

# C言語再入門講座

# c言語再入門講座 目次

NEXT STEP

1. c言語とは
2. 変数と定数
3. 式と演算子
4. 配列
5. 文字と文字列
6. 2次元配列
7. 変数とポインタ
8. キャスト変換
9. 構造体
10. 共用体
11. 制御
12. 関数

# 1. C言語とは

- (1) プログラミング言語について
- (2) インタプリタとコンパイラ
- (3) C言語の歴史と位置付け
- (4) なぜ組み込み現場ではC言語が現役なのか
- (5) コンピューター内部での数字の扱いとn進数
- (6) 2進数、10進数、16進数の相互変換
- (7) 16進数での負の数の表し方
- (8) C言語標準ヘッダー
- (9) C言語の規格

# 1. c言語とは

## (1) プログラミング言語について

プログラミング言語はコンピュータに解釈できるように作られた人工言語です。

コンピュータへの指令を行う為のプログラムを書くのに使われる言語です。

プログラミング言語は、人間がコンピュータに命令を指示するために作られており、コンピュータが曖昧さなく解析できるように設計されています。多くの場合構文上の間違いは許されず、人間はプログラミング言語の文法に厳密にしたがった文を入力しなければなりません。

初期にはコンピュータが直接解読して動作できる機械語(マシン語)が用いられていましたが、人の使う自然言語とあまりにもかけ離れていて、プログラムの作成効率が悪いため、人にとってわかりやすくしたアセンブラ言語が用いられるようになりました。

ただアセンブラ言語は基本的にマシン語と1対1に対応させただけなので、より人間に理解しやすい記号や代数表現を用いて書ける高級言語が開発されました。

プログラム言語の設計はその目的に応じてデータの形式、処理方法、文法などに違いが出てきます。

たとえば、計算機のシステムをつくるにはより機械語に近い処理が可能な Cが使われ、科学技術計算には数値の扱いに適したフォートランが使われます。

プログラミングの技法に応じて設計される場合も多く、手続き型言語のほか、論理構造を組み立てるのに適した PROLOGなどの論理型言語や関数を組み合わせて新しい関数をつくる LISPなどの関数型言語といった非手続き型言語があります。

また、C++などのオブジェクト指向プログラミング言語（→オブジェクト指向プログラミング）はプログラムの作成効率を上げるのに役立ちました。機種依存性がない言語として Javaはインターネットに向いていることから世界中に広まりました。

近年、たくさんの新しいプログラミング言語が登場しています。

# 1. c言語とは

## (2) インタプリタとコンパイラ

どのような言語で書かれたプログラムもそのままでは実行できません。

マイコンが実行できるマシン語に直して初めてマイコンで実行できます。

高級言語で書かれたプログラムを一括してマイコンで実行可能なマシン語に変換する処理系をコンパイラ、ソースコードを1行ずつ解釈しながらマシン語に直して実行する処理系がインタプリタです。

主なコンパイラ系プログラム

C/C++、Fortran、Cobol

主なインタプリタ系プログラム

BASIC、Ruby、Python

コンパイラの特徴

インタプリタより実行速度が速い

コンパイルの手間がかかる

コンパイルした機械語のプログラムは他の環境（OSやCPUが異なる場合）では実行できない

※Java は基本的にはコンパイル方式の言語ですが、Java仮想マシンの機械語に翻訳され、仮想マシン上で実行されます。

コンパイル方式とインタープリタ方式の中間的な方式となります。

**スピードが要求される組み込みプログラムにはインタプリタは向いていません**

# 1. c言語とは

## (3) c言語の歴史と位置付け

C言語の歴史は古く、1972年頃UNIXの開発から生まれました。

C言語の 'C' の意味は？ と問われても明確に答えることは難しいと思います。

B言語の次に作成されたから「C言語」となり、それ以外'C'自体には特に意味がありません。

B言語の元とされたものはBCPL、その元となったのはCPLとなり、A言語は存在しません。

また、近年ではC言語に続くD言語の開発も進んでいるとのこと。

また、C++以外にもC#、PHP、Java、JavaScript、Objective-CなどC言語の影響を受けた言語も多数あります。

# 1. c言語とは

## (4) なぜ組み込み現場ではc言語が現役で使われ続けているのか

c言語は40年以上の歴史のある古い言語です、この40年間に新しい言語も多数でできました。  
ではなぜ組み込みの現場では今でもc言語が主流なのでしょう。

### その①

コードが軽いので、資源が少ない環境や、制御などにリアルタイム性が要求される組み込みに適した言語である。

### その②

開発資産や主流のソフトウェアがC言語でできています。

### その③

C言語は、アセンブラレベルと同等の処理を簡潔に記述でき、プログラマが意図したコードだけがコンパイラで生成されるため、マイコンの細かい制御が可能となります。この事はマイコンのレジスタを操作したりと組み込みプログラムには不可欠です。

### その④

UNIXでの実績。

### その⑤

解説書が多数出版されている。

# 1. c言語とは

## (5) コンピューター内部での数字の扱いとn進数

日常生活では基本10進法ですが、10進法以外では  
 時間が 秒未満は10進法、秒、分は60進法、時は12進法、または24進法 です。  
 また最近は余り聞かなくなりましたがダースという単位は12進法です。  
 コンピューターは0と1からなる 2進法で動いています。

なぜコンピューターは2進数で動いているのでしょうか？

電気信号のオンとオフでそのまま0と1を表すことができますが、これを10進数で実現しようとする、  
 例えば 0V→0、1V→1、2V→2 …… 8V→8、9V→9  
 のように電圧を読み込む(計る)回路や、正しい電圧を発生させる回路が必要となりとても複雑になってしまいますし、  
 とても現実的ではありません。

プログラムを書く時は数字を表すのに10進数や16進数を使いますが、コンパイラが10進数や16進数を自動で2進数になおしてくれます。

## (6) 2進数、10進数、16進数の相互変換

**2進数 → 10進数変換** 例：2進数 10011011 を 10進数に変換

128	64	32	16	8	4	2	1	
11	11	11	11	11	11	11	11	
2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	桁の重み
1	0	0	1	1	0	1	1	2進数
↓	↓	↓	↓	↓	↓	↓	↓	
128	0	0	16	8	0	2	1	← 加算
						=	155	10進数



# 1. c言語とは

## (6) 2進数、10進数、16進数の相互変換

**10進数 → 2進数変換** 例：10進数 155 を 2進数に変換

2)155	...	1	余りがある場合は 1	下位
2)77	...	1		
2)38	...	0	余りが無い場合は 0	
2)19	...	1		
2)9	...	1		
2)4	...	0		
2)2	...	0		
2)1	...	1		上位
0				

逆さになっている余りを横書きに直すと

**10011011**

155 (10進数) = 10011011 (2進数)

**2進数 → 16進数変換** 例：2進数 10011011 を 16進数に変換

128	64	32	16	8	4	2	1	
2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	桁の重み
<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>1</b>	2進数
↓	↓	↓	↓	↓	↓	↓	↓	
8	0	0	1	8	0	2	1	← 加算
		=	<b>9</b>	<b>B</b>				16進数

# 1. c言語とは

## (7) 16進数での負の数の表し方

16進数の0xFFFFは符号有りとは符号無しでは10進数に変換したときの値が異なります。

符号なし 0xFFFF → 65535

符号有り 0xFFFF → -1

符号有りの場合最上位ビットが1のときはマイナスの値を、最上位ビットが0のときはプラスの値を表します。

符号ありの場合

最上位ビット	符号の種類
1	マイナスの値を表す
0	プラスの値を表す

signed と unsigned

ビット長	型	10進数	16進数	2進数
16ビット	short int	10x0001	0x0001	00000000 00000001
		-10xFFFF	0xFFFF	11111111 11111111
	unsigned short int	65535	0xFFFF	11111111 11111111
32ビット	long int	10x00000001	0x00000001	00000000 00000000 00000000 00000001
		-10xFFFFFFFF	0xFFFFFFFF	11111111 11111111 11111111 11111111
	unsigned long int	4294967295	0xFFFFFFFF	11111111 11111111 11111111 11111111

最上位ビット

マイナス表現には**2の補数**を使用します

2の補数は、その対象の2進数を**全ビットを反転させ、+1**することで実現します。

2進数では2の補数を求める = マイナスの数ということもいえます。

2の歩数の求め方 -1の場合

0000 0000 0000 0001	1の2進数
1111 1111 1111 1110	0と1を反転する
1111 1111 1111 1111	1をたす
0xFFFF	16進数

# 1. c言語とは

## (8) C言語の標準ヘッダ

標準ヘッダは、C言語の標準規格で定められている標準ライブラリの関数宣言や型の宣言やマクロの定義が行われているヘッダ群のことです。JIS X3010:2003(ISO/IEC 9899 : 1999)、通称C99では、標準ヘッダとして以下の24ファイルを定めています。

include <stdio.h> を良くおまじないと表現している書籍等がありますが、stdio.h は入出力に関連する型、マクロ、関数を宣言が定義されているヘッダです。

ヘッダ	名称	内容
assert.h	診断機能	実行時診断を行うassertマクロを定義する。
complex.h	複素数計算	複素数計算をサポートするマクロ、関数を宣言、定義する。C99より追加。
ctype.h	文字操作	文字分類、文字変換に有用な関数を宣言する。
errno.h	エラー	ライブラリ関数内エラーの報告用マクロを定義する。
fenv.h	浮動小数点環境	浮動小数点環境へのアクセス手段を提供するための型、マクロ、関数を宣言、定義する。C99より追加。
float.h	浮動小数点型の特性	浮動小数点型の大きさや特性を表すマクロを定義する。
inttypes.h	整数型の書式変換	最大幅の整数を操作する関数、及び数値文字列を最大幅の整数に変換する関数を宣言する。C99より追加。
iso646.h	代替つづり	演算子の代替つづりマクロを定義する。C95より追加。
limits.h	整数型の大きさ	整数型の大きさを表すマクロを定義する。
locale.h	文化圏固有操作	文化圏固有のデータ等の操作を行う型、マクロ、関数を宣言、定義する。
math.h	数学	数学的な演算を行う関数、及び関連するマクロを宣言、定義する。
setjmp.h	非局所分岐	関数の枠組みを越えた分岐を制御するための型、マクロ、関数を宣言、定義する。
signal.h	シグナル操作	種々のシグナルを操作するための型、マクロ、関数を宣言、定義する。
stdarg.h	可変個数の実引数	可変個の実引数を実現するための型、マクロを定義する。
stdbool.h	論理型及び論理値	論理型及び論理値に関連するマクロを定義する。C99より追加。
stddef.h	共通の定義	処理系に依存する型、マクロを定義する。
stdint.h	整数型	指定幅を持つ整数型を宣言する。また、それらの宣言に対応するマクロを定義する。C99より追加。
stdio.h	入出力	入出力に関連する型、マクロ、関数を宣言、定義する。
stdlib.h	一般ユーティリティ	一般ユーティリティに関連する型、マクロ、関数を宣言、定義する。
string.h	文字列操作	文字列の操作を行うための型、マクロ、関数を宣言、定義する。
tgmath.h	型総称数学関数	数学関数の型総称マクロを定義する。
time.h	日付及び時間	時間を扱うための型、マクロ、関数を宣言、定義する。
wchar.h	多バイト文字及びワイド文字拡張ユーティリティ	多バイト文字、ワイド文字に関連する型、マクロ、関数を宣言、定義する。C95より追加。
wctype.h	ワイド文字種分類及びワイド文字大文字小文字変換ユーティリティ	ワイド文字種の分類や大文字小文字変換に有用な型、マクロ、関数を宣言、定義する。C95より追加。

# 1. c言語とは

## (9) C言語の規格

### K&R

1978年に出版されたリッチーとカーニハンの共著である「The C Programming Language」に記載されている

内容がC言語の仕様として一般に用いられていました。

C89がせいていされるまではこのK&RのCがC言語の基準となっていました。

### C89(ANSI-C)

1990年に制定

この規規格は、様々な国際規格を制定しているISOと、米国規格協会、ANSIによってまとめられました。

この企画が、「ANSI-C」と呼ばれる規格で、それ以前のものを、「K&R」と呼び、同じC言語でも、明確に区別をしています。

日本でもこの企画がJISの規格として採用されました。なお、多くのCの処理系がいまだに「C89」を基準にしています。

### C99

1999年に制定

行コメント「//」が、正式にC言語のコメントとして使用されるようになりました。

そのほかにも、C99言語には、以下のような特徴があります。

- 。
- ・ 予約語追加
- ・ ヘッダーファイル追加
- ・ プリプロセッサの拡張
- ・ 字句追加
- ・ 配列の拡張
- ・ 整数型の拡張
- ・ 複素数型の導入
- ・ 文法の拡張
- ・ ライブラリの拡張

このほか 変数の型として bool型 が追加されました。

まだ一部のCの処理系でこの規格が完全に実装されていないといったこともあり、

C99の仕様はまだ本格的には使われていないのが現状です。

[※ C99による変更点はこちらのページを参考にしてください。](#)

### C11

2011年に制定

大きな特徴の一つに「脆弱性への対応」があります。

この「脆弱性」の一番の対応はC言語の大きな欠点であった「バッファオーバーフロー」のに対してです。

# 1. c言語とは

また、C11ではgets関数の削除や、printf関数における「%n」書式の廃止、fopen関数への排他モードの追加など、

さまざまな脆弱性対応が導入されています。そのため、「C11」は我々が一般に「C言語」と言っている言語とはずいぶん形が変わっています。

C99でもsnprintf関数の導入など、バッファオーバーフローへの対策は入っていましたが、本格的な対策を行ったのはC11からといえます。

## C17(C18)

2018年に制定

仕様の欠陥修正がメインのマイナーアップデート

### 主なcコンパイラの企画対応

処理系	C規格	備考
GCC 4.5	C99	
GCC 4.9	C11	
Clang	C11、C++17	
VC++	C99	ライブラリをほぼ実装 言語機能など規格自体はサポート無し
Intel C++ Compiler	C11	バージョン18.0でC11にほぼ対応
armcc	C90、C99	ARM コンパイラ
IAR C/C++	C99	IARシステムズが提供する組込みアプリケーション向けC/C++言語IDE（統合開発環境）
CS+	C99	Renesas Electronicsのマイコンチップ用の開発環境
ATmel Studio	C99	AVRマイコン用の開発環境

## 2. 変数と定数

- (1) 変数とは
- (2) 変数の宣言(定義)
- (3) 変数の型とサイズと範囲
- (4) フォーマット指定子
- (5) Int型を使うときの注意点
- (6) 算術演算時の型
- (7) ビッグエンディアンとリトルエンディアン
- (8) 定数
- (9) 別名定義 typedef

## 2. 変数と定数

### (1) 変数とは

変数とはデータを格納しておく領域のことです。変数は通常メモリ上に確保され、値を代入したり参照したりすることができます。

### (2) 変数の宣言

変数は以下のように宣言します、宣言とは変数を使用するための定義です。

**記憶域クラス名 型修飾子 型名 変数名;**

記憶域クラス名	説明
auto	通常は省略
static	静的変数、関数の処理が終了しても変数を破棄しない
const	初期化以外で変更不可、主に定数として使う
volatile	コンパイラの最適化を抑制する型修飾子
register	CPUの中にあるレジスタを使う、通常のメモリよりアクセスが早い
型修飾子	説明
short	短
long	長
long long	長長
signed	符号付き
unsigned	符号なし
型名	説明
void	型なし
char	文字型
int	整数型
float	単精度実浮動小数点型
double	倍精度実浮動小数点型

### 変数名のつけ方の制限

使用できる文字は、アルファベット (A～Z, a～z)、数字 (0～9)、アンダーバー ( \_ ) のみ。  
すべて半角で、全角は不可。

第1文字目は必ずアルファベットまたはアンダーバー。数字で始まる変数名は不可。

大文字と小文字は別の文字として扱われる。例えば、linenumberとlineNumberは別の変数となる。

if、else、intなどの予約語は使用できない。

### C言語予約語一覧 (ANSI規格)

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

## 2. 変数と定数

### (3) 変数の型とサイズと範囲

型	サイズ(byte)	範囲	備考
void	-	-	型なし
unsigned char	1	0~255	文字列は unsigned charの配列
signed char	1	-128~127	
unsigned short int	2	0~65535	
signed short int	2	-32768~32767	
unsigned int	2 or 4	0~65535 0~4294967295	処理系によってサイズが変わるので注意が必要
signed int	2 or 4	-32768~32767 -2147483648~2147483647	処理系によってサイズが変わるので注意が必要
unsigned long int	4 or 8	0~4294967295 0~18446744073709551615	処理系によってサイズが変わるので注意が必要
signed long int	4 or 8	-2147483648~2147483647 -9223372036854775808~9223372036854775807	処理系によってサイズが変わるので注意が必要
unsigned long long int	8	0~18446744073709551615	
signed long long int	8	-9223372036854775808~9223372036854775807	
float	4	最小の正の数 : 1.175494e-38 最大値 : 3.402823e+38	
double	8	最小の正の数 : 2.225074e-308 最大値 : 1.797693e+308	

2進数、8進数、10進数、16進数相互変換ツール  
<https://hoge hoge .tk/tool/number.html>

サンプルコード  
[https://paiza.io/projects/gGmLC\\_Cb5Br\\_oelZ5C0wow](https://paiza.io/projects/gGmLC_Cb5Br_oelZ5C0wow)



## 2. 変数と定数

### (4) フォーマット指定子

出力・入力フォーマット指定子

フォーマット指定子とは、のprintf()、fprintf()、sprintf()、scanf()、fscanf()、sscanf()などの関数で使用する、表示形式を指定するための記述子です。

指定子	対応する型	説明(出力の場合)	説明(入力の場合)	使用例
%c	char	1 文字を出力する	1 文字を入力する	"%c"
%s	char *	文字列を出力する	文字列を入力する	"%8s", "%-10s"
%d	int, short	整数を10進で出力する	整数を10進数として入力する	"%-2d", "%03d"
%u	unsigned int, unsigned short	符号なし整数を10進で出力する	符号なし整数を10進数として入力する	"%2u", "%02u"
%o	int, short, unsigned int, unsigned short	整数を8進で出力する	整数を8進数として入力する	"%06o", "%03o"
%x	int, short, unsigned int, unsigned short	整数を16進で出力する	整数を16進数として入力する	"%04x"
%ld	long int	32bit整数を10進で出力する		"%10ld"
%lu	unsigned long int	符号なし32bit整数を10進で出力する		"%10lu"
%lo	long int, unsigned long int	32bit整数を8進で出力する		"%12lo"
%lx	long int, unsigned long int	32bit整数を16進で出力する		"%08lx"
%lld	long long int	64bit整数を10進で出力する		"%-10lld"
%llu	unsigned long long int	符号なし64bit整数を10進で出力する		"%10llu"
%llo	long int, unsigned long int	64bit整数を8進で出力する		"%12llo"
%llx	long int, unsigned long int	64bit整数を16進で出力する		"%08llx"
%f	float	実数を出力する	実数を入力する	"%5.2f"
%e	float	実数を指数表示で出力する		"%5.3e"
%g	float	実数を最適な形式で出力する		"%g"
%lf	double	倍精度実数を出力する		"%8.3lf"
%p	ポインタ値(アドレス)	ポインタ値を16進数で表示		"%p"

## 2. 変数と定数

### 表示桁数の指定

表示桁数は<全体の幅>.<小数点以下の幅>で指定する。<小数点以下の幅>は、文字列の場合には最大文字数の意味になる。

指定例	出力結果	備考
<code>printf("[%8.3f]", 123.45678);</code>	[ 123.456]	小数点含めて8桁、小数点以下3桁
<code>printf("[%15s]", "I am a boy.");</code>	[ I am a boy.]	文字列を15文字幅で表示
<code>printf("[%6s]", "I am a boy.");</code>	[I am a]	最大文字数8で文字列を表示
<code>printf("[%8.3e]", 1234.5678);</code>	[1.235e+3]	最大表示8桁、小数点以下3桁で指数形式で表示

### リーディングゼロ(ゼロ埋め)の指定

数値フィールドの場合に、ゼロ詰めを指定することができる。桁数の指定のまえにゼロを付加する。

指定例	出力結果
<code>printf("[%08.3f]", 123.45678);</code>	[0123.456]
<code>printf("[%05d]", 1);</code>	[00001]

### 符号の指定の指定

数値の表示は、デフォルトではプラス記号を出さないで、付けたいときは+を指定する。

指定例	出力結果
<code>printf("[%+5d]", 32);</code>	[ +32]
<code>printf("[%+5d]", -32);</code>	[ -32]
<code>printf("[%+8.3f]", 1.414);</code>	[ +1.414]

### 右詰・左詰の指定の指定

デフォルトでは右詰なので、左詰にしたいときは桁数指定の前にマイナスをつけなければならない。

指定例	出力結果
<code>printf("[%15s]", "I am a boy.");</code>	[I am a boy. ]
<code>printf("[%8.3f]", 123.45678);</code>	[123.456 ]
<code>printf("[%5d]", 1);</code>	[1 ]

### サンプルコード

<https://paiza.io/projects/vO38qcYpmFGfpu495tx5og>

## 2. 変数と定数

### (5) int型(long int型)を使うときの注意点

変数の型とサイズと範囲表に書いていますがint型のサイズは使用するプロセッサや処理系によって2バイトと4バイトの場合があります。

例として Arduino UNO は2バイト Arduino DUO は4バイトです。  
両方ともに Arduino IDE で開発できますし、相互の移植もほとんど変更なしでできます。

ここで int 型に関して次のような注意が必要となります。

16ビットでは**負の数字**なのが32ビットでは**正の数**になってしまう。

例えば、16ビットの0xFFFFは10進数で表現すると **-1** ですが、これが 32ビットだと **+65355** となります。

	符号有り	符号なし
	0xFFFF	0xFFFF
16ビット	-1	65355
32ビット	65355	65355

次ページのマイコンチップによる、変数型のサイズの違いの例 を参照

## 2. 変数と定数

※ 同じ型でサイズが違う例

この部分

型名	説明	Arduino Uno		Arduino Due	
		変数が占めるサイズ(sizeof())	取り得る値	変数が占めるサイズ(sizeof())	取り得る値
bool (C++言語) _Bool(C言語)	真偽値(trueとfalse)を格納する。	1	true/false(実際には他の値をとることも可能)	1	true/false(実際には他の値をとることも可能)
char	1バイトの値を格納する。文字を格納する。	1	-128～127	1	-128～127
unsigned char	1バイトの値を格納する。文字を格納する。	1	0～255	1	0～255
short int	整数を格納する。	2	-32768～32767	2	-32768～32767
unsigned short int	非負整数を格納する。	2	0～65535	2	0～65535
int	整数を格納する。	2	-32768～32767	4	-2147483648～2147483647
unsigned int	非負整数を格納する。	2	0～65535	4	0～4294967295
long int	整数を格納する。	4	-2147483648～2147483647	4	-2147483648～2147483647
unsigned long int	非負整数を格納する。	4	0～4294967295	4	0～4294967295
long long int	整数を格納する。	8	-9223372036854775808～ 9223372036854775807	8	-9223372036854775808～ 9223372036854775807
unsigned long long int	非負整数を格納する。	8	0～ 18446744073709551615	8	0～ 18446744073709551615
float	浮動小数点数を格納する。	4	-3.4028235e+38～ 3.4028235e+38	4	-3.4028235e+38～ 3.4028235e+38
double	浮動小数点数を格納する。	4	-3.4028235e+38～ 3.4028235e+38	8	-1.79769313486232e308～ 1.79769313486232e308
long double	浮動小数点数を格納する。	4	-3.4028235e+38～ 3.4028235e+38	8	-1.79769313486232e308～ 1.79769313486232e308
float _Complex	複素数型を格納する。	8	-	8	-
double _Complex	複素数型を格納する。	8	-	16	-
long double _Complex	複素数型を格納する。	8	-	16	-

## 2. 変数と定数

前頁の表から、int型の変数を使っている場合お互いにプログラムを移植した場合、変数のサイズの違いにより意図しない動きとなったりバグの原因となったりします。

これを回避するために以下のように型を別名で定義して、それを使うことにより移植した場合にも不具合が生じないプログラムとなります。

なおこの別名は **stdint.h** に定義されています。

型名(幅指定整数型)	説明	stdint.h にでの定義
int8_t	8ビットの整数を格納する。	typedef signed char int8_t
uint8_t	8ビットの非負整数を格納する。	typedef unsigned char uint8_t
int16_t	16ビットの整数を格納する。	typedef signed int int16_t
uint16_t	16ビットの非負整数を格納する。	typedef unsigned int uint16_t
int32_t	32ビットの整数を格納する。	typedef signed long int int32_t
uint32_t	32ビットの非負整数を格納する。	typedef unsigned long int uint32_t
int64_t	64ビットの整数を格納する。	typedef signed long long int int64_t
uint64_t	64ビットの非負整数を格納する。	typedef unsigned long long int uint64_t

## 2. 変数と定数

### (6) 算術演算時の型(暗黙の型変換)

一般に異なる型の変数間の混合演算では、劣勢な方の型は優勢な方の型に変換されてから演算が行なわれると考えてよい。

(実際は少し違うが、結果としてこのように考えてよい)

型の優勢, 劣勢は次のようになっている。

劣勢 char型(1byte) < short int型(2byte) < long int型(4byte) < float型(4byte) < double型(8byte) 優勢

(同じバイト数の型の場合 劣勢 signed < unsigned 優勢 となっている)

混合演算	C 言語での解釈	例
int型とdouble型の混合演算	int型をdouble型に変換してから演算	<pre>int a=3,c; double p=1.5,q; の時 q=a+p; → qは4.5になる。 c=a+p; → 演算では4.5になるが,           代入時に小数点以下が失われてcは4になる。</pre>
int型とchar型の混合演算	char型をint型に変換してから演算	<pre>int a=3,c; char b=5; の時 c=a+b; → cは8になる。</pre>

## 2. 変数と定数

### (7) エンディアンとは?

2byte以上のデータ型(short int 以上)は複数byteにデータを配置します。

エンディアンとは簡単に言えば「データの並び順」です。バイトオーダーとかバイト順とも言います。

データの先頭byteを小さいアドレスに置く方式⇒ビッグエンディアン、

データの先頭byteを大きいアドレスに置く方式⇒リトルエンディアン です。

**long int型 2882400001 = 0xABCDEF01**

#### メモリ上での配置

	小さい	← アドレス →	大きい
ビッグエンディアン	0xAB	0xCD	0xEF
	1010 1011	1100 1101	1110 1111
リトルエンディアン	0x01	0xEF	0xCD
	0000 0001	1110 1111	1100 1101

アドレス	リトルエンディアン	ビッグエンディアン
0x1000	0x01	0xAB
0x1001	0xEF	0xCD
0x1002	0xCD	0xEF
0x1003	0xAB	0x01

※ アドレスは仮に0x1000番地としています

Intelのマイクロプロセッサ 8086からPentiumシリーズ **リトルエンディアン**

Motorolaのマイクロプロセッサ **ビッグエンディアン**

MIPS や ARM、SHなど どちらにもなれる(**バイエンディアン**)

#### エンディアンの変換関数 (endian.h のインクルードが必要)

ホストバイトオーダーはホストマシンのエンディアン (CPU依存)、ネットワークバイトオーダーはビッグエンディアンのことを指します

uint32_t htonl(uint32_t hostlong)	32bitのホストバイトオーダーをネットワークバイトオーダーに変換する
uint16_t htons(uint16_t hostshort)	16bitのホストバイトオーダーをネットワークバイトオーダーに変換する
uint32_t ntohl(uint32_t netlong)	32bitのネットワークバイトオーダーをホストバイトオーダーに変換する
uint16_t ntohs(uint16_t netshort)	16bitのネットワークバイトオーダーをホストバイトオーダーに変換する

## 2. 変数と定数

### (8) 定数

C言語における定数とは、プログラム中で変化することのない一定の値を持つデータのことを指します。

定数にはconst変数を使う場合と#defineマクロ、それから列挙型を使う方法があります。

#### 1. const変数を使う場合

const変数とは変数の宣言、記憶域クラス名のページに出てきましたが、宣言時にのみ初期化可能で、その後は値を変えられない変数です。

この性質上const変数は定数として使うのが一般的です。

const 変数の 宣言

const int x=3;

int const x=3;



このように const は型名の前か後に付けることができます。

#### 2. #defineマクロを使う場合

#define マクロも定数として使う事が出来ます。

例えば次のような使い方があります。

```
#define MAX_VAUE 50000
```

```
#define MIN_VALE 100
```

```
#define DIM_NUMS 10
```

#defineマクロで定義した定数は変数と区別するために **大文字** を使うことを推奨します。

サンプルプログラム

<https://paiza.io/projects/kQvspcOHmyDWfZ9dUZJe6g>



## 2. 変数と定数

マクロを使う場合の注意点：「置換する文字列はカッコで囲む」  
以下のようなマクロ定義を使ったプログラムの場合

```
#define COST 100
#define PRICE COST + 10
int main(int argc, char *argv[])
{
    int num = 20;
    int total;
    total = PRICE * num;
    printf("%d¥n", total);
    return 0 ;
}
```

total は  $100 + 10 * 200$  となります。

意図している計算は  $(100 + 10) * 200$  です。

これはマクロが単純に文字を置き換えるだけの機能しかない為に起こるミスです。

この場合 マクロ定義部分を

```
#define COST 100
#define PRICE (COST + 10)
```

と書き換えれば

意図した計算になります。

このようにマクロの置換後の字句は、カッコで囲むようにしましょう。

サンプルプログラム

<https://paiza.io/projects/kQvspcOHmyDWfZ9dUZJe6g>

## 2. 変数と定数

### 3. 列挙型を使う場合

列挙型は、複数の変数に一連の整数値を付けると便利な場合に使用します。

列挙型の例

```
enum week {  
    Mon,  
    Tue,  
    Wed,  
    Thu,  
    Fri,  
    Sat,  
    Sun  
};
```

#defineで定義する場合

```
#define Mon 0  
#define Tue 1  
#define Wed 2  
#define Thu 3  
#define Fri 4  
#define Sat 5  
#define Sun 6
```



つまり上のメンバから順に0から始まる数字がふられます。

また、始まりの数字や途中から数字を変えることも出来ます。

```
enum month {  
    Jan = 1,    /* 1からスタート */  
    Feb,  
    Mar,  
    Apr,  
    May,  
    Jun,  
    Jul = 10,   /* 10設定 */  
    Aug,  
    Sep,  
    Oct,  
    Nov,  
    Dec  
};
```

サンプルコード

[https://paiza.io/projects/E6jRobk9ePKe8X\\_IP8gt1Q](https://paiza.io/projects/E6jRobk9ePKe8X_IP8gt1Q)  
<https://paiza.io/projects/BQ15iEynVyTMd5s6Fwg9Lg>

## 2. 変数と定数

### (9) 別名定義 typedef

typedef を用いると既存のデータ型に新しい名前をつけることができます。

書式 : **typedef 既存の型名 新しい型名;**

例 : typedef unsigned short int u16i;

これは **u16i** を unsigned short int型として別名で定義しています。

プログラム中に

**u16i a;**

と変数を宣言すると これは

unsigned short int a; と宣言したのと同じことになります。

Typedef は #define と似ていますが最後に ; が必要です。

変数の型の別名に関しては標準ヘッダー  
**stdint.h**  
でも定義されてます  
P21参照

**typedef は 構造体に名前をつける場合も良く使われます。**

```
// 構造体の宣言
struct employee {
    char name[12];
    int age;
};
struct employee yamada;
// struct employee型の変数 yamada を作成
```



```
// 構造体の宣言 と typedef 定義
typedef struct employee {
    char name[12];
    int age;
} EMPLOYEE;
EMPLOYEE yamada;
// EMPLOYEE 型の変数 yamada を作成
```

## 2. 変数と定数 演習 1 変数を表示する

変数の表示にはC言語の標準関数である `printf()` を使用します、またキーボードからの入力には`scanf()`関数を使います  
`printf()`と`scanf()`に関しては 別途説明していますのでそちらを参照してください。

(1) キーボーから任意の整数を入力してその数字を

`char`、`unsigned char`、

`short int`、`unsigned short int`、

`float`

で表示してください。

(2) キーボーから任意の実数(小数点あり)を入力してその数字を

`char`、`unsigned char`、

`short int`、`unsigned short int`、

`float`

で表示してください。

(3) キーボーから任意の文字(1文字)を入力してそれを

`unsigned char`、文字

で表示してください。

## 3. 式と演算子

- (1) 演算子の種類
- (2) 演算子の優先順位と結合規則
- (3) ビット演算子
- (4) 論理シフトと算術シフト
- (5) インクリメント・デクリメント演算子
- (6) sizeof 演算子

## 3. 式と演算子

### (1) 演算子の種類

演算子は演算の内容を指示する記号で、式は定数、変数、関数の返却値などを演算子を使って結合したものです。

算術演算子	構文	説明
+	$x + y$	xにyを加えます。
-	$x - y$	xからyを引きます
*	$x * y$	xにyを掛けます
/	$x / y$	xをyで割ります
%	$x \% y$	xをyで割った余りです

インクリメント・デクリメント演算子	構文	説明
++	++x	xに+1してxを評価します。(前置演算)
	x++	xを評価したあとxに+1します。(後置演算)
--	--x	xに-1してxを評価します。(前置演算)
	x--	xを評価したあとxに-1します。(後置演算)

ビット演算子	構文	説明
&	$x \& y$	論理積 (AND) を行います
	$x   y$	理和 (OR) を行います
^	$x \wedge y$	排他的論理和 (XOR) を行います
~	~x	否定 (NOT) を行います


シフト演算子	構文	説明
<<	$x << n$	xを左にnビットシフトさせます
>>	$x >> n$	xを右にnビットシフトさせます

※ sizeof も演算子として扱います

代入演算子	構文	説明
=	$x = y$	yをxに代入します
+=	$x += y$	xに ( $x + y$ )
-=	$x -= y$	xに ( $x - y$ ) を代入します
*=	$x *= y$	xに ( $x * y$ ) を代入します
/=	$x /= y$	xに ( $x / y$ ) を代入します
%=	$x \% = y$	xに ( $x \% y$ ) を代入します
&=	$x \& = y$	xに ( $x \& y$ ) を代入します
=	$x   = y$	xに ( $x   y$ ) を代入します
^=	$x \wedge = y$	xに ( $x \wedge y$ ) を代入します
<<=	$x << = y$	xに ( $x << y$ ) を代入します
>>=	$x >> = y$	xに ( $x >> y$ ) を代入します
比較演算子	構文	説明
==	$x == y$	xとyは等価である
!=	$x != y$	xとyは等価ではない
<	$x < y$	xはyより小さい (未満)
<=	$x < = y$	xはy以下である
>	$x > y$	xはyより大きい
>=	$x > = y$	xはy以上である
論理演算子	構文	説明
&&	$a \&\& b$	aとbが共に真の場合「真」
	$a    b$	aまたはbが真の場合「真」
!	!a	aが偽の場合「真」、aが真の場合「偽」

# 3. 式と演算子

## (2) 演算子の優先順位と結合規則

種類線順位を	演算子	結合規則	優先順位
関数, 添字, 構造体メンバ参照,後置増分/減	() [] . -> ++ --	左→右	高
前置増分/減分, 単項式※	++ -- ! ~ + - * & sizeof	左←右	
キャスト	(型名)	左→右	
乗除余	* / %		
加減	+ -		
シフト	<< >>		
比較	< <= > >=		
等値	== !=		
ビットAND	&		
ビットXOR	^		
ビットOR			
論理AND	&&		
論理OR			
条件	?:		
代入	= += -= *= /= %= &= ^=  = <<= >>=	左←右	
コンマ	,	左→右	低

※ 式が複雑になる優先順位が分かりにくくなるので ( ) を使って優先順位を明確にする

## 3. 式と演算子

### (3) ビット演算子

ビット演算とは、2進数の0か1で表現するビット単位で計算することです。フラグの確認でよく使われていたりレジスタの設定等で良く使われます。

#### ① & AND

両方 1 のときに 1、それ以外は0にする演算子

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

$$C=A\&B$$

#### ③ ^ XOR

両方が異なる値の時に 1、それ以外は0にする演算子

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

$$C=A\wedge B$$

#### ⑤ << 左シフト演算子

A << 1と表記され、2進数で表記されたAを左へ1ビット桁移動をする演算子です。左側にあふれたビットは削除され、右側に0が追加されます。

$$A<<2$$

$$0b11001110(0xC7) \Rightarrow 0b00111000(0x38)$$

#### ② | OR

どちらかが 1 のときに 1、それ以外は0にする演算子

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

$$C=A|B$$

#### ④ ~ NOT

反転させる演算子

A	C
0	1
1	0

$$C=\sim A$$

#### ⑥ >> 右シフト演算子

A >> 1と表記され、2進数で表記されたAをひだり右へ1ビット桁移動をする演算子です。右側にあふれたビットは削除され、左側に0が追加されます。

$$A>>2$$

$$0b1100111(0xC7) \Rightarrow 0b00111001(0x39)$$



## 3. 式と演算子

### (4) 論理シフトと算術シフト

ビットシフト演算子で時々問題になるのが、**符号ビット**が1となっている場合の右シフト演算です。

例えば

**0b11110010** を 1ビット右シフトした場合、**0b01111001** となると予想できますが、

この変数がsigned型charの型を持っていたら

**0b11111001** となり先頭ビットに 1 がセットされます。

これはsigned型の変数の場合 先頭ビットは 符号ビット として使われているためです。

元々の数字が **-14** だったために右シフトしても最上位ビット(符号ビット)を1にしています。

**0b11111001** の10進数表記は **-7** なので右シフトして **元の値の 1/2** になっていて

右シフトしたら 元の値の 1/2 になる法則にもあっていることになります。

このように符号ビットを反映したシフトを **算術シフト** といいます。

**算術シフト**対して符号ビット無視したシフトを **論理シフト** といいます。

ただし、算術シフトは右シフトのみで左シフトに関しては適応されません。

	2進数	>> 1	
符号なし(unsigned)	0b11111001	0b <b>0</b> 1111001	論理シフト
符号有り(signed)	0b11111001	0b <b>1</b> 1111001	算術シフト

サンプルコード

右シフト : <https://paiza.io/projects/rY6rESkebOtJRj61aoex4g>

左シフト : <https://paiza.io/projects/UXexWBetIpZRvO0i5jpN6w>

## 3. 式と演算子

### (5) インクリメント・デクリメント演算子

インクリメント演算子(++)とデクリメント演算子(--)には2通りの使い方があります。

(++, --共通の内容なので ++ を使って説明します)

演算子	名前	使い方	
++	前置インクリメント演算子	++i	変数「i」の値を +1
++	後置インクリメント演算子	i++	変数「i」の値を +1

この表だとどちらも同じに見えますが、インクリメント演算子の変数の前か後ろにあるかで動きが違ってきます。

#### 前置・後置インクリメント演算子の違い

演算子	コード	説明
前置インクリメント	y = ++x;	xの値をインクリメント (+1) し、yにxの値を代入
後置インクリメント	y = x++;	xの値をインクリメント (+1) するが、 <b>インクリメントする前のx</b> の値をyに代入

このようにインクリメント後に処理を行うか、インクリメント前に処理を行うかの違いがあります。

サンプルコード

[https://paiza.io/projects/gAnBedSUMB3bf\\_8OLP0YPQ](https://paiza.io/projects/gAnBedSUMB3bf_8OLP0YPQ)

## 3. 式と演算子

### (6) Sizeof演算子

sizeof演算子は、変数や型のメモリサイズを調べるための演算子で、変数や型のメモリサイズをバイト単位(1バイトは8ビット)で返してくれます。

ほとんどの言語には、配列の要素数を求めるためのマクロやメソッドが用意されていますが、

C言語ではsizeof演算子を使って、配列の全体のサイズを求めます。

またsizeof演算子はその他にも、構造体のサイズやポインタのサイズを取得するためにも使われます。

C言語以外のsizeofは配列の要素数を返す場合が多いですが C言語の場合は 要素数ではなくその配列が使っているメモリのサイズが返ります。

そのためC言語の場合は配列の要素数を求めるには配列全体のメモリサイズを配列の型のサイズで割る必要があります。

例えば

```
short int a[100];
```

の要素数は **sizeof a** (配列全体のメモリサイズ) / **sizeof a[0]** (配列の1要素分のメモリサイズ) で求められます。

sizeof演算子は指定したデータ型や変数のサイズを返しますが、その値はint型ではありません。

size\_tという標準型と呼ばれる特殊な型で、これは<stddef.h>ライブラリで定義されていますが、

通常は<stdio.h>や<stdlib.h>をインクルードすれば使えるようになります。

サンプルコード

<https://paiza.io/projects/woM9-t8-4kXTsf3XBx8iOA>

## 4. 配列

- (1) 配列とは
- (2) 配列の定義
- (3) 配列の初期化
- (4) 配列へのアクセス

## 4. 配列

### (1)配列とは

配列は表にしたいとなるようなデータのように多数の同じ用途のデータを取り扱う場合に使います。

例えばあるクラスの成績を扱う変数を定義する場合

配列を使わないと

```
int seiseki_1=70, seiseki_2=50, seiseki_3=80, seiseki_4=90, seiseki_5=20;
```

この平均をとる場合

```
int heikin = (seiseki_1+ seiseki_2+ seiseki_3+ seiseki_4+ seiseki_5)/5;
```

となります。

ここで生徒が2人増えた場合は

```
int seiseki_1=70, seiseki_2=50, seiseki_3=80, seiseki_4=90, seiseki_5=20 , seiseki_6=56, seiseki_7=100 ;
```

```
int heikin = (seiseki_1+ seiseki_2+ seiseki_3+ seiseki_4+ seiseki_5 + seiseki_6+ seiseki_7)/7;
```

と変更箇所が複数あります。

これを配列にすると

```
int seiseki[] = {70,50,80,90,20};
```

```
int nums = sizeof(seiseki)/sizeof(seiseki[0]);
```

```
int sum = 0;
```

```
int heikin=0;
```

```
int i;
```

```
for(i=0; i<nums;i++){
```

```
    sum += seiseki[i];
```

```
}
```

```
heikin = sum/ nums;
```



生徒が2人増えた場合

```
int seiseki[] = {70,50,80,90,20,56,100};
```

```
int nums = sizeof(seiseki)/sizeof(seiseki[0]);
```

```
int sum = 0;
```

```
int heikin=0;
```

```
int i;
```

```
for(i=0; i<nums;i++){
```

```
    sum += seiseki[i];
```

```
}
```

```
heikin = sum/ nums;
```

このように変更箇所も少なくなります。

## 4. 配列

### (2) 配列の定義

配列を使用する場合も変数と同様に定義が必要です。

配列の定義は以下のように行います。

**データ型名 配列名 [要素数];**

例

```
short int seiseki[10];
```

この場合 short int 型の変数が10個分確保されます。

```
seiseki[0] seiseki[1] seiseki[2] seiseki[3] seiseki[4] seiseki[5] seiseki[6] seiseki[7] seiseki[8] seiseki[9]
```

配列の最初の要素番号は0から始まり最終データの要素番号は定義で指定した要素数から **1引いた数字** となります。

要素数の指定は結果が定数となる定数または定数式となります。

~~変数を含むことは出来ません~~ **C99以降は変数も仕様出来るようになりました。**

- 正の整数のみとなり、負の数や実数は使えません

例		意味
double array_d[30];	○	double型の30要素の配列を確保
long int array_l[4*10]	○	long int型の4*10要素(40要素)の配列を確保
char array_c[4*size]	×	要素数の指定に変数は使えない
float array_f[3.5]	×	要素数の指定に実数は使えない
short int array_i[-10]	×	要素数の指定に負の数は使えない

## 4. 配列

### (3) 配列の初期化

配列の初期化に関して

- 配列は**定義時のみ**、まとめて代入できます
- 初期化したい値を「,」(カンマ)で区切り中括弧でかこみます

例 `short int a[10] = {1,2,3,4,5,6,7,8,9,10};`

- 指定した要素数を超えるとエラーになります

例 `short int a[5] = {1,2,3,4,5,6,7,8,9,10}; /* エラーとなる */`

- 配列を宣言時に初期可する場合は要素数の指定は省略できます

例 `short int a[] = {1,2,3,4,5,6,7,8,9,10};`

逆に初期化する場合は要素数は指定しない方が望ましい

これは `short int a[10] = {1,2,3,4,5,6,7,8,9,10};` と定義していて要素を2つ追加する場合

`short int a[12] = {1,2,3,4,5,6,7,8,9,10,11,12};`

と変更しますが、要素数を変更するのを忘れる場合が良くあるためです。

`short int a[] = {1,2,3,4,5,6,7,8,9,10};` と書いた場合 `short int a[] = {1,2,3,4,5,6,7,8,9,10,11,12};` と要素数の変更を行う必要がありません。

- 指定した要素数に満たない個数の値を与えた場合残りは 0 で初期化されます

例 `short int a[10] = {}; /* すべて 0 で初期化されます */`

## 4. 配列

### (4) 配列へのアクセス

- 要素毎への代入

配列名[添え字] = 値 ;

例 a[0] = 1;

a[1] = 2;

- 配列毎の参照

配列名[添え字]

例 b = a[0];

- 配列全体への参照、代入は不可

例

```
short int a[] = {1,2,3,4,5};
```

```
short int c[5];
```

c = a ; ← このように **aの配列全体をcの配列全体**に代入する処理はできません。

配列 a を配列 c にコピーしたい場合は **for文**等を使って 1要素ずつコピーします。

- 処理としての一括代入は不可

例

```
short int a[5] ;
```

```
a[] = {1,2,3,4,5}; ←この処理は出来ません、まとめて代入出来るのは定義時のみ有効です */
```



## 5. 文字と文字列

- (1) C言語での文字列
- (2) ‘(シングルクォーテーション) と “(ダブルクォーテーション) の違い
- (3) 文字列の最後にはNULL文字(‘\0’) が必要
- (4) 文字列リテラル

## 5. 文字と文字列

### (1) C言語での文字列

char型とは**1文字(1バイト)**を格納するデータ型です。

char型の変数に値を代入する際には代入する文字を「'」(シングルクォーテーション)でかこみます。

ただし、char変数には文字だけでなく8ビット長の変数として符号ありの場合 **-128~127** の値、符号なしの場合 **0~255** の値を扱うことができます。

日本語の全角文字は**2バイト以上**の文字となりますので、char型の変数には格納できません。

C言語での文字列は **char型の配列** を使います。

数値の配列と同じように配列名のあとに **[ ]** 記号を使って文字列サイズを指定し宣言する方法と、ポインタを使って宣言する方法があります。

配列として宣言する場合

char str\_1[]="Hello" ;                      ← '¥0' は書かなくても自動で入る

char str\_2[]={ 'H','e','l','l','o','¥0' } ;    ← '¥0' を書く

どちらも同じ意味となりメモリの中は 右の図のようになります。

**最後に '¥0' が入っているのに注意してください。**

アドレス	1000	1001	1002	1003	1004	1005
データ(文字)	'H'	'e'	'l'	'l'	'o'	'¥0'
データ(16進数)	0x48	0x65	0x6c	0x6c	0x6f	0x00

C言語の文字列は必ず最後に **'¥0'** が入ります。配列の要素数としては**文字数 +1('¥0'の分)** となります。

char str\_1[]="Hello" ; と宣言した場合配列の要素数は6が自動的にセットされます。

ポインタを使って宣言する方法に関しては **(4) 文字列リテラル** を参照してください。

## 5. 文字と文字列

### (2) ‘(シングルクォーテーション) と“(ダブルクォーテーション)の違い

#### A) “(ダブルクォーテーション)

文字(文字列)を“で囲むとそれは**文字列**として扱われます。

例：“Hello” → 0x48,0x65,0x6c,0x6c,0x6f, '¥0'

例：“H” → 0x48, '¥0'

例：“ハロー” → 0x83,0x6e,0x83,0x8d,0x8, 0x5b,0x00 (Shift-JIS)

このように文字列の最後には**NULL文字(¥0(0x00))**が入ります。

#### B) ‘(シングルクォーテーション)

文字(文字列)を‘で囲んでも文字列として扱われません。

例：‘H’ → 0x48

例：‘Hello ‘ → 複数の文字は使えません

例：‘ハ’ → 全角文字は最低2バイト以上必要なので複数文字扱いとなり、使えません

## 5. 文字と文字列

### (3) 文字列の最後にはnull文字('¥0') が必要

文字列の終わりには必ずnull文字('¥0')が必要です。

これは文字列の最後を示す重要な役割を担っています。

標準関数の中の文字列操作関数もすべて、このnull文字を文字列の最後として処理が行われます。

文字列とnull文字

A) 文字列を入れる配列サイズを決めるときは、null文字の分も考慮します

```
char passwd[9];
```

8文字を保存する文字配列なら 8+1文字分必要となります

B) 文字配列の初期化では、コンパイラによって null文字が自動的にセットされます

```
char str[] = "Hello";
```

strにはHelloの後ろにnull文字をつけた内容が入ります

C) 文字配列にひと文字ずつ入れるなら、null文字も指定します

```
char str[] = { 'H', 'e', 'l', 'l', 'o', '¥0' };
```

サンプルコード

[https://paiza.io/projects/\\_Hj23PSBHFfsMqGwM78IAA](https://paiza.io/projects/_Hj23PSBHFfsMqGwM78IAA)

## 5. 文字と文字列

### (4) 文字列リテラル

文字列リテラルとは、0文字以上の連続した文字列を示す定数のことです。

(1) `char str[] = "abc";`

(2) `char *str = "abc";`

この2つは同じ処理のように見えますが、(1)が**配列の初期化**で、(2)が**ポインタの初期化**です。

(1)は文字の配列のため、書き換えが可能です。

```
char str[] = "abc";
```

```
str[0] = 'd'; /* 可能 */
```

(2)の場合における“abc”は、「char型の配列」ですが、「char型のポインタ」がstrに代入されます。

strは“abc”のアドレスを格納するポインタ変数です。

こちらは書き換えが不可です。

文字列リテラルは、通常は書き換え禁止領域に確保されるので、書き換えようとすると書き込み禁止のエラーがおきます。

```
char *str = "abc";
```

```
str[0] = 'd'; /* エラーとなります */
```

## 6. 2次元配列

- (1) 2次元配列とは
- (2) 2次元配列の定義
- (3) 2次元配列の初期化
- (4) 2次元配列へのアクセス
- (5) 2次元配列のアドレスの求め方

## 6. 2次元配列

### (1) 2次元配列とは

2次元配列は1次元配列を1つの行として複数の行を持つ配列で、Excelの表みたいなデータを扱う時に使います。

### (2) 2次元配列の定義

要素の型 配列名[行数][列数]

```
char meibo[10][20];
```

これは10人分の名前が入る配列となります。

行数、列数の指定は1次元配列と同じで正の整数のみが指定できます。

また、要素の指定も1次元配列と同じように0から始まります。

### (3) 2次元配列の初期化

初期化したい列の値を「,」で区切って中括弧で囲みます。

次に列を「,」で区切ったものを並べて中括弧で囲みます。

```
short int[4][5] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15},{16,17,18,19,20}};
```

なお定義時に初期化を行う場合は行数のみ省略できます。

```
short int[][5] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15},{16,17,18,19,20}};
```

列数は省略できません。

## 6. 2次元配列

### (4) 2次元配列のアクセス

2次元配列へのアクセスは配列名[添字1][添字2]の形式でおこないます。  
次の例はfor文の2重ループで処理する例です。

```
short int a[10][5];
short int x,y;
for(x=0;x<10;x++){
    for(y=0;y<5;y++){
        a[x][y] = 数字;
        変数 = a[x][y];
    }
}
```

### (5) 2次元配列のアドレスの求め方

2次元配列のアドレスの求め方

short int a[10][5] という2次元配列のアドレスは  
配列 a のアドレス      a ← 配列名のみ  
各行の先頭アドレス      a[行添え字]  
各要素のアドレス      &a[行の添字][列の添字]

サンプルプログラム

<https://paiza.io/projects/87xReNpHSaOfNv0sVGuDZw>



## 7. 変数とポインタ

- (1) ポインタとは
- (2) マイコンにおけるアドレス
- (3) ポインタ演算子
- (4) ポインタ変数の宣言
- (5) ポインタ変数へのアドレスの代入
- (6) ポインタ変数を使ったデータへのアクセス
- (7) 文字型配列とポインタ
- (8) ポインタの配列
- (9) ポインタのポインタ
- (10) 関数のポインタ
- (11) void型へのポインタ

# 7. 変数とポインタ

## (1) ポインタとは

ポインタは関数や変数、配列などが格納されているメモリ領域の先頭番地を記憶するための変数です。ポインタの主な用途としてはポインタ変数を介した間接的なメモリアクセスです、例えば

- 実行時に要素数を指定する配列の実現
- リスト・ツリーなどの柔軟かつ複雑なデータ構造の実現
- 関数から複数の値を返す
- 文字列処理、配列処理

などがあります。

## 7. 変数とポインタ

### (2) マイコンにおけるアドレス

マイコンのメモリには図(左)のようにアドレス(番地)がふられています。  
変数を宣言するとメモリ上のその領域が確保されるので、当然その変数はアドレスを持ちます。  
通常変数は自動的に配置されるので、そのアドレスも自動で振られることになります。

例えば

```
short int a = 100;
```

```
long int b = 10000;
```

```
char c[] = {'a','b','c','d','e','\0'};
```

と定義した場合、図(右)のようにデータが入り、各変数のアドレスが次のように決まります。

a : 0000番地

b : 0002番地

c[0] : 0007番地

c[1] : 0008番地

c[2] : 0009番地

c[3] : 000A番地

c[4] : 000B番地

c[5] : 000C番地

このアドレスがポインタと密接な関係があります。

これらの変数のアドレスは次の書式で参照できます。

変数aのアドレス &a

変数bのアドレス &b

配列c[]のアドレス c

配列の要素c[0]のアドレス &c[0]

配列の要素c[1]のアドレス &c[1]

配列の要素c[2]のアドレス &c[2]

配列の要素c[3]のアドレス &c[3]

配列の要素c[4]のアドレス &c[4]

配列の要素c[5]のアドレス &c[5]

変数名の前に「&」を付けると  
その変数のアドレスを表します。

配列の場合は配列の**配列名**だけでアドレスを表します  
配列の各要素のアドレスは**配列名[要素番号]**の前に  
「&」を付けるとその要素のアドレスを表します。

番地	メモリ
FFFF	
FFFE	
FFFD	
FFFC	
FFFB	
FFFA	
FFF9	
↑	
0011	
0010	
000F	
000E	
000D	
000C	
000B	
000A	
0009	
0008	
0007	
0005	
0004	
0003	
0002	
0001	
0000	

番地	メモリ	変数名
FFFF		
FFFE		
FFFD		
FFFC		
FFFB		
FFFA		
FFF9		
↑		
0011		
0010		
000F		
000E		
000D		
000C	'\0'	c[5]
000B	'e'	c[4]
000A	'd'	c[3]
0009	'c'	c[2]
0008	'b'	c[1]
0007	'a'	c[0]
0005	10000	b
0004		
0003		
0002	100	a
0001		
0000		

## 7. 変数とポインタ

### (3) ポインタ演算子

ポインタを操作するためには、次のポインタ演算子を使います。

- **&** 変数のアドレス（変数データが格納されているメモリ番地）を取り出す

&に関しては前ページにも出て来ましたが、変数の前に付けることで、その変数が格納されているメモリのアドレスを参照できます。

- **\*** 変数の値をアドレスとして捉え、そのアドレスにあるデータを取り出す

```
short int *a;
```

と宣言された変数は `a` がアドレスを格納する変数、`*a`と書くと `a`という変数の値をアドレスとするデータとなります。

```
short int a = 100;
```

```
short int *b = &a;
```

と宣言した場合メモリの内容は以下ようになります。

番地	メモリの値	変数名	説明	備考
0005				
0004				
0003	0000	b	変数aのアドレスが入ります	*bで 0000番地のデータにアクセスできます
0002				
0001	100	a	変数 a の値が入ります	
0000				

`a` と `*b` の値は同じ100になります。

サンプルコード

<https://paiza.io/projects/Q869GzYjFMrG74g0h9l1Ew>

## 7. 変数とポインタ

### (4) ポインタ変数の宣言

ポインタ変数の宣言は次のように書きます。

```
int *a;
```

```
int* a;
```

この2つは、aという名前のintへのポインタ型の変数を宣言する書き方です。

2つ目の方法で

```
int* a,b;
```

と書いた場合は

```
int* a;
```

```
int b;
```

と書いたのと同じになり、2つ目の変数 b は普通のint型変数になってしまいます。

これを避けるために複数の変数を1行でポインタ変数として宣言する場合は

```
int *a,*b;
```

と書くようにしましょう。

## 7. 変数とポインタ

### (5) ポインタ変数へのアドレスの代入

ポインタ変数へのアドレスの代入方法

```
int *p ;
int a;
char st[] = "Hello" ;
char st_dim[][10] = { "one" , " two" , " three" , " four" }

p=&a;                /* 変数aのアドレスをポインタ変数 p に代入 */

p=st;                /* 文字型配列 stの先頭アドレスを p に代入 */
p=&st[0];             /* 文字型配列 stの先頭アドレスを p に代入 */
p=&st[4];             /* 文字型配列 stの4番目(0から数えて)の要素のアドレスを p に代入 */

p= st_dim;           /* 2次元配列 st_dimの先頭アドレスを p に代入 */
p= st_dim[2];         /* 2次元配列 st_dimの2行目の先頭アドレスを p に代入 */
p= &st_dim[2][1];     /* 2次元配列 st_dimの2行目、1列目のアドレスを p に代入 */
```

サンプルコード

<https://paiza.io/projects/UbviHJdTnWf5rebeHv9Lqw>

## 7. 変数とポインタ

### (6) ポインタ変数を使ったデータへのアクセス

要素へのアクセス

```
char array[] = "ABC";  
char*pt = "DEF" ;  
  
printf("arrayの2, 3番目の文字¥n");  
printf("%c", array[1]);  
printf("%c", *(array + 2));  
printf("¥n");  
  
printf("ptの2, 3番目の文字¥n");  
printf("%c", *(pt + 1));  
printf("%c", pt[2]);  
printf("¥n");
```

## 7. 変数とポインタ

### (7) 文字型配列とポインタ

char型の配列とポインタのできること・できないこと

配列とポインタは良く似ていますが、この2つを比べてみます。

初期化

```
char array[] = "ABC"; // 可
char*pt = "DEF";      // 可, const修飾推奨

char array2[] = {'A','B','C','\0'}; // 可
char *pt2 = {'D','E','F','\0'};    // 不可
char *pt2 = (char[]){'D','E','F','\0'}; // 可, 配列の実体ができる

char *pt2 = (char[]){'D','E','F','\0'};
=
char array[]={'D','E','F','\0'};
char *pt2=array;
```

配列・ポインタの相互の代入

```
array = pt; // 不可
pt = array; // 可
```

配列・ポインタの文字列の書き換え

```
char array[4] = "ABC";
array = "GHI"; // 不可
strcpy(array, "GHI"); // 可

char *pt = "DEF";
strcpy(pt, "JKL"); // 不可
pt = "JKL"; // 可
```



## 7. 変数とポインタ

### (8) ポインタの配列

```
int array[10];
```

これは int型の10個の要素を持つ配列です。

```
int *array[10];
```

こちらは 10個のint型のポインタを持つ配列です。

```
char *pastr[3];
```

配列 pastr はあくまでも文字へのアドレスを 3 個持てるだけの メモリ領域しか確保しません。

従って、使うときには次のように します。

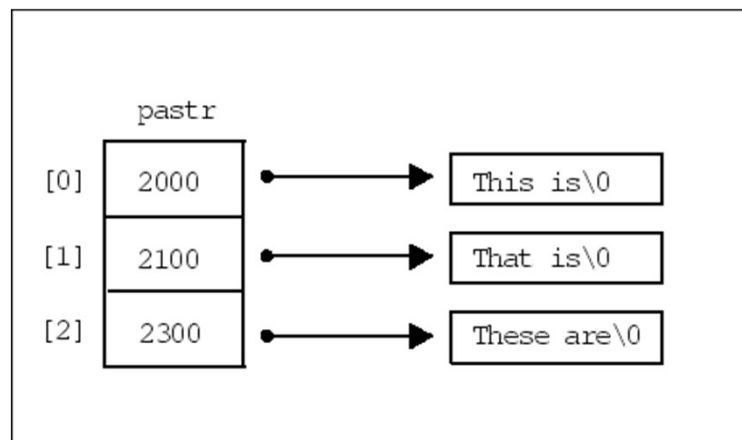
```
char *pastr[3];
```

```
pastr[0] = "This is ";
```

```
pastr[1] = "That is ";
```

```
pastr[2] = "These are ";
```

この時、文字列 "This is" はメモリ上に自動的に配置され、その文字列の アドレス(実際には 'T' のアドレス) が pastr[0] に入っていることになります。



# 7. 変数とポインタ

## (9) ポインタのポインタ

ポインタは変数のアドレスを格納する特殊な変数です。  
ポインタも当然、変数のアドレスをメモリのどこかに保存していることになります。  
つまり、ポインタ変数も独自のアドレスを持つということが考えられます。

これはポインタのアドレスをさらにポインタに格納することが出来るということです。  
このことをポインタのポインタと呼びます。

ポインタのポインタから、参照しているポインタがさらに参照している大元の変数へアクセスすることもできます。  
このような多重間接参照の理解は、ポインタの概念において重要な部分です。

ポインタのポインタを宣言するにはアスタリスク \* をさらに付加します

型 \*\*変数名;

さらに、ポインタのポインタのポインタを作ることも可能です。  
いくつポインタを連鎖してもかまいませんが、ポインタのポインタ以降の連鎖はプログラマ自身が参照先がどこなのかを把握しにくくなるため、  
通常は使用されません  
ポインタの多重間接参照は、連鎖の数だけ \* をつけて宣言します。

サンプルコード

<https://paiza.io/projects/6D3Tc7cw93VyWBKWwYZHWQ>

# 7. 変数とポインタ

## (10) 関数のポインタ

関数ポインタとは、ポインタの関数版です。

ポインタの参照先を関数にすると、そのポインタが指し示す関数を呼び出すことができます。

ポインタに格納する関数のアドレスを変えることで、呼び出す関数を動的に変更することが可能です。

関数ポインタは、アドレスを格納する関数と同じ戻り値の型を使って宣言します。

関数ポインタの仮引数の型と、ポインタに格納する関数の型は一致する必要があります。

関数ポインタを使うときには、ポインタ名に()を付けて使いましょう。

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

```
int addition(int x, int y);    // 足し算  
int subtraction(int x, int y); // 引き算
```

```
int main(void) {  
    int a = 3;  
    int b = 5;  
    int answer;  
    int (*funcp)(int, int);  
  
    funcp = addition;  
    answer = (*funcp)(a, b);  
    printf("a + b = %d\n", answer);  
  
    funcp = subtraction;  
    answer = (*funcp)(a, b);  
    printf("a - b = %d\n", answer);  
  
    return 0;  
}
```

```
int addition(int x, int y) {  
    return (x + y);  
}
```

```
int subtraction(int x, int y) {  
    return (x - y);  
}
```

一回目の関数呼び出し時は、funcpに関数additionのアドレスが格納されていて、二回目の呼び出し時には関数subtractionのアドレスが格納されています。

このようにしてポインタに関数のアドレスを格納することで、呼び出す関数を変更することができます。

サンプルコード

[https://paiza.io/projects/kbfq74KHqv6viw\\_NKcRbOQ](https://paiza.io/projects/kbfq74KHqv6viw_NKcRbOQ)

## 7. 変数とポインタ

### (11) void型へのポインタ

```
int array[10];
```

これは int型の10個の要素を持つ配列です。

```
int *array[10];
```

こちらは 10個のint型のポインタを持つ配列です。

## 8. 型変換

- (1) 型変換とは?
- (2) 変数の型変換
- (3) 暗黙の型変換
- (4) 汎整数拡張
- (5) ポインタの型変換
- (6) 定数おけるキャスト

## 8. 型変換

### (1) 型変換とは?

変数やオブジェクトを宣言時に定義にした型とは別の型に変えることを言います。

型の変換には明確に記述しなくても行われる暗黙的型変換と記述して行う明示的型変換とがあります。

明示的型変換のことを特にキャストと言います。

#### int型とdouble型の型変換

```
int main(void) {  
    int num = 3;  
    double result;  
  
    // キャストを行う場合  
    result = 1 / (double)num;  
    printf("double型変数resultの値は: %lf\n", result);  
  
    // キャストを行わない場合  
    result = 1 / num;  
    printf("double型変数resultの値は: %lf\n", result);  
  
    return 0;  
}
```

#### 実行結果

```
double型変数resultの値は: 0.333333  
double型変数resultの値は: 0.000000
```

## 8. 型変換

### (2) 変数の型変換

キャストは、ある型の変数を別の型に変換する場合に使います。

`(float)a;`

例えばこれは変数aをfloat型にキャストしています。

マイナスの値の変数を符号なし型の変数にキャストしてもマイナスを取った値にはなりません。

意図しない値となります。

#### 例

```
int main(void) {
    char chr = -100;
    unsigned char uchr;

    // 符号付きchar型変数から符号なしchar型変数へのキャスト
    uchr = (unsigned char)chr;
    printf("unsigned char型変数uchrの値は: %u\n", uchr);

    return 0;
}
```

#### 実行結果

unsigned char型変数uchrの値は: 156

## 8. 型変換

### (3) 暗黙の型変換

明確に記述しなくても、代入や式中で自動的に行われるキャスト変換が暗黙のキャスト変換です。

代入記号「=」の左辺の型と右辺の型が違う場合、左辺の型に変換されます。

1. 一方が long double なら、他方を long double に型変換する。
2. 一方が double なら、他方を double に型変換する。
3. 一方が float なら、他方を float に型変換する。
4. 一方が unsigned long int なら、他方を unsigned long int に型変換する。
5. 一方が long int で、他方が unsigned int で、更に long int が unsigned int のすべてが表現できるならば、long int に型変換する。
6. long int が unsigned int のすべてを表現できないならば、unsigned long int に型変換する。
7. 一方が long int なら、他方を long int に型変換する。
8. 一方が unsigned int なら、他方を unsigned int に型変換する。
9. 上記に該当しない場合には、int に型変換する。

double型 > float型 > long long(int)型 > long (int)型 > int型 > short (int)型 > char型

(※処理系によっては long(int)型と int型は同じです)



## 8. 型変換

### (4) 汎整数拡張

C の規格では、整数に対して算術演算子またはビット演算子を適用しようとしたとき、コンパイラは先にすべての項の型を「昇格」させてから演算子を適用するというルールがあります。これを汎整数拡張といいます。

汎整数拡張では、すべての整数型は符号の有無にかかわらず、収容できる限り int に変換されます。つまり次の型はすべて、演算子を当てた瞬間 int に変換されてしまうということです。

- signed char
- unsigned char
- short
- unsigned short

符号拡張

昇格前の型	昇格前の値	昇格後の値
signed char	0x40	0x00000040
signed char	0x80	0xFFFFFFFF80
unsigned char	0x80	0x00000080
signed short	0x4000	0x00004000
signed short	0x8000	0xFFFF8000
unsigned short	0x8000	0x00008000

符号あり整数の昇格では、符号の維持のため、最上位ビットを上位側に敷き詰めるような拡張がなされます。これを符号拡張といいます。

## 8. 型変換

### (5) ポインタの型変換

ポインタも、キャストすることができます。特に変数を引数に入れるとき、引数として受け取った変数を使用する前などでよく使われます。

関数の引数にvoid\*を使用することがあります。これは、適当なポインタを入れろ、という意味なので、文字どおり適当なポインタを入れればよいわけですが、その引数を関数内で使う場合は、そのままでは使用できません。そこで、ポインタのキャストを行います。

```
void func(void* t) {
    int *i;
    /* tの(void*)→(int*)キャスト */
    i = (int *)t;
    printf("%d\n", *i);
}

int main(int argc, char* argv[]) {
    int i;
    i = 2;
    func(&i);
    return 0;
}
```

結果、2

## 8. 型変換

### (6) 定数おけるキャスト

#define で定数を定義するときは下記のように記述して、定数の型を明示的に指示します。

int型

#define a **0**

long int型

#define a **0L**

float型

#define a **0.0**

unsigned int型

#define a **0U**

## 9. 構造体

- (1) 構造体とは?
- (2) 構造体とポインタ
- (3) ビットフィールド

## 9. 構造体

### (1) 構造体とは？

構造体とは、「いろいろな種類のデータをまとめて、1つのかたまりにしたもの」です。

たとえば、「名前, 性別, 年齢, 身長, 体重」などのデータを一人分だけまとめたもののことを言います。

構造体を構成する要素を、構造体のメンバと呼び上の例では、「名前」「性別」「年齢」「身長」「体重」などが、メンバにあたります。

#### 1. 構造体の宣言方法

構造体は、一つのデータ型であり、その型枠を初めに宣言する必要があります。

その後、その型枠を型とする変数を宣言して構造体の実体（オブジェクト）を使用できるようにします。

構造体の型枠の宣言と、その型枠をもつ構造体変数の宣言は次のように行います。

```
struct 構造体タグ名 {メンバの並び}; /* 型枠の宣言 */  
  
struct 構造体タグ名 構造体変数名; /* 構造体変数の宣言 */
```

```
struct Person {  
    char name[20];  
    int sex;  
    int age;  
    double height;  
    double weight;  
};  
  
struct Person p;
```

## 9. 構造体

### 2. 構造体メンバの初期化

構造体は変数定義時のみ、構造体のメンバーすべてをまとめて初期化できます。

```
struct 構造体タグ 変数名 = {メンバぬの値, メンバ2の値, ……., メンバーnの値}; /  
struct person_s p = {"Tom", 0, 20, 175.2, 66.5};
```

### 3. 構造体メンバへのアクセス

構造体変数名. メンバ名

```
a.customer_id  
b.object_id
```

```
a.customer_id = 30;  
b.object_id   = 10;  
  
customer_id = a.customer_id;  
object_id   = b.object_id;
```

### 4. 構造体の代入

同じ型を用いた構造体変数同士であれば、構造体を代入することができます。

```
person_s p1 = {"Tom", 'M', 19, 175.2, 69.5};  
person_s p2;  
person_t p3;  
  
p2 = p1; // 代入OK  
p2 = p3; // 代入NG
```

## 9. 構造体

### 5. 構造体の配列

構造体もintなど他の型の変数と同様に配列を用いて宣言することができます。

```
#define PERSON_NUM 5  
  
person_s p[PERSON_NUM];
```

p[3]のnameにアクセスしたい場合はp[3].nameでアクセスすることができます。

また構造体配列の初期化は以下のようにして行えます。

```
person_s p[PERSON_NUM] = {  
    {"Bob", 'M', 19, 165.4, 72.5},  
    {"Alice", 'F', 19, 161.7, 44.2},  
    {"Tom", 'M', 20, 175.2, 66.3},  
    {"Stefany", 'F', 18, 159.3, 48.5},  
    {"Leonardo", 'M', 19, 172.8, 67.2}  
};
```

## 9. 構造体

### (2) 構造体とポインタ

サイズの大きな構造体変数を関数に値渡しするのは合理的ではありません。

時間もかかるし、容量（スタックといいます）も必要となります。

構造体変数は値渡しより参照渡しの方が望ましいということになります。

構造体変数を参照渡しするには構造体変数のポインタが必要です。

構造体のポインタの宣言方法は、通常のポインタ同様に \* 演算子を使用します

**struct tag-name \*reference;**

各要素にアクセス方法ですが、構造体のポインタはドット演算子は使えません。

代わりに、構造体変数がポインタであることを明示的に表すアロー演算子 ( -> ) を用います。

アロー演算子は マイナス記号 - と大なり記号 > で構成されます。

**reference->member**



## 9. 構造体

ポインタによる構造体変数の参照渡しであれば、実行速度に深刻な問題を及ぼすことなく、構造体を関数で処理することによって作業の効率化をはかれます。

```
struct LOVE_HINA {
    char *name;
    int age;
};

struct LOVE_HINA LOVE_HINA(char *, int);

void write(struct LOVE_HINA *);

int main() {
    struct LOVE_HINA naru = LOVE_HINA("成瀬川なる", 17);
    struct LOVE_HINA sinobu = LOVE_HINA("前原しのぶ", 13);
    printf("名前\t\t年齢\n");
    write(&naru);
    write(&sinobu);

    return 0;
}

struct LOVE_HINA LOVE_HINA(char name[], int age) {
    struct LOVE_HINA obj;
    obj.name = name;
    obj.age = age;
    return obj;
}

void write(struct LOVE_HINA *obj) {
    printf("%s\t%d\n", obj->name, obj->age);
}
```

```
名前    年齢
成瀬川なる 17
前原しのぶ 13
```

write()関数の定義を見てください

仮引数では、ポインタで構造体変数を受け取っています  
そして、printf()関数の引数ではポインタ型構造体変数なのでアロー演算子でメンバにアクセスしています

main()関数からwrite()関数の呼び出し時には、構造体変数のアドレスを渡していることも注目してください

## 9. 構造体

### (3) ビットフィールド

通常構造体のメンバーとなる変数の最少はchar型の8ビットです。

ただ、ビットフィールドを使うと1ビット単位の長さの変数を定義できるようになります。

プログラマ側で変数の領域(長さ)を定めたい時にビットフィールドは有効です。

ビットフィールドの定義は、構造体の定義とほぼ同じです

ビットフィールドで使われる型は、通常intまたはunsignedです（signedの場合は上位ビットが符号に使われます）

なお、ビットフィールドのアドレスを取得することはできません。

#### 1. ビットフィールドの定義

型 名前: サイズ;

```
struct 構造体名 {
    データ型 メンバー名: サイズ;
    .
    .
    .
} 変数名;
```

```
struct BF {
    unsigned int a : 10;
    unsigned int b : 8;
    unsigned int c : 12;
    unsigned int d : 2;
};
```

メモリ上の配置

BIT	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
変数	a										b								c												d	

## 9. 構造体

次の例はビットフィールドを共用体の中で使っています。

これは組み込みプログラムの中で、ペリフェラル(周辺IC)のレジスタの設定やportのON/OFFに良く使われています。

```
union{
    int port_a;
    struct{
        short  b16:1;
        short  b15:1;
        short  b14:1;
        short  b13:1;
        short  b12:1;
        short  b11:1;
        short  b10:1;
        short  b9:1;
        short  b8:1;
        short  b7:1;
        short  b6:1;
        short  b5:1;
        short  b4:1;
        short  b3:1;
        short  b2:1;
        short  b1:1;
    }b;
}io_port;
```

ビット1をOFFにするとき

```
io_port.port_a= io_port.port_a & 0xfffe
```

```
io_port.b.b1 = 0;
```

ビット2をONにするとき

```
io_port.port_a= io_port.port_a | 0x02
```

```
io_port.b.b2 = 1;
```

# 10. 共用体

NEXT STEP

## (1) 共用体

# 10. 共用体

## (1) 共用体

共用体は構造体に似ていますが、構造体と異なり 1 つのメモリ領域を使います。

したがって、あるフィールドが変更されると、他のフィールドが影響を受ける場合があります。

共用体の使い道としては、あるワードの上位バイトと下位バイトを入れ替えるとか、いくつかのバイトからなるデータを 1 回でクリアしたいとかなどが考えられます。

### 1. 共用体の定義

共用体の定義は構造体とほとんど同じで、「struct」を「union」に変更するだけです。

```
union 共用体名 {
    データ型 メンバー名;
    .
    .
    .
} 変数名;
```

共用体でのデータ管理イメージは次のようになります。

	アドレス	a	b
union test { char a; short int b; }	100		
	101		

構造体の場合

	アドレス	a	b
struct test { char a; short int b; }	100		
	101		
	102		

# 10. 共用体

## 2. 共用体の宣言

**union 共用体名 変数名;**

## 3. 共用体の初期化

宣言と同時に値を代入（初期化）するときは、構造体とは異なり最初の内部変数にのみ可能です。

## 4. 共用体変数のメンバーへのアクセス

構造体と同じように "."（ドット演算子）を使用します

**共用体変数名.メンバー変数名;**

配列の場合

**共用体変数名[添え字].メンバー変数名;**

共用体のポインタからメンバーへのアクセス

"->"（アロー演算子）を使用します。

**共用体へのポインタ名 -> メンバー変数名;**

# 11. 制御

- (1) 判断させる
- (2) 関係演算子
- (3) 論理演算子・否定演算子
- (4) 制御構文
- (5) ネスト(入れ子)構造
- (6) インデント
- (7) テスト

# 11. 制御

## (1) 判断させる

コンピュータに「判断」させるためには条件を書く必要があります。

- ある変数が0以上か?
- ある変数は1か?
- 年齢が20以上30未満か?
- ボタンが押されているか?

「判断」した結果「何を実行するか」

- もし年齢が20以上30未満だったら……(if)
- ボタンが押されている間繰り返す……

## (2) 関係演算子

演算子	意味	記述例	使用例の意味
==	等しい	a == b	bがaに等しい
!=	等しくない	a != b	bがaに等しくない
>	より大きい	a > b	bよりaが大きい
>=	以上	a >= b	bよりaが大きい等しい
<	より小さい	a < b	bよりaが小さい
<=	以下	a <= b	bよりaが小さい等しい



# 11. 制御

関係演算子は、真(true)もしくは偽(false)を計算結果として返します。

C言語の場合、真偽値はint型の整数で

偽(false)は「0」

真(true)は「0以外」

## (3) 論理演算子・否定演算子

演算子	意味	記述例	使用例の意味
&&	論理積	a && b	a かつ b
	論理和	a    b	a もしくは b
!	論理否定	!a	a ではない

関係演算子は、真(true)もしくは偽(false)を計算結果として返します。

使用例;

使用例	意味
1 <= a && a <= 30	1以上、30以下
a < 30    300 <= a	aが30より小さいか、300以上のとき
!(1 <= a)	aが1以上 でない とき (a < 1 と同じ)

# 11. 制御

浮動小数点の比較は誤差を考慮します。

- 浮動小数点はごさ誤差を含みます  
「==」や「!=」などでの比較はバグの温床になります  
ちょうど意図したとおりの値になることはほとんどあり得ません
- 比較したい場合は、許容誤差を考慮して比較する  
-0.001 <= real && real <= 0.001

```
float real = 3.3;
real -= 1.1
real -= 1.1
real -= 1.1
char a;
if(real == 0.0) {
    .....
}
```

## (4) 制御構文

- |                    |                    |
|--------------------|--------------------|
| • if文              | • for文             |
| • ifとelse          | • for文とwhile文      |
| • else if文         | • for文のパターン        |
| • switch文          | • do-while文        |
| • switch文とbreak文   | • while文とdo-while文 |
| • if文とswitch文の使い分け | • break文           |
| • while文           | • continue文        |

# 11. 制御

## if文

- コンピュータに判断させて、振る舞いを変えるために使います  
例1. もしボタンがクリックされたら、ウィンドウを閉じる  
例2. 自動販売機の返却ボタンが押されたら、お金を返却する
- 条件式を満たしたとき、次の文(ブロック)を実行する
- 条件式を満たさないときは何も実行しない

```
if(条件式)
{
    実行する文;
    .....
}
```

```
if(a<0)
{
    a *= -1;
}
```

## ifとelse

- 条件式を満たしたとき、次の文(ブロック)を実行する
- 条件式を満たさないときはelse以下の文(ブロック)を実行する

```
if(条件式)
{
    満たすとき実行する;
    .....
}
else
{
    満たさないとき実行;
    .....
}
```

```
/* bを a で除算 */
if(a == 0)
{
    printf( "divided by 0¥n" );
}
else
{
    b /= a;
}
```

# 11. 制御

## else if文

```
if(条件式1)
{
    条件式1が真のとき実行す;
    .....
}
else if(条件式2)
{
    条件式1が偽で、条件式2が真のとき実行す;
    .....
}
else if(条件式3)
{
    条件式1、2が偽で、条件式3が真のとき実行す;
    .....
}
else
{
    条件式1、2、3が偽実行のとき実行;
    .....
}
```

## 分岐の影響範囲を明示する

- ブロック内が1文の時は中括弧を省略できる  
ただしバグの温床となるため省略しない方が望ましい
- 中括弧 { } で複数の文を囲むと1つの文とみなされる、これを複文と呼ぶ

```
if(a < 0)
{
    a *= -1;
}
```

```
if(a < 0)
    a *= -1;
```

# 11. 制御

## switch文(1)

- (単純な)場合分けをするときに使います  
変数の中身に応じて
  - 1: "one"を表示
  - 2: "two"を表示
  - ...
  - 9: "nine"を表示

```
switch(a)
{
case 1:
    printf( "one" );
    break;
case 2:
    printf( "two" );
    break;
...
case 9:
    printf( "nine" );
    break;
```

## switch文(2)

```
switch(式)
{
case 定数式1:
    文1;
    文2;
    ...;
case 定数式2:
    ...
default:
    文n;...
}
```

この式の値に基づき動作

式の値と一致する  
case 定数式: にジャンプ

定数式には変数を使えない  
×  $a*3$   
○  $5*3$

どの定数式にもあてはまらない場合は  
default以下を実行

# 11. 制御

## switch文とbreak文(1)

- break文があると制御はswitch文から抜けます

```
switch(a)
{
case 1:
    printf( "one" );
    break;
case 2:
    printf( "two" );
    break;
...
case 9:
    printf( "nine" );
    break;
```



```
switch(a)
{
case 1:
    printf( "one" );
case 2:
    printf( "two" );
    break;
...
case 9:
    printf( "nine" );
    break;
```

break文の書き忘れ!

↓  
思わぬところまで実行される

※ caseとbreakは  
セットにして使用する

## switch文とbreak文(2)

- switch文の中でのbreakは省略しない  
バグの温床になる
- 意図的に省略しているときはコメントとを入れる  
/\* 下に続くなど \*/  
ただし複数条件を書いているときなど、  
入れない方がいい場合もある

```
switch(a)
{
case 2:
    /* 2月の処理 */
    break;
case 4:
case 6:
case 9:
case 11:
    /* 30日の月の処理 */
    break;
default:
    /* 31日の月の処理 */
}
```

すべてにコメントが  
入ったら、可読性を損  
なう

# 11. 制御

## if文とswitch文の使い分け

- switch文は整数型を定数で場合分けするときに使います  
シンプルにわかりやすく記述できますが  
条件に変数を使えないという制限があります
- switch文が使えないときはif文で記述します  
と手に加速文で記述できる文は  
if, else if, elseの組み合わせでswitch文でも記述できますが  
煩雑になります

```
switch(a)
{
case 1:
    printf( "one" );
    break;
case 2:
    printf( "two" );
    break;
...
case 9:
    printf( "nine" );
    break;
```

```
if(a==1)
{
    printf( "one" );
}
else if(a==2)
{
    printf( "two" );
}
...
else if(a==9)
{
    printf( "nine" );
}
```

## elseやdefaultを省略しない

```
if(a < 0)
{
    a *= -1;
}
else
{
    /* 清野数の時は無視 */
}
```

考えていない、抜けや漏れでないことを、コメントを残すことで明示します

書いていないと、バグを見つけようとするときに、もう一度考えなければいけなくなります

# 11. 制御

## while文

- 条件式が真の間繰り返します

```
while(条件式)
{
    文
    ...
}
```

1. 条件式を計算

2. 条件式が真(0以外)であれば命令文を実行

3. 命令文を最後まで実行したら1.に戻る

## for文

1. 初期化式を実行

2. 条件式を計算

```
for(初期化式;条件式;更新式)
{
    文;
    ...
}
```

3. 条件式が真(0以外)であれば命令文を実行

4. 命令文を実行し終わったら更新式を実行して2.に戻る

while文: 使用例  
1~10までを加算するプログラム

```
int sum=0;
int count;
count=1;
while(count <= 10)
{
    sum += count;
    count++;
}

printf( "%d\n" , sum);
```

for文: 使用例  
1~10までを加算するプログラム

```
int sum=0;
int count;
count=1;
for(count=1; count <= 10; count++)
{
    sum += count;
}

printf( "%d\n" , sum);
```



# 11. 制御

## for文のパターン

- nからmまで変数の付帯を更新しながら繰り返す

```
int I;  
for (i=n; i<=m; i++)  
{  
    処理 ;  
}
```

- 無限ループ

```
for (;;)   
{  
    処理;  
}
```

## do-while文

- 条件式が真の間繰り返す(ただし最初の1回は無条件に実行)

```
do  
{  
    文;  
    ...  
}  
while(条件式)
```

1. 命令文を実行

2. 命令文を実行し終わったら条件式を計算

3. 条件式が真であれば 1. に戻る

# 11. 制御

## while文とdo-while文

- 条件判断のタイミングに注意

```
while(条件式)
{
    文 ;
    ...
}
```

条件式によっては1回も実行されない場合もある

```
do
{
    文 ;
    ...
}
while(条件式);
```

条件式に寄らず必ず1回は実行される

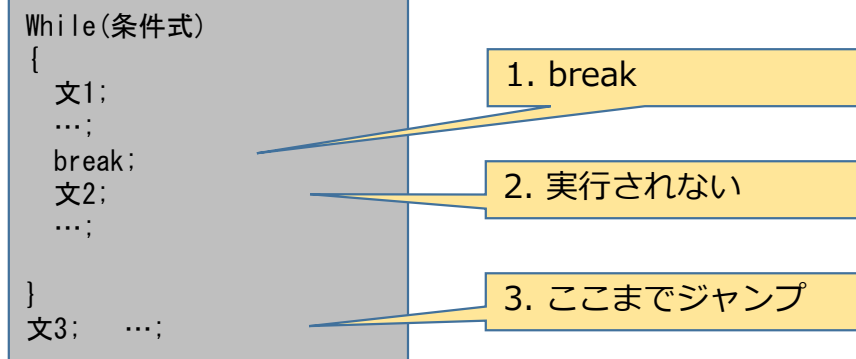
## 適切なループを選択する

- 3種類のループはそれぞれの典型的な使い方に即して使う
  - for: 繰り返し回数が決まっている場合
  - while: 条件が成り立っている間、繰り返す場合
  - do-while: ループの最後で条件判断をする場合
- コードを読んだときに、糸を瞬時に把握できるようにする
  - 繰り返し回数が決まっているのに、whileだと混乱する
  - 逆に繰り返し回数が決まっていないのに、forを使うと同様に混乱する

# 11. 制御

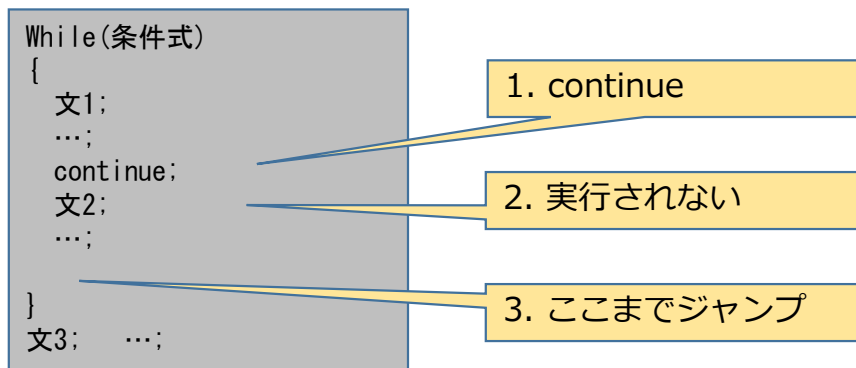
## break文

- while / do-while / for文の中でbreak文が呼ばれるとif文以外の制御構文を終了して次の処理に移る



## continue文

- while / do-while / for文の中でcontinue文が呼ばれるとcontinue文を含む一番内側の残りの処理をスキップする



# 11. 制御

## (5) ネスト(入れ子)構造

- 制御構造は入れ子にすることができる

例1 : while(for)-break

```
while(式) {  
    文1;  
    文2;  
  
    if(式) {  
        break;  
    }  
}
```

例2 : 多重ループ

```
for(int i=1; i<=n ; i++) {  
    for(int j=5 ; j<=m ; j++) {  
        iとjを使用した処理  
    }  
}
```

## (6) インデント

- ソースコード内では、「タブ」を使うと制御構造がわかりやすくなります
- エディタが自動的に整形してくれることが多い

```
while(式) {  
    文1;  
    文2;  
  
    if(式) {  
        break;  
    }  
}
```



```
while(式) {  
    文1;  
    文2;  
  
    if(式) {  
        break;  
    }  
}
```

制御構造がわからない

# 11. 制御

## (7) テスト

- テスト：ソフトウェアのバグを発見する  
テストを適切に実施することで、容易にバグを発見できる
- 格言「石橋をたたいて渡る」  
正しいと思い込んでいるプログラムの途中の状態を確認する  
printf  
処理の途中の変数をprintfで表示して、正しく動作しているかを確認する  
(組み込みプログラムの場合はprintfの代わりにシリアル通信を使う場合が多い)  
特に複雑に変化するループ内の変数は、表示させてみる価値がある
- assert  
途中の変数が意図通りの性質を満たしているかどうか確認する

```
#include<assert.h>  
...
```

assertを使うためにヘッダー  
ファイルをインクルード

```
assert(I == 10);  
assert(sum >= 50);
```

カッコ内に記述した条件を満た  
していないときエラーとなる

## 12. 関数

- (1) C言語の構造
- (2) 関数
- (3) 関数の定義と構造
- (4) 関数の呼び出しと引数
- (5) 値渡しとポインタ渡し
- (6) main()関数
- (7) 関数のプロトタイプ宣言

# 12. 関数

## (1) C言語と構造

C言語のプログラムの構成は、基本的に次のようになっています。

```
#include <stdio.h> ← 入出力に関するヘッダーファイルの取り込み
int main(void) { ← 関数の開始 (int型のmain関数)

    return (0); ← 関数の結果を戻す (「0」の場合、通常の終了)
} ← 関数の終わり
```

## (2) 関数

関数は処理をまとめたプログラムの部品です。

C言語プログラムは関数を使わなくても作成できますが、とても冗長で可読性の悪いプログラムになります。  
2回以上同じ処理をするプログラムや、1回しか実行しないが処理に分かりやすい名前を付けたい場合、関数にして使用します。

関数は呼び出し元から引数としてデータを受け取り、これを基に計算や処理を行い結果を呼び出し元に返すことができます。  
関数内では引数もしくは関数内で定義した変数(Global変数は別)のみ使用することができます。

C言語には標準関数として良く使う処理が準備されています。

# 12. 関数

## (3) 関数の定義と構造

```
型名 関数名(型名 引数1, 型名 引数2, . . . , 型名 引数n) {  
    処理  
    return オブジェクト  
}
```

変数と同じように関数にも型の指定が必要です。関数の型名は return オブジェクト の**オブジェクト(戻り値)の型**となります。

また戻り値なしの場合は

```
void 関数名(型名 引数1, 型名 引数2, . . . ) {  
    処理  
}
```

または

```
void 関数名(型名 引数1, 型名 引数2, . . . ) {  
    処理  
    return オブジェクトなし  
}
```

となります。

以下のように処理の途中で return句 を記述するとそれ以降の処理は実行されません。

```
型名 関数名(型名 引数1, 型名 引数2, . . . , 型名 引数n) {  
    処理1  
    return オブジェクト  
        処理2          /* 実行されない */  
}
```



## 12. 関数

### (4) 関数の呼び出しと引数

関数は以下のようにして呼び出して、処理を実行します。

- ・関数を呼び出す方法 - 引数なし&戻り値なし

関数名();

- ・関数を呼び出す方法 - 引数あり&戻り値なし

関数名(型名 引数1, 型名 引数2, . . . , 型名 引数n);

- ・関数を呼び出す方法 - 引数なし&戻り値あり

変数 = 関数名();

- ・関数を呼び出す方法 - 引数あり&戻り値あり

変数 = 関数名(型名 引数1, 型名 引数2, . . . , 型名 引数n);

#### 仮引数と実引数

まず、引数（引き渡された数）は、関数につけられた括弧内にある数値あるいは変数のことです。

たとえば  $f(a, b)$  の  $a$  と  $b$  が引数です。

**仮引数とは関数定義時に使用される引数のことです。**

```
int f(int x, int y)
```

```
{  
    return x+y;
```

```
}
```

$x, y$  は**仮引数**です。

## 12. 関数

仮引数に対して、関数を実際に使用するときに関数に引き渡される引数のこと実引数といいます。

```
main()
{
    int a = 5, b = 10;
    a = f(a, b);
}
a,bは実引数です。
```

C言語では、実引数に関数に引き渡されるとき、その実引数自身ではなく、実引数のコピーが引き渡されます。すなわち、関数の側から見ると、引き渡されたのは関数に渡された引数そのものではなく、その引数の値のコピーだけです。たがって、関数からその実引数の値を変更することはできません。

次のような関数で関数に引き渡した実引数 a の値は変わりません。

```
f(int x)
{
    x = 10;
}

main()
{
    int a = 5;
    f(a);
    printf("%d¥n", a);
}
```

この場合表示される a の値は **5** となります。

# 12. 関数

仮引数と実引数のメモリの内容の推移

前ページのプログラム

```
f(int x)      ②
{
    x = 10;    ③
}

main()
{
    int a = 5;  ①
    f(a);      ②
    printf("%d\n", a); ④
}
```

この場合のメモリは以下ようになります。

①

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002		
0000	5	A

②

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002	5	X
0000	5	A

③

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002	10	X
0000	5	A

④

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002	破棄	
0000	5	A

コピー

コピー

# 12. 関数

## (5) 値渡しとポインタ渡し

関数の引数の渡し方には値を直接渡す値渡しと、変数のポインタを渡すポインタ渡しがあります。

値渡しは、コピーした引数の値をコピーして渡します、それに対してポインタ渡しは変数のアドレスを渡します。  
値渡しの場合メモリの内容は全ページの説明のようになりますが、ポインタ渡しは次のようになります。

```
f(int *x)           ②
{
    *x = 10;         ③
}

main()
{
    int a = 5;        ①
    f(&a);             ②
    printf("%d¥n", a); ④
}
```

結果は **10** になります。

サンプルコード

<https://paiza.io/projects/shrrGQ9BOxgEdVs6oZNi8w>

ポインタ渡しの場合  
関数fに渡されるのは  
aの値ではなく**アドレス**です

関数fではポインタ変数\*x  
の**内容をアドレスとする**メモリの  
内容を変更しますです

①

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002		
0000	5	a

アドレス

②

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002	0000	*x
0000	5	a

③

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002	0000	*x
0000	<b>10</b>	a

④

番地	メモリ	変数名
000A		
0008		
0006		
0004		
0002	破棄	
0000	10	a

# 12. 関数

## 値渡しとポインタ渡しの違い

値渡し：変数の中身を書き換えても呼び出し元には影響しない

ポインタ渡し：変数の中身を書き換えると呼び出し元の変数の中身も書き変わる

## ポインタ渡しを使うと便利な場合

- ・ 呼び出し元の変数の値を書き換えたいとき
- ・ 複数の返り値が必要な時

通常関数は1つの返値しか呼び出し元に返せませんが、ポインタ変数を使うと複数の返値に対応できます。

- ・ 巨大な配列データを関数に渡したいとき

値渡しの場合は関数にデータを渡す時に呼び出し元の変数をコピーして渡すので、そのデータの分のメモリを消費します。

これが巨大な配列データだった場合はメモリを大量消費してしまいます、またコピーに時間もかかってしまいます。

この配列データをポインタで渡すと、消費メモリは配列の先頭アドレス分のみで、またコピーもしませんので実行速度にも大きな影響がありません

char a[0x10000] を渡す場合

値渡し

番地	10001 メモリ	変数名
10004		
10003		
10002		
10001		
10000		
0FFFF	0	a[0xffff]
00001	0	a[1]
00000	0	a[0]

番地	10001 メモリ	変数名
1FFFF	0	a[0xffff]
10001	0	a[1]
10000	0	a[0]
0FFFF	0	a[0xffff]
00001	0	a[1]
00000	0	a[0]

配列a全体が  
コピーされる

番地	10001 メモリ	変数名
10000	0	a*[]
0FFFF	0	a[0xffff]
00001	0	a[1]
00000	0	a[0]

配列aの先頭ア  
ドレスのみ

# 12. 関数

## (6) main()関数

C言語において main()関数は 一番最初に実行されます。

例えば main()関数の前に 別の関数を書いても、最初に実行されるのは main()関数です。

```
#include <stdio.h>
f(int x)
{
    x = 10;
}

main()
{
    int a = 5;
    f(a);
    printf("%d\n", a);
}
```

コマンドライン引数をもつプログラムのmain()関数

```
int main(int argc, char *argv[]){
```

このように書かれたmain()関数を見かけたことがあると思いますが、  
これがコマンドライン引数に対応したmain()関数の書き方となります。

```
int argc      ← コマンドライン引数の数
char *argv[] ← コマンドライン引数
```

## 12. 関数

コマンドラインで

% ./test 100 abc と入力してプログラム test を実行した場合

argc=3; コマンドライン引数の数

argv[0]="./test";

argv[1]="100";

argv[2]="abc";

となります。

これらの変数は main()関数内で参照できます。

argv[0]には**プログラム名が格納**、またコマンドライン引数は**常に文字列**としてプログラムに取り込まれることに注意して下さい。

サンプルコード

<https://paiza.io/projects/ZH8fGKZxPA9qMwPhbaOuKw>

(paiza.ioでコマンドライン引数を送る方法が不明の為、piazza.ioでは正常な結果を得られません)

# 12. 関数

## (7) 関数のプロトタイプ宣言

C言語で関数を使う場合は、呼び出す関数は使用する前に記述しておく必要があります。

間違い例:

```
#include <stdio.h>
void main(void)
{
    int a = 5;
    f(a);
    printf("%d\n", a);
}
```

main()関数から呼ばれている、  
f()関数が main() の後に  
記述されています。

```
void f(int x)
{
    x = 10;
}
```

正しい例:

```
#include <stdio.h>
void f(int x)
{
    x = 10;
}

void main(void)
{
    int a = 5;
    f(a);
    printf("%d\n", a);
}
```

main()関数から呼ばれている、  
f()関数が main() の前に  
記述されています。



## 12. 関数

関数の数が少ない場合は良いのですが、関数が多くなってくるとその管理も大変になります。  
C言語では関数の処理を記述せずに、関数の宣言だけを行う「プロトタイプ宣言」が用意されています。

このプロトタイプ宣言で関数を宣言すれば、関数の記述の順番に関係なくいつでも関数を利用できるようになります。

プロトタイプ宣言例:

```
#include <stdio.h>
void f(int);
main()
{
    int a = 5;
    f(a);
    printf("%d¥n", a);
}
```

```
void f(int x)
{
    x = 10;
}
```

プロトタイプ宣言の記述は

**戻り値の型 関数名(引数の型);**

のように書きます。

main()関数から呼ばれている、  
f()関数が main() の後に  
記述されていますがmain()関数  
の前にプロトタイプ宣言されて  
います。

## 12. 関数

プロトタイプ宣言の記述は  
**戻り値の型 関数名(引数の型);**  
 のように書きます。

```
void f(int x)
{
    x = 10;
}
```

この関数のプロトタイプ宣言は  
 void f(int);  
 となります。

引数を持たない関数

```
void f(void)
{
    int x;
    x = 10;
}
```

このプロトタイプ宣言は  
 void f(void);  
 のように、引数名のところに **void** と記述します。

```
void a(short int b){
    short int z;
    z = b;
    printf("func a : z=%d¥n",z);
}
```

この関数のプロトタイプ宣言を

- ① void a(short int);      仮引数名の記述無し
  - ② void a(short int b);    実関数の仮引数名と同じ
  - ③ void a(short int y);    実関数の仮引数名と異なっている
- のように書いても3つともエラーとはなりませんが、  
 引数の部分は ① のように引数の型のみを記述するようにしましょう。

サンプルコード

<https://paiza.io/projects/B1RWuf4oDdM0FjpUcnRQwQ>

# Webで見つけた良くある間違い

## 間接参照演算子ってややこしそうな名称

略すよりマシ。(をい  
ポインタ型の変数は、どこかの変数を指し示しています。  
じゃあ、その指し示している変数の値は？ 値を書き換える  
には？

`int *fuga = &hoge;` として宣言したint型ポインタ型変数  
が指し示しているのは、int型変数のhoge。

じゃあ、hogeの値を参照すればいいし、hogeに値を代入  
したら、書き換わるね♪～

、、いやそうなんだけど、それならfugaと言う変数さ  
んの立場が無いので。

間接参照演算子「\*」を、ポインタ型の変数の前に付け  
ると、あら不思議。ポインタ型が指し示す変数の値にアクセ  
ス出来ちゃう！

と、ポインタ型変数を経由して、指し示されている変数に  
対して間接的にアクセスしちゃう演算子なので間接参照演  
算子。

```
int fuga = 127; /* int型の変数 値は127 */  
int *ihen;      /* int型へのポインタ型 値未初期化 */  
ihen = *fuga;   /* ihenは、fugaを指し示せ！！ */
```

\*ihen とすると、fugaと同じように扱えます。

\*ihen = 0; /\* fugaが0になってもたー \*/ とか。

とある Web書籍から

単純なミスだと思いますが「&」と書くべきなのに  
「\*」と書いてしまっています。

`ihen = *fuga; /* ihenは、fugaを指し示せ！！ */`  
↓

`ihen = &fuga; /* ihenは、fugaを指し示せ！！ */`

# 付録

ASCIIコード表

上位3ビット→		0	1	2	3	4	5	6	7
下位4ビット↓	0	NUL	DLE	(SP)	0	@	P	'	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAC	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

制御コード表

2進	16進	略語	CS	エスケープシーケンス	名前/意味
000 0000	0	NUL	^@	¥0	ヌル文字
000 0001	1	SOH	^A		ヘッディング開始
000 0010	2	STX	^B		テキスト開始
000 0011	3	ETX	^C		テキスト終了
000 0100	4	EOT	^D		伝送終了
000 0101	5	ENQ	^E		問い合わせ
000 0110	6	ACK	^F		肯定応答
000 0111	7	BEL	^G	¥a	ベル
000 1000	8	BS	^H	¥b	後退
000 1001	9	HT	^I	¥t	水平タブ
000 1010	0A	LF	^J	¥n	改行
000 1011	0B	VT	^K	¥v	垂直タブ
000 1100	0C	FF	^L	¥f	書式送り
000 1101	0D	CR	^M	¥r	復帰
000 1110	0E	SO	^N		シフトアウト
000 1111	0F	SI	^O		シフトイン
001 0000	10	DLE	^P		伝送制御拡張
001 0001	11	DC1	^Q		装置制御1、XON
001 0010	12	DC2	^R		装置制御2
001 0011	13	DC3	^S		装置制御3、XOFF
001 0100	14	DC4	^T		装置制御4
001 0101	15	NAK	^U		否定応答
001 0110	16	SYN	^V		同期信号
001 0111	17	ETB	^W		伝送ブロック終結
001 1000	18	CAN	^X		取消
001 1001	19	EM	^Y		媒体終端
001 1010	1A	SUB	^Z		置換
001 1011	1B	ESC	^[	¥e	エスケープ
001 1100	1C	FS	^¥		ファイル分離標識
001 1101	1D	GS	^]		グループ分離標識
001 1110	1E	RS	^^		レコード分離標識
001 1111	1F	US	^_		ユニット分離標識
111 1111	7F	DEL	^?		抹消

# サンプルコード一覧

章	内容	Paiza.io内のURL
2-(3)	変数のサイズ確認	<a href="https://paiza.io/projects/gGmLC_Cb5Br_oelZ5C0wow">https://paiza.io/projects/gGmLC_Cb5Br_oelZ5C0wow</a>
2-(4)	printf()サンプル	<a href="https://paiza.io/projects/vO38qcYpmFGfpu495tx5og">https://paiza.io/projects/vO38qcYpmFGfpu495tx5og</a>
2-(8)	#define Macro サンプル 1	<a href="https://paiza.io/projects/FjVjLp9jUSc802UXOKU_eQ">https://paiza.io/projects/FjVjLp9jUSc802UXOKU_eQ</a>
	#define macro サンプル 2	<a href="https://paiza.io/projects/kQvspcOHmyDWfZ9dUZJe6g">https://paiza.io/projects/kQvspcOHmyDWfZ9dUZJe6g</a>
	列挙型 サンプル1	<a href="https://paiza.io/projects/E6jRobk9ePKe8X_IP8gt1Q">https://paiza.io/projects/E6jRobk9ePKe8X_IP8gt1Q</a>
	列挙型 サンプル2	<a href="https://paiza.io/projects/BQ15iEynVyTMd5s6Fwg9Lg">https://paiza.io/projects/BQ15iEynVyTMd5s6Fwg9Lg</a>
3-(4)	論理シフト算術シフトの例(右シフト)	<a href="https://paiza.io/projects/rY6rESkebOtJRj61aoex4g">https://paiza.io/projects/rY6rESkebOtJRj61aoex4g</a>
	論理シフト算術シフトの例(左シフト)	<a href="https://paiza.io/projects/UXexWBetIpZRvO0i5jpN6w">https://paiza.io/projects/UXexWBetIpZRvO0i5jpN6w</a>
3-(5)	前置、後置インクリメント	<a href="https://paiza.io/projects/gAnBedSUMB3bf_8OLP0YPQ">https://paiza.io/projects/gAnBedSUMB3bf_8OLP0YPQ</a>
3-(6)	sizeof サンプル	<a href="https://paiza.io/projects/woM9-t8-4kXTsf3XBx8iOA">https://paiza.io/projects/woM9-t8-4kXTsf3XBx8iOA</a>
5-(3)	文字列とnull	<a href="https://paiza.io/projects/_HJj23PSBHFfsMqGwM78IAA">https://paiza.io/projects/_HJj23PSBHFfsMqGwM78IAA</a>
6-(5)	2次元配列	<a href="https://paiza.io/projects/87xReNpHSaOfNv0sVGuDZw">https://paiza.io/projects/87xReNpHSaOfNv0sVGuDZw</a>
7-(3)	ポインタ演算子	<a href="https://paiza.io/projects/Q869GzYjFMrG74g0h9l1Ew">https://paiza.io/projects/Q869GzYjFMrG74g0h9l1Ew</a>
12-(5)	ポインタ渡し	<a href="https://paiza.io/projects/shrrGQ9BOxgEdVs6oZNi8w">https://paiza.io/projects/shrrGQ9BOxgEdVs6oZNi8w</a>
12-(6)	コマンドライン引数	<a href="https://paiza.io/projects/ZH8fGKZxPA9qMwPhbaOuKw">https://paiza.io/projects/ZH8fGKZxPA9qMwPhbaOuKw</a>
12-(7)	関数のプロトタイプ宣言	<a href="https://paiza.io/projects/B1RWuf4oDdM0FjpUcnRQwQ">https://paiza.io/projects/B1RWuf4oDdM0FjpUcnRQwQ</a>

# 演習用C言語実行環境

演習用にC言語の実行環境が必要となります。基本コンソールアプリを作成できる環境が必要です。

既にコンソールアプリ用の構築・実行環境がインストールされている場合は、インストールする必要はございません。

## 1. Visual studio 2019

Windows をお使いの場合 **Visual Studio 2019 community**版をお勧めします。

ダウンロードはこちらから <https://visualstudio.microsoft.com/ja/downloads/>

Macをお使いの方も上記ページから **Visual Studio Community 2019 for Mac** をダウンロードできます。

(ただし私がMacを持っていないので動作の確認はしていません)

## 2. Paiza.io

学習する為だけに Visual Studio 2019 をインストールするのに抵抗がお有りの方(結構ハードディスクの容量を必要とします)は

ブラウザ上で C言語のビルド・実行が出来る Paiza.io <https://paiza.io/> をお使いください。

演習問題のプログラムは Paiza.io でも問題なく実行できるように作成します。

※ Paiza.ioのC言語は clang 7.0 / LLVM 7.0 (C17) です。

## 3. Web コンパイラ

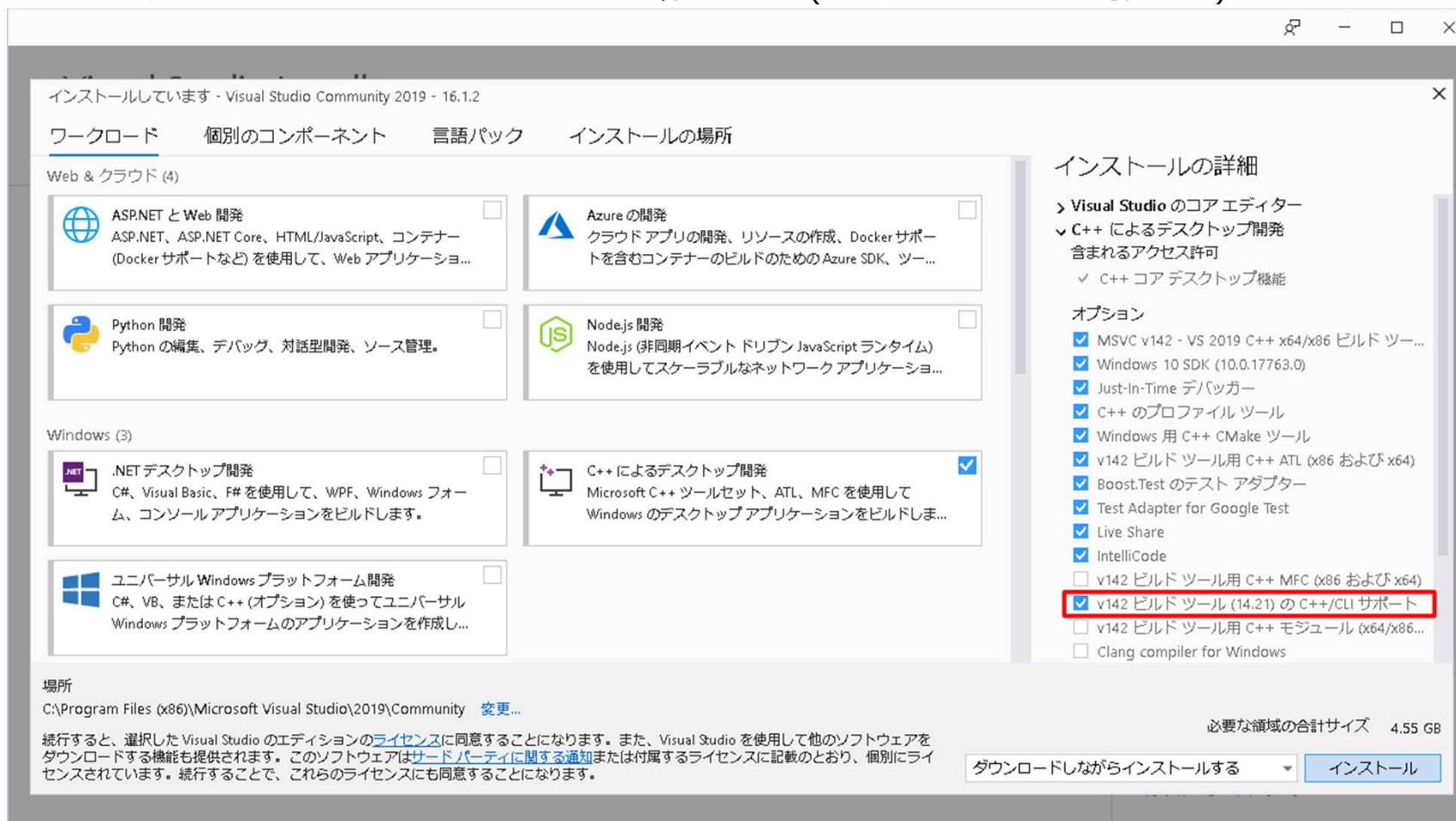
Paoza.ioと同じようにWebブラウザでコンパイルできるサービスが他にも多数公開されています。

実際にためしてはませんが、次のページで紹介されています。

<https://qiita.com/tttamaki/items/2b009aa957cfa4895d50>

## VS 2019インストールの際の注意点

VS 2019 のインストールの際、**CLIサポート** にチェックを入れてください。  
このチェックを入れないとコンソールアプリを作れません(後で追加することは可能です)。





## VS 2019での C言語プログラムの開発

VS 2019 C/C++ はデフォルトで C++プログラム用になっています。

そのままだとC言語プログラムの開発も可能ですが、標準関数のヘッダファイルとかが違っていますので C言語の環境でのプログラム作成をお勧めします。

開発言語をC言語に設定する方法

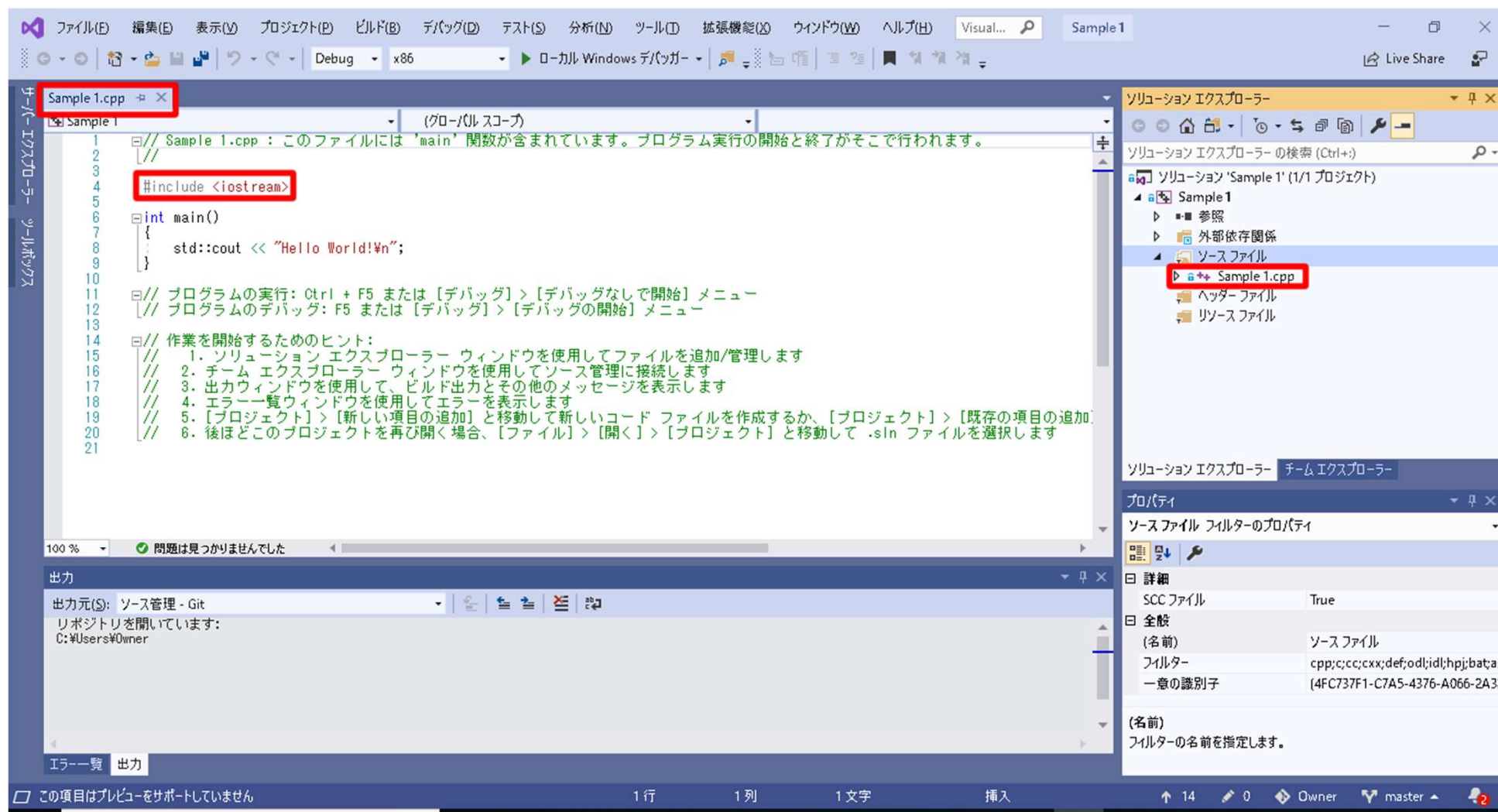
新しいコンソールアプリプロジェクトを作成します。

VS C/C++ の画面が次のようになって ソースファイルの拡張子が .cpp になっています。

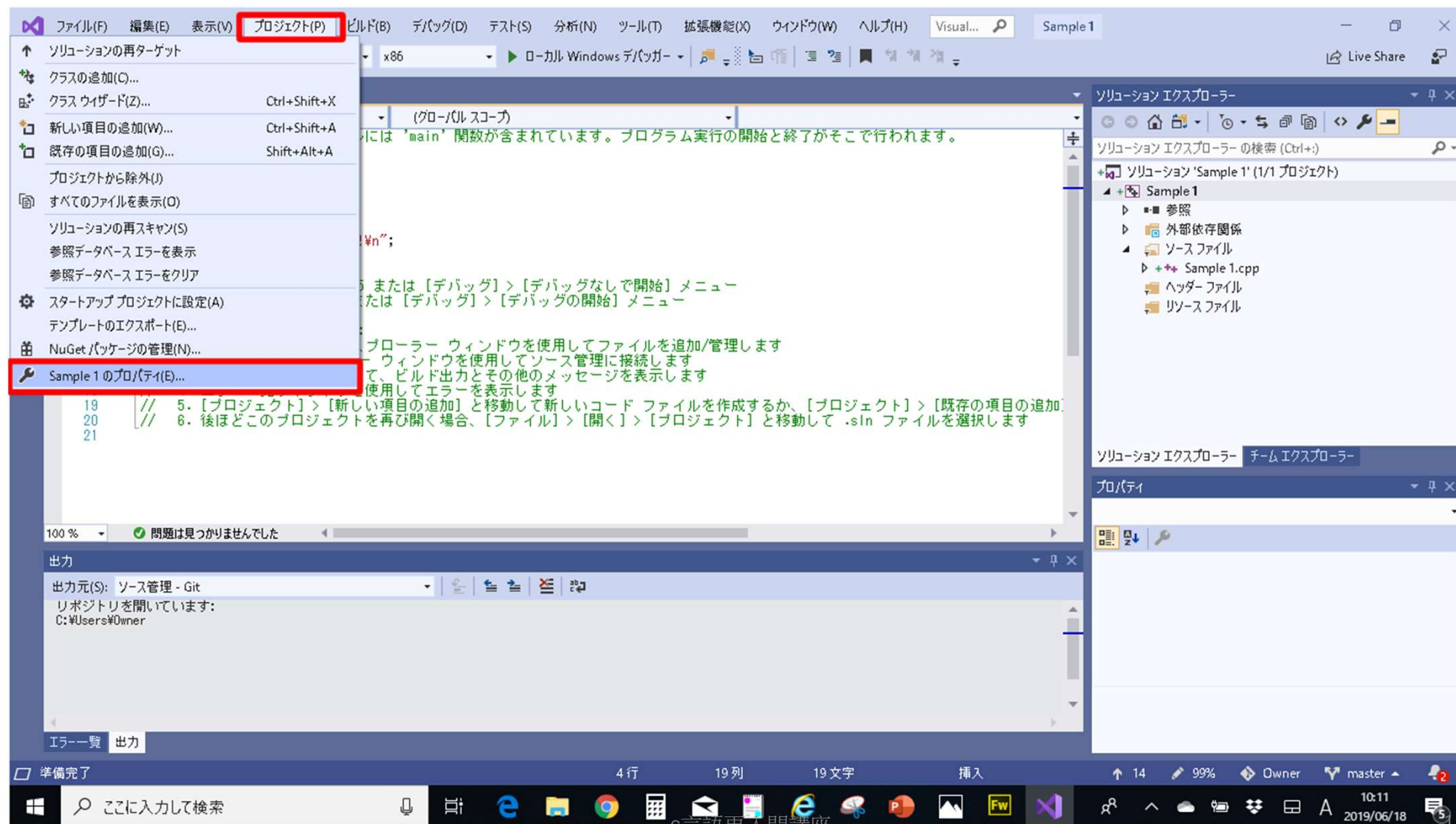
以下のページで C言語 への設定方法を説明します。



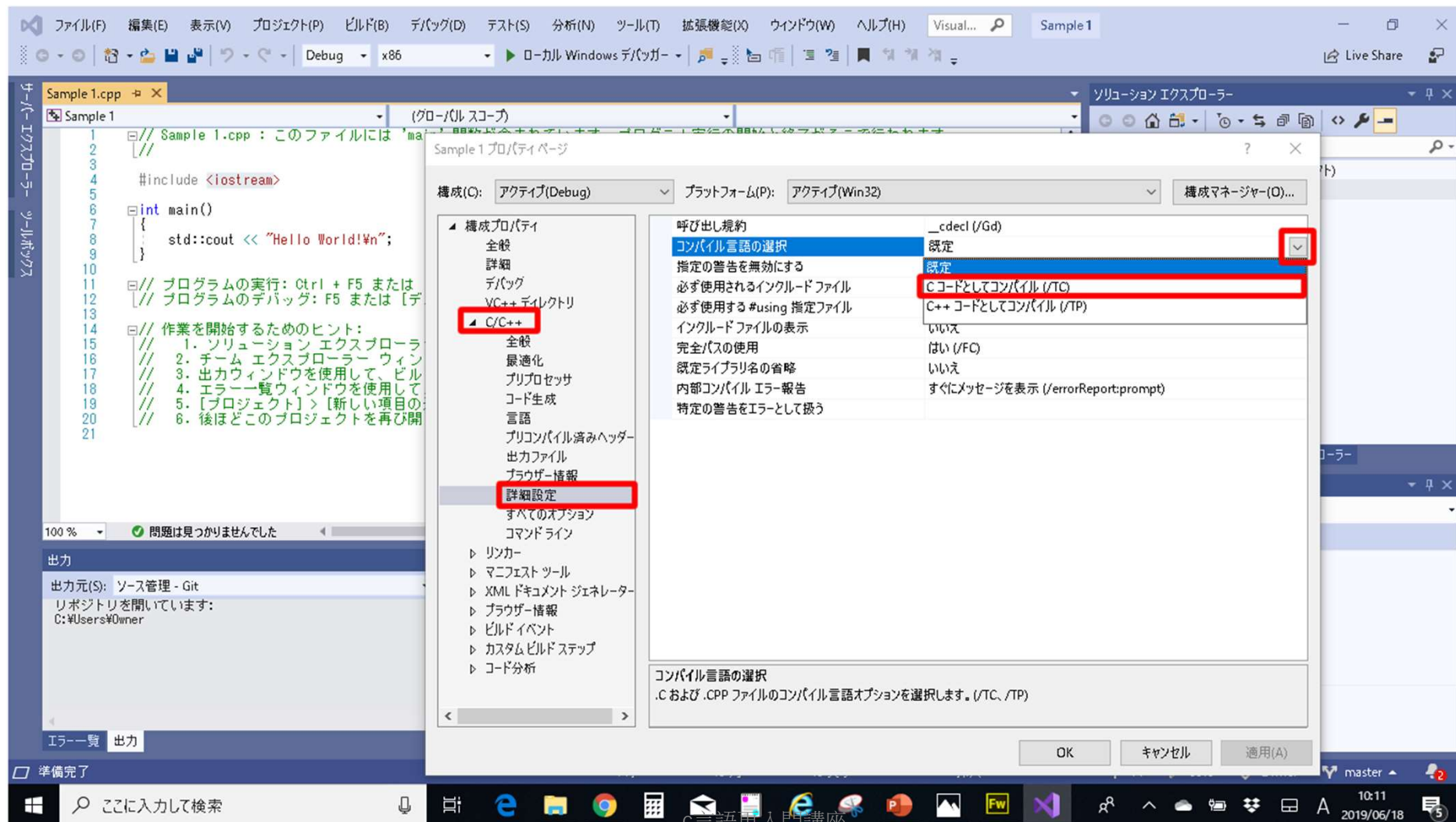
ソースファイル名が \*\*\*\*\*.cpp インクルードされているファイルが <iostream> となっています



[プロジェクト] [\*\*\*\*\*のプロパティ] をクリックします



C/C++の詳細設定ページの コンパイル言語の選択 で Cコードとしてコンパイルを選択してください。



コンパイル言語をC言語に設定してビルドすると大量のエラーが発生します。

The screenshot shows the Visual Studio IDE with a C++ file named 'Sample 1.cpp' open. The code is a simple 'Hello World' program. The 'ビルド' (Build) menu is open, and the 'ビルド + IntelliSense' option is selected. The 'エラー一覧' (Error List) window at the bottom shows a list of compilation errors. The errors are as follows:

コード	説明	プロジェクト	ファイル	行	抑制状態
C2059	構文エラー: ;	Sample 1	cmath	198	
C2449	'/' を見つけました (関数のヘッダーがないかもしれませんが)。	Sample 1	cmath	198	
C2059	構文エラー: ;	Sample 1	cmath	204	
C2061	構文エラー: 識別子 'noexcept'	Sample 1	cmath	210	
C1003	プロシージャ定義の重複: 'noexcept'。このエラーは、このファイルで 'noexcept' が複数回定義されたためです。	Sample 1	cmath	210	

The status bar at the bottom indicates 'ビルド失敗' (Build Failed).



ソースファイル名を \*\*\*\*\*.c に変更して、プログラムもC言語で書き直してビルドするとエラーがなくなります

