 4 だった場合の処理を実行し switch 文から抜ける。
   case 5://i が 1 だった場合
  実行処理
  break; //1 だった場合の処理を実行し switch 文から抜ける。
  default://すべての case に当てはまらなかった場合に実行されます。(省略可)
  実行処理
}
```

＊case 文はまとめて下記のように書く事もできます。また break 文を記述しないことで次の case 文まで実行することも可能です。

例　i が 1 と 5 で同じ内容の処理を行う場合

```
case 1:
case 5:
実行処理//入力値が 1 と 5 のどちらかの場合に実行される
break;
```

例 case 1:の場合 case 1 と次の case 定数:を実行する場合

```
case 1:
実行処理//入力値が 1 の場合ここから実行され、次の case 3:の処理も実行される。
case 3:
実行処理//入力値が 3 だった場合ここから実行され次の break 文で、
break;//switch 文を抜ける
```

- while 文、インクリメント、デクリメント
  while 文は while(条件式)の条件式が true の場合{}で囲った範囲、または直下の 1 行を繰り返す
  制御文になります。

  インクリメントは変数++又は++変数と記述すると変数に 1 加算される演算子になります。
  変数名の前と後ろに置く違いは、++変数の様に変数名の前に記述した場合 if 文 while 文などの
  条件判定前や値を読み取る前に加算され、変数名++の様に後に記述した場合は条件判定後や値を読み取った後に加算されます。

  デクリメントは--変数名または変数名--と記述しインクリメントと同じく条件判定等の前後に 1 減算されます。
  インクリメント、デクリメントは while 文や後述の for 文の様な繰り返し制御文の繰り返し回数を
  制御する為などによく併用されます。

  例 int i が 10 より小さい間 i を 1 ずつインクリメントし処理を繰り返す

```
int i=0;
while(i<10)//i が{}の中でインクリメントされ 10 以上になるまで繰り返す
    {
        実行処理
        i++;//インクリメントで 1 加算
    }
```

例 int i が 0 より大きい間 i を 1 ずつデクリメントし処理を繰り返す

```
int i=10;
while(i>0)
{
   実行処理
   i--//デクリメントで 1 減算
 }
```

- do while 文

  do while 文は while と同じく while(条件式)の条件式が true の時に繰り返しを行う制御文です while と違い初回は必ず実行されます。
  例を下に示します。

  例　do while と while の違い

```
int i=11;
do
{
    実行処理//i が 11 で 10 より大きいのですが判定が後の為 1 度だけ実行されます。
}while(i=<10)

while(i=<10)
{
    実行処理//i が 11 で 10 より大きく、判定が先の為、実行されません。
}
```

- for 文、continue 文、代入演算子の略記方法
  for 文は while 文と同じく繰り返し制御文になりますが、繰り返しの回数や条件、
  判定に使う変数などを下記の様に
  for(判定用変数;条件式;判定用変数増減方法)という方式で記述する制御文になります。

  continue 文は記述された箇所より後を実行せず、
  繰り返し制御の次の回に移るための制御文になります。

  また、変数への値の代入演算子の略記が for 文の増減処理とよく併用されます。
  主に用いられる四則演算の略記を下に示します。
  変数+=値 → 変数 = 変数 + 値と等価 (加算)
  変数-=値 → 変数 = 変数 - 値と等価 (減算)
  変数*=値 → 変数 = 変数 * 値と等価 (乗算)
  変数/=値 → 変数 = 変数 / 値と等価 (除算)

  例 int i を判定用変数として 10 を代入し、2 ずつ減算しながら,
  i が 4 の時に処理をスキップし 0 になるまで繰り返す

```
int  i=0;
for(i=10;i>0;i-=2) //i に 10 を代入し i が 0 より小さくなるまで 2 ずつ減算
{
  if(i==4)//i が 4 の時 continue 文より下をスキップ
    continue;

  実行処理
}
```