2. C言語（2）関数，構造体，配列とポインタ

2.1. 関数

C 言語のプログラムは関数を基本として構成されています。関数にはユーザが作成する関数とコンパイラがライブラリとして提供する関数とがありますが，基本的な使い方は同じです。言い換えると，ユーザが作成する関数もライブラリで提供される関数と同様に使えるように設計する必要があります。

ユーザが関数を定義する場合，その関数がどのような値を必要とし，どのような処理を行って，どのような値を結果として出力するかを考えて設計する必要があります。そして，この考え方はそのまま関数の構造となります。

C 言語の関数構文は以下のようになります。

```
戻値の型 関数名(型 1 引数 1, 型 2 引数 2, 型 3 引数 3, ...){
    ローカル変数の宣言
    処理
    return 文など
}
```

このような形を持つという意味では main 関数も一つの関数であるといえます。また，C 言語の関数は自分自身を呼ぶこともできます。また，main 関数に引数を持たせる場合は，int main(int argc, char *argv[]) (Microsoft 固有の仕様で int main(int argc, char *argv[], char *envp[])というのもあります)となります。

2.1.1. 戻値の型

ユーザ定義関数は戻値の型を一つ定義する必要があります。型が宣言されていない場合，戻値は暗示的に int 型であるとみなされます。また，空の戻値である「void」という型を用いることもできます。戻値は一つの関数に一つしか設定できないため，複数の値を戻したい場合はグローバル変数(あまり好ましくない)や構造体(後述)，ポインタ(後述)を使用します。

関数は他の関数もしくは自分自身から呼び出されるので，関数の型をあらかじめ明らかにしておく必要があります。そこで，関数を呼び出す記憶クラス内(スコープ)に関数の定義をしておきます。これをプロトタイプといいます。一般にプロトタイプ宣言はソースの初めの方で行いますが(図 1.2 参照)，ローカル変数と同様に宣言することもできます。

システムが提供する標準関数もユーザ定義関数と同様に，プロトタイプが必要になります。そこで，標準関数のプロトタイプ(および，それらに必要な変数)はシステムの提供するヘッダファイルの中で宣言されています。読み込むためにはプリプロセッサ命令 include を用いて，

```
#include <ヘッダファイル名>
```

と宣言します(すでに何度か使ってますね)。また，関数の実体はライブラリとして提供されていますので，必要に応じて指定する必要があります(大抵は自動的に読み込まれます)。

2.1.2. 関数名

関数名は英数字で始まり，途中には数字やアンダーバー「_」も使用できます。予約語と呼ばれる特殊な語や標準関数と同じ名前は使用できません。また，C 言語では大文字と小文字は区別されるので注意が必要です。(C 言語全盛のときは，単語の区切りに「_」を使うのが一般的でしたが，Windows 用のプログラムが多くなるに連れて単語の先頭を大文字，他を小文字で書く表記法が一般的になってきました。例，set_data→SetData)。

2.1.3. 引数

関数を呼び出す際に与える引数の型と名前を指定します。複数の引数を指定する時は「,」で区切って並べます。引数がない場合は「void」を指定します。引数が多い場合は，グローバル変数を使用する方法もありますが，構造体を使用するほうが一般的です。

2.1.4. 関数内の処理

最初にローカル変数の宣言を行います。ただし, 引数はそのまま変数として使用できます。関数が戻値を持つときは return 文を使用して, 値を返すようにします。

2.2. プリプロセッサ命令

ソース中の「#」で始まる命令をプリプロセッサ命令と言い, コンパイルに先立って処理されます。C 言語では本来改行は意味を持ちませんが, プリプロセッサ命令は改行が命令の終端になります。複数行にまたがる命令を記述する時には終端に「¥」を書きます。代表的なものを列記すると表 2.1 のようになります。

表 2.1 プリプロセッサ命令

命令	意味	例
#include	他のファイルを読み込む命令です。ヘッダファイルの読込みなどに使用します。	#include 〈stdio.h〉 #include ″sample.c″
#define	マクロを定義するために使用します。	#define PI 3.1415926535 #define Rad2Deg(rad) ((rad) * 180.0 / PI)
typedef	ユーザ定義の型を作ります。先頭に「#」がないのと, 最後に「;」が必要です。	typedef unsigned size_t; typedef struct{double real, image;} comp_t;
#if 〜 #else 〜 #endif	コンパイルに先立って条件分岐を行う場合に使用します。#else #if の代わりに#elif も使えます。	#ifdef _DEBUG #define lprintf(a) printf(a) #else #define lprintf(a) #endif
#ifdef, #ifndef	#if definend, #if not defined の略	

2.3. 配列, 構造体

複数の変数をまとめて管理したい時に使用するのが配列と構造体です。

2.3.1. 配列

同じ方のデータをまとめて扱う時に使用するのが配列です。C 言語では文字列という型がないため, 文字(char)の配列として表されています。配列を宣言する時は,

```
int     idata[10];          int 型のデータ 10 個の 1 次元配列
char    filename[60];       60 文字までの文字列, または char 型のデータ 60 個の 1 次元配列
double  ddata[20][50];      double 型 20×50 個の 2 次元配列
```

また, 初期値を与えて定義することもできます。

```
int     idata[] = {1, 2, 3, 4};
double  ddata[][3] = {{1.0, 2.0, 3.0}, {2.0, 4.0, 6.0}, {3.0, 6.0, 9.0}, {4.0, 8.0, 12.0}};
```

C 言語では実際には 1 次元配列しか使用することができません。そのため, 2 次元以上の配列は既定値を入力する必要があります。後々, これが問題になってきます。

配列の添え字は必ず 0 から始まります。data[n]と宣言したときは data[0]から data[n-1]までの n 個のデータが使用できるということになります。添え字の値を n 以上にしてもコンパイラはエラーを出してくれないので, ユーザ側で注意する必要があります。

2.3.2. 構造体

違う型のデータをまとめて扱いたい時に使用するのが構造体です。関数の戻値は一つしか使用できないので, 複数

の変数を戻したい時など構造体を使用すると便利です。構造体を使用するためには以下のような宣言をする必要があります。

```
struct タグ名{
    型　要素;
      ……
};
```

ソースの中でこの構造体を使用するためには変数定義で,

```
struct タグ名 変数名;
```

と宣言します。宣言が長くなるので不便な時は typedef を使用すると大変便利です。例えば, 動物の名前と足の数の関係をを構造体で定義すると,

```
typedef struct{
    char  kind[20];
    int    legs;
}     animal_t;
```

のように定義すると, ソースの中では,

```
animal_t animal;
```

と, 他の型と同様に使用することができます。構造体に初期値を与えて定義するには,

```
animal_t animal[] = { {"人間", 2}, {"ねこ", 4}, {"たこ", 8} };
```

とすることができます。構造体の要素への参照には「.」を用います。animal[1].kind とすると"ねこ"を参照することができます。また, 構造体がポインタのときは, 「.」の代わりに「->」を用います。例えば, animal_ptr->kind という具合です。

2.4. ポインタ

C 言語では頻繁にポインタが使用されます。ポインタとは変数や関数などのアドレスとその大きさを持つ変数です。このようなポインタを使用する理由は以下の2点です。

1. ポインタを使用したほうが, タの方法よりも簡単で効率的に表現できる。
2. ポインタを使用しないと表現できない場合がある。

2.4.1. ポインタ演算子とアドレス演算子

C 言語の中で宣言された変数はそれぞれアドレスを持っています。図 2.1 では変数 x, y はそれぞれ 100H, 106H というアドレスに保存されています。ポインタはこのような変数のアドレスを保存することができます。宣言は普通の変数を宣言するのと同じように宣言しますが, 違いはポインタ演算子を付けるということです。

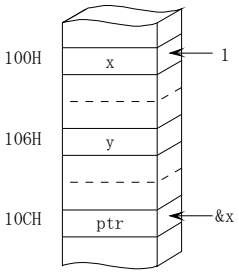


図2.1 ポインタ

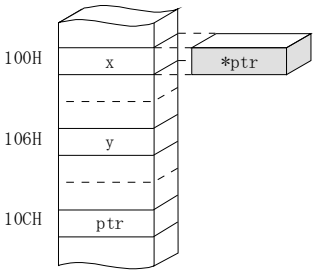


図2.2 エイリアス

```
int    x, y;
int    * ptr;
```

また, ポインタと対で使用されるのがアドレス演算子です, アドレス演算子はある変数のアドレスを取得するのに使用されます。変数に値を代入するのと同様に, ポインタ型変数にアドレスを代入します。

表2.1 ポインタ演算子とアドレス演算子

変数	&変数	*変数
x = 1	&x = 100H	*x = ?
ptr = 100	&ptr = 10cH	*ptr = 1

```
x = 1;
ptr = &x;
```

また, ポインタ変数にポインタ演算子を付けることで, ポインタの指す変数の実体を参照することができます。*ptr と書

くと, ptr が指している･･･すなわち, 100H の･･･int 型の整数を間接的に表すことになります。このように, ptr が x を指す時, *ptr は x のエイリアスといいます。ちょうど図 2.2 のようになります。

2.4.2. 関数の呼び出しとポインタ

```
#include    <stdio.h>

void swap(int x, int y);

int main(void)
{
    int                 a = 5, b = 3;

    swap(a, b);
    printf("A = %d¥n", a);
    printf("B = %d¥n", b);

    return 1;
}
```

```
void swap(int x, int y){
    int                 temp;

    temp = x;
    x = y;
    y = temp;
}
```

実行結果

list2.1 変数を入れ替えるプログラム

2つの変数を入れ替える関数を list2.1 のように作ってみます。実行例から判るように a と b の入れ替えは行われません。C 言語では関数の呼び出しを値による呼び出しで行います。これは,

1. 呼び出し側は実引数として「値」をわたす。
2. 呼び出される側は仮引数として受け取った値の「コピー」を使う。

という決まりによるものです。list2.1 の例では図 2.3 のように main 関数は a, b という変数の実体をわたすのではなく, その値 5, 3 を渡します。呼び出される関数 swap は, その値を x, y の初期値として受け取ります。そのため, swap 関数の中でいくら x と y を入れ替えても main 関数の a, b の値には影響がないことになります。

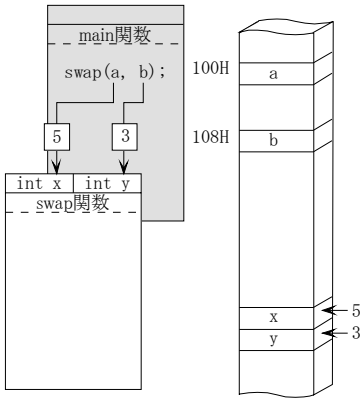


図2.3 関数呼出し1

```
#include    <stdio.h>

void swap(int *x, int *y);

int main(void)
{
    int                 a = 5, b = 3;

    swap(&a, &b);
    printf("A = %d¥n", a);
    printf("B = %d¥n", b);

    return 1;
}
```

```
void swap(int *x, int *y){
    int                 temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

実行結果

list2.2 ポインタを使った入れ替えプログラム

このような場合に使用されるのがポインタです。list2.2 はポインタを使用するように書き換えたソースです。この場合には図 2.4 のように, 呼び出される関数 swap に main 関数の a, b のアドレス 100H, 108H がコピーされ, ポインタ x, y に格納されます。このポインタを使用して, それぞれのエイリアスの値を入れ替えるというものです。

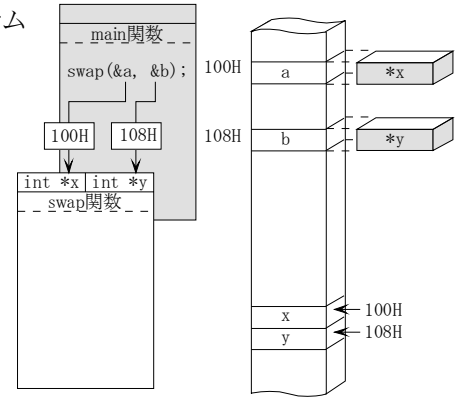


図2.4 関数呼出し2

2.4.3.　ポインタと配列

C¥言語ではポインタと配列は密接な関係があります。引数として配列を受け取るとき，下のように3種類の方法で記述されますが，これらは全て同じものです。

```
void func(int a[10]){      void func(int a[]){        void func(int *a){
      :                          :                          :
}                          }                          }
```

(1) 配列(大きさつき)　(2) 配列(大きさなし)　(3) ポインタ

このように，配列の大きさを指定してもこの数字は無視されます。ただし，この数字は可読性を高めるために役立ちます。また(3)の宣言からもわかるように関数に配列を渡すときはポインタを使用しています。実は C 言語のコンパイラは配列を渡す代わりに(3)の方法を使ってポインタを渡しています。

ポインタと配列の関係は以下のように考えることができます。

```
int a[10];
int *ptr;
ptr = &a[0];
```

表2.2 配列とポインタの関係

配列の要素 a[i]	=(同じ)=	ポインタのエイリアス *(a + i)
配列の要素のアドレス &a[i]	=(同じ)=	ポインタ a + i

を実行すると，ptr に a[0]のアドレスが代入されるので，ptr は a[0]を指すことになるので，*ptr は a[0]のエイリアスになります。一般に ptr+i は ptr の指す要素の i 個後ろの要素を指すので，*(ptr + i)は a[i]のエイリアスになります。C 言語ではこれらの関係について表 2.2 のような規則があります。これをまとめると，図 2.5 のようになります。ここで注意するのは，ptr[i]は 10 個以上あるということです。これは現在 ptr が a を指しているというだけで，上限が決められていないからです。ただし，実際にここに何かを書き込んだときの動作は補償されていません。

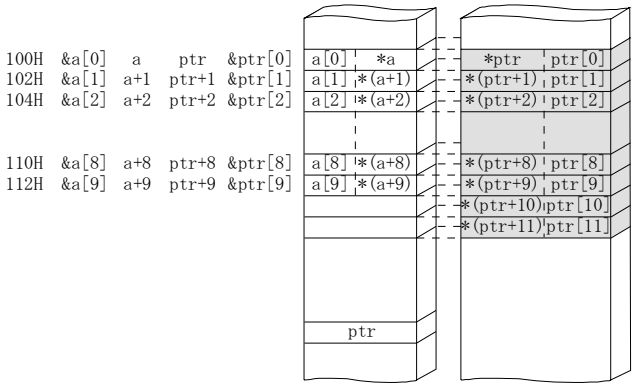


図2.5　配列とポインタ

2.4.4.　文字列とポインタ

文字列とは文字の配列のことです。初期値つき配列の定義を

```
int x[] = {1, 2, 3, 4};
```

とやるのと同じように書くことができます。ただし，C 言語では文字列の最後はNULL 文字('¥0')とするように決まっているので，

```
char c[] = {'a', 'b', 'c', '¥0'};
```

とすることができます。しかし，毎回このように書くのは不便ですから，

```
char str1[] = "abc";
```

という書き方が許されています。また，

```
cahr *str2 = "def";
```

という定義法もあります。

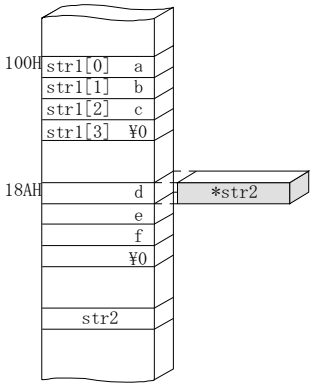


図2.6　文字列

以上のことを踏まえた上で，文字列をコピーする関数を作成してみます。list2.3 が基本になります。これをもっとポインタらしく書き直したのが list2.4 です。関数の流れを図 2.7 に示します。こちらの場合，余分な変数 i を省いてあるぶん変数の処理が減り，速度の向上が望めます。さらに，ポインタら

```
void strcpy(char *string1, char *string2){
    int                 i = 0;

    while(string2[i] != NULL){
        string1[i] = string2[i];
        i++;
    }
```

list2.3 文字列のコピー1

しく書き直すとlist2.5になります。

```
void strcpy(char *string1, char *string2){
    while((*string2 = *stinrg1) != NULL){
        string1++;
        stinrg2++;
    }
}
```

list2.4 文字列のコピー2

```
void strcpy(char *string1, char *string2){
    while((*string2++ = *stinrg1++) != NULL)
        ;
```

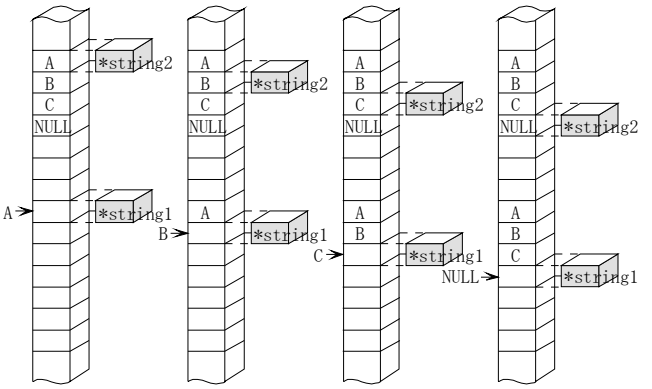


図2.7 文字列のコピー

```
void strcpy(char *string1, const char *string2){
    char               *ptr = string2;

    while((*string2++ = *string1++) != NULL)
        ;
    return ptr;
```

list2.6 strcpyらしく

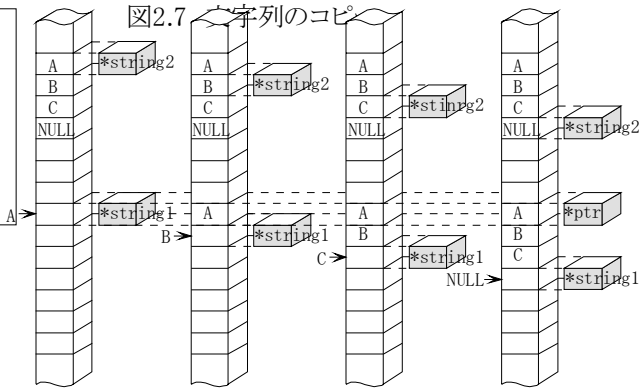


図2.8 strcpyらしく

実際に標準ライブラリで定義されるstrcpy関数のプロトタイプは,

char *strcpy(char *string1, const char *string2);

となっています(ここでconst char *string2はcharへのポインタがconstant, つまり変化しないことを意味しています)。これにあわせてlist2.5を書き替えるとlist2.6, 図2.8のようになります。

※おかしやすいミス

文字列処理でおかしやすいミスを list2.7 に示します。ポインタと文字列が図2.9 a)のようになっています。strcat関数はb)のように文字列の最後のNULLのところから1文字ずつコピーをします。最終的には c)のようになるわけですが, 追加分の部分が空いている補償はありません。この領域にデータが保存されている可能性があります。このような場合にはlist2.8や図2.9 d)のように予め文字列用のメモリを確保しておく必要があります。

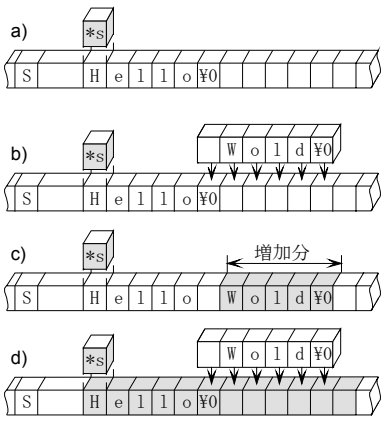


図2.9 文字列の追加

2.4.5. ポインタと多次元配列

C言語では, 厳密な意味での多次元配列は存在せず, 1次元配列のみが存在します。そのため多次元配列を使用する際は, それらの特徴を知った上で使用する必要があります。

1次元配列と同じように, 多次元配列を受け取る関数の定義をまとめると以下のようになります。

```
void func(int a[4][3]){      void func(int a[][3]){      void func(int (*a)[3]){
    :                            :                           :
}                            }                           }
```

(1) 配列(大きさつき)　(2) 配列(大きさなし)　(3) ポインタ

すべての場合で, aは「int型の大きさ3の配列」へのポインタを示しています。そのため, 「3」は省略できません。

例えば。図2.10に示した配列を確保する場合, int c[4];と定義するのと同様にint x[4][3];と定義します。ここで, 前者は「int型の変数」を1つの要素とする大きさ4の配列cとなり, 後者は「int型の大きさ3の配列」を1つの要素とする大きさ4の配列xとなります。

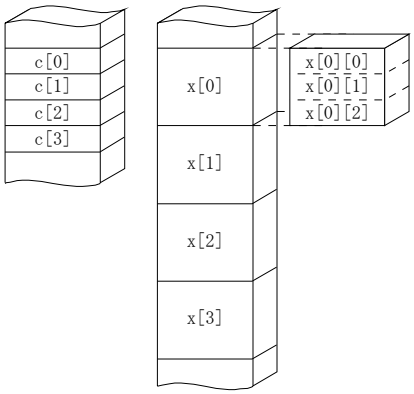


図2.10　多次元配列

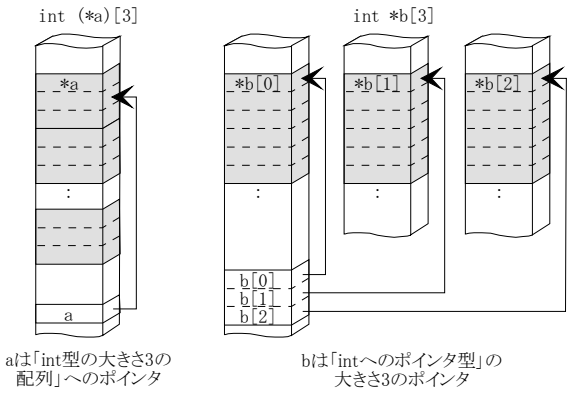


図2.11　int (*a)[3] と int *b[3] の違い

1 次元配列のときは配列名 c だけだと int へのポインタを示しました。2 次元配列の時は配列名 x は int 型の大きさ 3 の配列へのポインタになります。int (*a)[3]と int *b[3]の違いは図 2.11 のようになります。

【練習問題 2.1】 配列

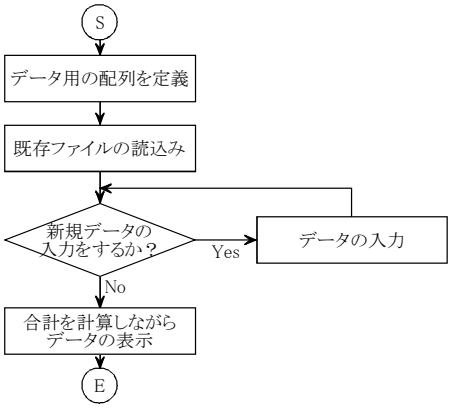
振幅 A，周期 T，波長 λ の x 軸上を進む正弦波の時間 t，位置 x における変位は $y = A \cdot \sin(2\pi(t/T - x/\lambda))$ で表される。$A = 50$，$T = 4$，$\lambda = 10$，$x = 0,2,4,\cdots,20$ の 11 個の点に置かれた上述の調和振動子の任意の時刻（t を適当に与える）における変位をそれぞれ配列に格納し，表示しなさい。

【練習問題 2.2】 ポインタと配列

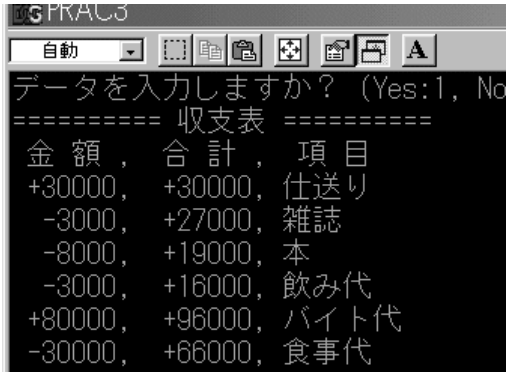
練習問題 1.7 で作成したデータ t, x, y を配列に読込み，一覧に表示するプログラムを作成せよ。

【練習問題 2.3】 構造体と配列

項目，金額(+/−で収支を表す)の 2 項目で構成される構造体を作成し，こづかい帳のプログラムを作成しなさい。既存データがファイル「syushi.dat」として存在するとして，データを読み込んだ後，新規データを入力，合計とともに表示しなさい。余裕があれば，入力したデータを既存ファイルに追加するようにしなさい。



「syushi.dat」
30000,仕送り
−3000,雑誌
−8000,本
−3000,飲み代
80000,バイト代
−30000,食事代



【練習問題 2.4】 関数の呼び出しとポインタ

2 つに int 系の変数 x, y を受け取り，その和と差をまとめて返す(2 つの変数 wa, sa が指す int 型の変数に格納する)関数を作成し，その結果を確認しなさい。

【練習問題 2.5】

1 階の常微分方程式

$$dv/dt = F(v) = g - c \cdot v$$

を 4 次のルンゲ・クッタ法で解き, 時刻 $t = 0.0$ から 0.1 刻みで $t = 1.0$ までの v の値を求めるプログラムを作成しなさい。ただし, $g = 9.81$, $c = 0.1$ とする。

[ルンゲ・クッタ法]

1 階の常微分方程式を,

$dvv / dt = F(t, v)$

とする。変数の初期値を t_0, v_0 とし, t の刻みを Δt とすると, 時刻 $t_1 = t_0 + \Delta t$ における v の値 v_1 は,

$v_1 = v_0 + k$

で, 求められる。ただし,

$$
\begin{aligned}
k_1 &= \Delta t \cdot F(t_0, v_0) \\
k_2 &= \Delta t \cdot F\left(t_0 + \frac{\Delta t}{2}, v_0 + \frac{k_1}{2}\right) \\
k_3 &= \Delta t \cdot F\left(t_0 + \frac{\Delta t}{2}, v_0 + \frac{k_2}{2}\right) \\
k_4 &= \Delta t \cdot F(t_0 + \Delta t, v_0 + k_3) \\
k &= \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$

である。この t_1, v_1 の値から上の計算を同様に行い, 時刻 $t_2 = t_0 + 2 \cdot \Delta t$ における v の値 v_2 を求める。これを繰り返して行くのが 4 次のルンゲ・クッタ法である。

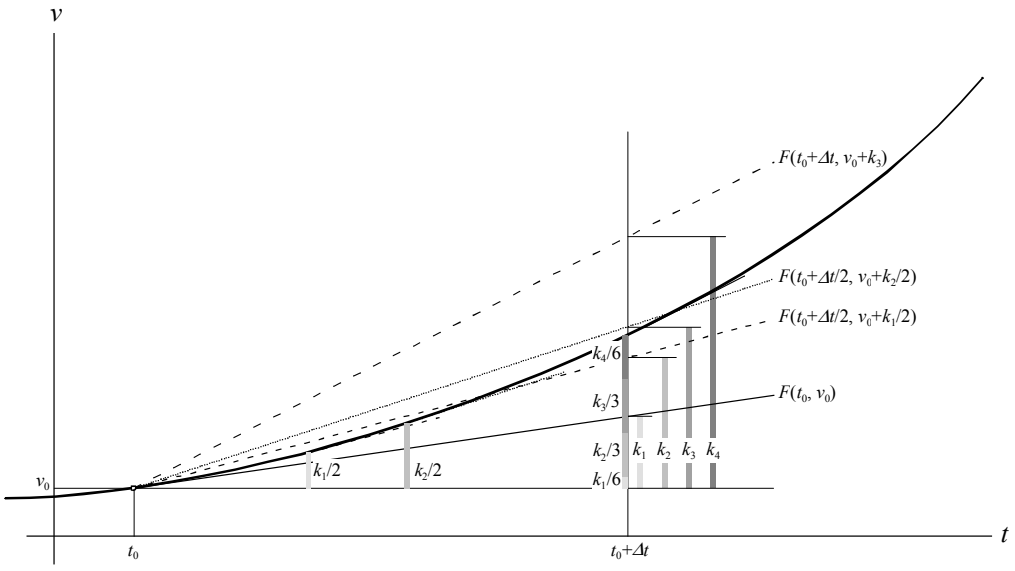


図2.12 ルンゲ・クッタ法