生物生産工学特論Ⅰ
応用数理解析学

C/C++言語による
実験・計測アプリケーションの作成

北海道大学大学院農学研究科
生物資源生産学専攻
生物生産工学講座

目　次

1.　C 言語（1）C 言語の基本

1.1.　プログラムができるまで

C 言語でプログラムを作成する流れを図 1.1 に示します。ユーザが作成するのは基本的に，プログラムの本体であるソースファイル(拡張子が.c)と，プログラム中に必要な定義を記述するヘッダファイル(拡張子が.h)です。これらに「コンパイル」という処理を行うと，オブジェクトファイル(拡張子が.obj)というファイルができます。この中には，プログラムを実行するのにはどのような標準的な機能(例えば，画面に表示したり，キーボードからの入力を理解したり)が必要かが記述されます。そして，「リンク」という処理をすると，オブジェクトファイルに記述してある標準機能をライブラリ(拡張子が.lib)から読み込み，組み合わせることで実行ファイル(.exe)を作成します。つまり，「**ソースファイルと**(必要なら)**ヘッダファイルを書いて，コンパイルとリンクをするとプログラムができる。**」ということです。

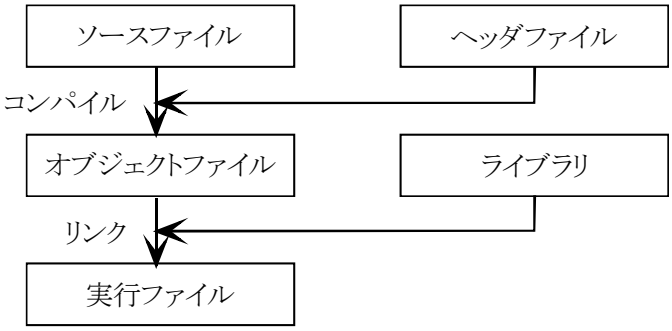


図1.1　プログラムができるまで

C 言語ソースは基本的に図 1.2 のような構造になっています。いくつかの基本規則があります。

1) ソースは 1 つまたは複数の関数からなり，main という関数から実行される。
2) 「#」で始まる命令はプリプロセッサ命令といい，コンパイル前に実行される。
3) 命令中の改行・スペースは無視され，「;」までを 1 行の命令とする。
4) 「/*」，「*/」で挟まれた部分はコメントとして，コンパイル時に無視される。
5) 「{」，「}」で挟まれた部分を 1 つのブロックとして処理する。

```
#include <ヘッダファイル>
```
～ ヘッダファイルの指定

```
int func1(void);
int func2(void);
```
～ プロトタイプ宣言（関数の定義）

```
int main(void)
{
        :
        return 1;
}
```
～ main関数（1つのプログラムに1つだけ存在）

```
int func1(void){
        :
}

int func2(void){
        :
}
```
～ その他の関数

図1.2　ソースファイルの基本構造

実際に C 言語で既述したプログラムソースを list1.1 に示します。各行の意味は以下の通りです。

1 行目　stdio.h を読み込むプリプロセッサ命令
3 行目　メイン関数の定義。引数はなし，戻値は int(整数)。
5 行目　変数 a, b を int と定義し，1, 2 を代入する。
7・10 行目　コメント
8 行目　printf という標準関数を使用して，「Hello world。」という文字を表示し，改行する。
11・12 行目　a と b の和・差を計算し，表示する。
14 行目　戻値として 1 を返して，この関数(すなわち，このプログラム)を終了する。

```
 1:    #include      <stdio.h>
 2:
 3:    int main(void)
 4:    {
 5:          int a = 1, b = 2;
 5:
 7:          /* 文字の表示 */
 8:          printf("Hello world.¥n");
 9:
10:          /* 計算式 */
11:          printf("A + B = %d¥n", a + b);
12:          printf("A - B = %d¥n", a - b);
13:
14:          return 1;
15:    }
```

list1.1　Cプログラムソースの例

このように，C 言語でプログラムを書くには，「**変数の使い方と標準関数を覚えて，独自の関数を書く。**」のが基本と言えます。

表 1.1 変数型

型	表記	意味	バイト	範囲	32bit
文字	char	文字	1	−128～128	1
整数	short (int)	整数	2	−32768～32768	2
	int		2	−32768～32767	4
	long (int)		4	-2^{31}～$2^{31}-1$	4
実数	float	浮動小数点実数(7 桁) 実数部 24bits，指数部 8bits	4	$\pm 3.4\times 10^{38}$	4
	double	倍精度浮動小数点実数(15 桁) 実数部 53bits，指数部 11bits	8	$\pm 1.8\times 10^{308}$	8
	long double	倍精度浮動小数点実数(15 桁) 実数部 64bits，指数部 16bits	10	$\pm 1.14\times 10^{4932}$	10

1.2. 変数の使い方

1.2.1. 変数型

C 言語で使用する基本変数型を表 1.1 に示します。それぞれの変数には，signed(符号つき)と unsigned(符号なし正数)の 2 種類があり，省略すると signed になります。(例えば，char は singed char と同意で−128～127 まで，unsigned char は 0～255 まで)

自動変数と静的変数

変数にはスコープと呼ばれる使用可能範囲があり，基本的に「{」から「}」までがその範囲となります。変数はスコープ内の宣言された時点でメモリ上に配置され，スコープがなくなった時点で消滅します。つまり，関数などで宣言した変数は，その関数の中でだけ使用でき，関数が終了すると共に消滅します。このように自動的に消滅する変数を自動変数と言います。通常の宣言で作成される変数は全て自動変数となります。

一方，変数の宣言時に「static」と付けるとある特定のメモリ領域が確保され，スコープがなくなっても変数は消滅しません。このような変数を静的変数と呼びます。静的変数も使用可能範囲はスコープ内なので他の関数から使用することはできませんが，スコープを失っても値を保持しているので，値を保存しておきたい時などに使用します。

ローカル変数(内部変数)とグローバル変数(外部変数)

変数はスコープによってローカル変数とグローバル変数に分けられます。ローカル変数は「{}から「}」までのスコープを持ち，その範囲内だけで使用できます。一方，グローバル変数は「{」から「}」の外で宣言をするので，スコープを持ちません(正確にはファイルスコープというスコープがある)。そのため，ファイル内のどの関数からも参照できます(「extern」という宣言を使うとファイル外からも参照できる)。関数が多くなると変数の受け渡しが面倒になるので，グローバル変数を多用したくなりますが，可搬性の問題から一般的には使用しないほうがいいとされています。

1.2.2. 定数

数定数 C 言語では 10 進数，16 進数が主に使用されます。10 進数はそのままの数字で，16 進数は数字の先頭に「0x」を付けて示します。また，long 型の定数には最後に「L」を付けます。

単一文字 char 型の変数は単一の文字を表すのにも使用され，「’(シングルクォーテーション)」で挟んで定義します。

文字列 文字列は char 型の配列として扱い，「”(ダブルクォーテーション)」で挟んで定義します。文字列の最後には必ず文字列の終端を示す NULL(ヌル，ナル)文字(¥0)が付けられます。

エスケープ文字 コントロールコードのように表現できない文字を示すためにはエスケープも時を使用します。エスケープ文字は記号「¥」の後に文字を続けて定義します。例えば，改行コードは「¥n」，タブコードは「¥t」です。

1.2.3.　演算子

算術演算子　加減乗除を示す「+」,「−」,「*」,「/」と，剰余(モジュロ)を示す「%」があります。

代入演算子　代入式を示す「=」と，複合演算子「+=」,「−=」,「*=」,「/=」があります。複合演算子の意味は以下の通りです。

```
x += 1;  →  x = x + 1;          x −= 1;  →  x = x − 1;
x *= 1;  →  x = x * 1;          x /= 1;  →  x = x / 1;
```

インクリメント／デクリメント演算子　変数の値を 1 増減させます。頻繁に使用しますが前置と後置で高価が違うので注意してください。

```
i++;  →  i = i + 1;             i−−;  →  i = i − 1;
++i;  →  i = i + 1;             −−i;  →  i = i − 1;
j = i++;  →  j = i; i = i + 1;  j = i−−;  →  j = i; i = i − 1;
j = ++i;  →  i = i + 1; j = i;  j = −−i;  →  i = i − 1; j = i;
```

関係演算子　条件を表す「==」,「!=」,「>」,「<」,「>=」,「<=」があります。

論理演算子　変数の論理演算を行う演算子です。NOT は「!」，論理 AND は「&&」，論理 OR は「||」を使用します。例えば，「もし，a < x < b ならば･･･をしなさい」は以下のようになります。

```
if((a < x) && (x < b))  ･･･;
```

条件演算子　条件分岐に使用します。「?」は条件に。「:」は分岐を示します。例えば，「a と b の大きいほうを max に代入する」は以下のようになります。

```
max = (a > b) ? a : b;
```

ビット演算子　ビット単位の演算を行います。ビット AND は「&」，ビット OR は「|」，XOR は「^」，左シフトは「<<」，右シフトは「>>」，1 の補数[1)]は「~」を使用します。

キャスト演算子　変数型を強制的に変換する時に使用します。「int の変数 i に double の変数 d を代入する」は以下のようになります。

```
i = (int)d;
```

ポインタ演算子　C 言語の変数型の一つに，ある変数のメモリアドレスとその型を保持する型，ポインタがあります。ポインタ演算子「*」はあるポインタ ptr の頭に*ptr と付けることで，その変数の実体を示すことができます。ポインタの扱いはかなり複雑なので改めて取り上げることにします。

アドレス演算子　変数の格納されるメモリアドレスを取り出す場合にはアドレス演算子「&」を使用します。

1.2.4.　制御文

プログラムの流れを制御するための文を制御文と言います。

if 文　条件分岐を制御します。

```
if(条件式){
        実行される文;
}
```

実行される文が 1 行の場合は「{」,「}」を省略することもできます。また，条件にあてはまらない時に実行する文は「else」で指定します。

```
if(条件式){
        実行される文;
}else{
        実行される文;
```

1) 1 の補数：各ビットに対して 1 を 0 に，0 を 1 にした値．2 の補数は 1 の補数に 1 を加えたもので，負数を示すのに用いられる．

```
}
```

if～else～を組み合わせることで多重分岐をすることもできます。

for 文　数値を変化させながら，ある処理を繰り返す制御を行います。

```
for(初期設定; 実行条件; 繰返し文){
        実行される文;
}
```

実行の流れは，まず初期設定が行われ，実行条件を評価します。実行条件を満たしていれば，ブロック内の文を実行し，繰返し文を実行します。その結果，実行条件を満たしていれば，繰り返しブロック内を実行し，条件を満たしていなければ，ブロックを抜けます。初期設定には繰り返しに使用する変数以外の変数を指定してもかまいません。

```
int     a, b, i;
for(a = 0, b = 0, i = 0; i < 100; i += 2){
        実行される文;
}
```

また，実行条件を設定しなければ無限ループになります。無限ループは抜けることができなくなる時もあるので，注意しましょう。

while 文　for 文と同様に繰り返しを制御するために使用します。

```
while(条件式){
        実行される文;
}
```

実行の流れは，まず条件式が評価され，条件式が真(TRUE)であれば，ブロック内を実行します。そのため，条件式が初めから偽であれば，文は実行されません。

do～while 文　while 文の変形で，条件判断はループの出口で行われます。そのため必ず 1 度はブロック内の文は実行されます。

```
do{
        実行される文;
}while(条件式);
```

最後の「;」を忘れずに。

switch 文　多重分岐をするために用います。

```
switch(変数){
case 条件 1:    実行される文 1;
                break;
case 条件 2:    実行される文 2;
                break;
・・・・・・・・・・・
default:        実行される文 n;
}
```

実行は指定された変数をもとに条件分岐を行います。変数には整数を使用します。条件にあてはまるラベルがあった場合はそのラベルにあたる文を実行します。文の途中に break 文があると実行を中止し，switch 文の最後までジャンプします。break 文がない場合は，次のラベルに行っても処理を継続します。条件にあてはまるラベルがない時は，default で指定された処理を行います。default で何もしたくない時は continue;を記述しておきます。

1.3. 標準関数

1.3.1. 標準入出力関数 （使う時は，stdio.h を include する）

1) 1 文字の入出力

`int getchar(void);`

標準入力(普通はキーボード)から 1 文字入力する。成功すると読み込んだ文字を，エラーで「EOF」(End Of File: stdio.h で定義済みのファイルエンドを示す値)を返します。

`int putchar(int c);`

標準出力(普通は画面)に 1 文字出力する。成功すると文字「c」を，エラーで「EOF」を返します。

2) 文字列の入出力

`char *gets(char *s);`

標準入力から復帰文字(リターン)で終了する文字列を読み込んで，s に格納します。s の中の復帰文字は NULL に変換されます。成功すると文字列 s を，エラーで NULL を返します。

`int puts(const char *s);`

標準出力に NULL で終了する文字列 s を出力し，最後に改行をします。成功すると負でない値を，エラーで EOF を返します。

3) 書式付き入出力

`int printf(const char *format, ...);`

format に文字列や書式文字列，その後ろに書式指定に対応した引数を指定することで，書式付きの出力を行います。書式文字列に変換指定子を入れることで，様々な出力をすることができます。各変換指定子は「%」で始まり，左から右に解釈されます。format の左から最初の書式指定子を(もしあれば)見つけると，format の後の引数をその書式に従って変換して出力します。2 つ以上の書式指定子がある場合も同様です。各書式指定子は，

% [flag] [width] [.prec] [h | l | | I64 | L] type_char

という形式で記述され，それぞれ以下のような意味を持っています。[ ]のものは省略も可能です。

[flags]　出力の位置決めと符号，空白，小数点，10 進数・8 進数・16 進数の出力を制御します。1 つの書式指定に，複数のフラグを指定できます。

- −　左詰めで表示する。
- +　出力値が符号つきの場合，「+」または「−」を表示する。
- 0　最小幅まで 0 が付加される。0 フラグと−フラグを指定すると無視される。整数書式(i, u, x, X, o, d)では無視される。
- ’ ’(空白)　出力値が符号つきで整数であると，出力値の前に空白もしくは「−」を表示する。+フラグと指定すると無視される。
- #　o, x, X で 0 以外のすべての出力の前に 0, 0x, 0X が着く。e, E, f で出力値に強制的に小数点が着く。c, d, i, u, s では無視される。

[width]　出力する最小文字数を指定する。

[.prec]　出力する最大文字数，または出力精度を指定します。

[h | l | | I64 | L]　入力引数のサイズを指定します。「h」は short，「l」，「L」は long，「I64」は_int64(Windows 用)を示します。

type_char　変数の種類を指定します。

- c　1 個の文字 （ただし，1byte 文字）
- d　符号付き 10 進整数
- i　符号付き 8 進整数
- o　符号なし 8 進整数
- u　符号なし 10 進整数

x	符号なし 16 進整数（a, b, c, d, e, f を使用）
X	符号なし 16 進整数（A, B, C, D, E, F を使用）
e	[-]d.dddd e [sign]ddd 形式符号付きの値。d は 1 個の 10 進数，dddd は 1 個または複数の 10 進数，ddd は正確に 3 桁の 10 進数，sign は + または -。
E	e と同じ。「e」の代わりに「E」を用いる。
f	小数
g	f または e の書式を，数値の精度によって使い分ける。
G	g と同じ。「e」の代わりに「E」を用いる。
p	ポインタ
s	文字列

成功すると出力した文字数を，エラーで負の値を返します。

`int scanf(const char *format, ...);`

書式付きの読込みを行います。format には printf と同様の書式指定子を用います。ただし，後続の引数にはポインタを使用します。成功すると変換された入力フィールドの数を，エラーで EOF を返します。scanf で書式指定子に%s を用いたときは空白文字(空白，タブ，または改行)までをひとまとまりとして読み込みます。空白文字で区切られていない文字列を読み取るには%s の代わりに%[]を指定します。対応する入力フィールドには[]で囲まれた文字セットに含まれていない最初の文字が見つかるまで読み取られます。また文字セットの最初の文字がカレット(^)のときは逆に作用します。例，%[a-z]は"a～z"以外の文字まで，%[^,]は",の直前までの文字が読み込まれます。

4) 文字列に対する書式付き入出力

`int sprintf(char *buffer, const char *format, ...);`

文字列 buffer に書式付き出力を行います。仕様は printf と同じです。

`int sscanf(const char *buffer, const char *format, ...);`

文字列 buffer から書式付き入力を行います。使用は scanf と同じです。

1.3.2. ファイル入出力関数 （使う時は，stdio.h を include する）

データをファイルに保存したり，読み込んだりする場合に使用します。C 言語でファイルを読み書きするには，あらかじめファイルをオープンしておく必要があります。オープンされたファイルはファイルストリームと呼ばれるデータで管理され，どのようなモード(読み込みのみ，書き込みのみ，読み書き両用など)か，どこまで読み書きしたか，などが管理されています。このストリームの中には標準入力(stdin)，標準出力(stdout)，標準エラー出力(stderr)なども含まれ，あらかじめオープンしてあります。

1) ファイルストリーム操作関数

`FILE *fopen(const char *filename, const char *mode);`

filename で指定されたファイルを mode で指定されたモードでオープンし，ストリームに結びつけます。モード文字列は以下のようになっています。

"r"	読出モード
"w"	書込モード:ファイルがない時は新しく作成します。
"a"	追加モード:ファイルがない時は新しく作成します。
"r+"	読出，書込両用モード:既存ファイルのみ対象となります。
"w+"	読出，書込両用モード:既存ファイルがあっても上書きします。
"a+"	読出，追加両用モード:ファイルがない時は作成します。

モード文字列に「b」を追加するとバイナリモードに，「t」を追加するとテキストモードになります。指定しないとグ

ローバル変数「_fmode」に従ってオープンします。標準ではテキストモードです。

成功するとストリームをさすポインタを，エラーのときは NULL を返します。

`int fclose(FILE *stream);`

オープンしたストリーム stream をクローズします。成功すると 0 を，エラーで EOF を返します。

`int fcloseall(void);`

オープンしているすべてのストリームをクローズします。成功するとクローズしたストリーム数を，エラーで EOF を返します。

2) ファイルに対する 1 文字の入出力

`int getc(FILE *stream);`

stream から 1 文字を読み込みます。成功すると読み込んだ文字を，エラーもしくはファイルエンドで EOF を返します。

`int putc(const int c, FILE *stream);`

stream に 1 文字を出力します。成功すると文字 c を，エラーで EOF を返します。

3) ファイルに対する文字列の入出力

`char *fgets(char *s, int n, FILE *stream);`

stream から文字列を読み込んで s に格納します。読込みは，n-1 個の文字を読み込むか，改行文字を読み込んだときに終了します。改行文字で終了した場合は改行文字も保存し，その後ろに NULL 文字を付加します。成功すると文字列 s を，エラーで NULL を返します。

`int fputs(const char *s, FILE *stream);`

stream に文字列 s を書き込みます。成功すると最後に書き込まれた文字を，エラーで EOF を返します。

4) ファイルに対する書式付き入出力

`int fprintf(FILE *stream, const char *format, ...);`

`int fscanf(FILE *stream, const char *format, ...);`

ファイルストリームに入出力を行う以外は printf, scanf と同じです。

ファイル入出力の一般的な例

ファイルからの読み込み	ファイルへの書き出し

`FILE * fp;`
`fp = fopen("xxxxx.txt", "rt");`
`if(fp == NULL){`
`  printf("Cannot open file¥n");`
`  return 0;`
`}`
`fscanf(fp, ...)`
`fclose(fp);`
	`FILE * fp;`
	`fp = fopen("xxxxx.txt", "wt");`
	`if(fp == NULL){`
	`  printf("Cannot create file¥n");`
	`  return 0;`
	`}`
	`fprintf(fp, ...);`
	`fclose(fp);`

1.3.3.　コンソール入出力関数 （使う時は，conio.h を include する）

ハードウェアから直接入出力する関数。Windows 上ではあまり使用しないほうが良い。

`int kbhit(void);`

キーボードが押されたかをチェックします。押されたキーは getch か getche で取得できます。キーが押されていれば 0 以外の整数，押されていなければ 0 を返します。

`int getch(void);`

`int getche(void);`

getch はエコーバック(画面表示)なし，getche はエコーバック付きでキーボードから文字を読み込みます。文字コードを返します。

1.3.4. 文字列操作関数 (使う時は, string.h を include する)

1) 文字列のコピー

```
char *strcpy(char *dst, const char *src);
char *strncpy(char *dst, const char *src, size_t maxlen);
```

文字列 src を dst にコピーします。コピーは NULL 文字をコピーしたところで終了します。strncpy は最大文字数 maxlen 個までコピーし, 必要なら NULL を付加します。文字列 dst へのポインタを返します。

2) 文字列の追加

```
char *strcat(char *dst, const char *src);
char *strncat(char *dst, const char *src, size_t maxlen);
```

文字列 src を dst の最後に追加します。得られる文字列の長さは dst の長さ+src の長さになるの, dst のサイズに注意してください。srrncat は最大文字数 maxlen 個まで追加し, 必要なら NULL を付加します。文字列 dst へのポインタを返します。

3) 文字列の比較

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t maxlen);
```

文字列 s1 と文字列 s2 の各文字を比較します。strncmp は最大文字数 maxlen 個まで比較します。両者とも以下のような値を返します。

s1 が s2 よりも小さければ	< 0
s1 が s2 と等しければ	= 0
s1 が s2 よりも大きければ	> 0

4) 文字列の長さを知る

```
size_t strlen(const char  *s);
```

文字列 s の長さを返します。NULL 文字は数に含まれません。

1.3.5. その他の標準関数 (使う時は, stdlib.h を include する)

1) 乱数の発生

```
int rand(void);
void srand(unsigned int seed);
```

周期 2^{32} の 乗法合同法を用いて乱数を発生し, 呼び出されるたびに 0～RAND_MAX (設定されていないものもある)の間で擬似乱数を発生させます。srand は, 適当な seed を与えることで, 乱数発生の新しい開始点にセットします。

2) クイックソートアルゴリズム

```
void qsort(void *base, size_t num, size_t width, int (*compare)(const void *elem1,
            const void *elem2));
```

クイックソートアルゴリズムを使用して, 配列 base の num 個, サイズ width バイトのデータを関数 compare に従ってソートします。比較関数を func(a, b)とした時, 戻値を下のように定義すると昇順にソートされます。

比較	a < b	戻値	< 0
	a = b		= 0
	a > b		> 0

1.3.6. 数学関数 (使う時は, math.h を include する)

1) 三角関数

```
double sin(double x);  (cos, tan, asin, acos, atan も型は同じ)
```

`double atan2(double y, double x);`

sin, cos, tan はそれぞれ x の三角関数値を返します。角度の単位は rad です。asin, acos, atan はそれぞれ x の逆参画関数値を返します。atan2 は y/x の atan 値を返します。atan は角度が$\pi/2$, $-\pi/2$ のとき誤差が大きいので，一般的に atan2 を使用します。

2) 対数・指数関数

`double log(double x);` （log10, exp も型は同じ）

`double pow10(int p);`

log, log10 はそれぞれ x の自然対数，常用対数を示します。exp, pow10 はそれぞれ基数 e, 10 の指数関数値を示します。

3) その他の関数

`double ceil(double x);`

`double floor(double x);`

ceil は x の小数点以下を切り上げ，floor は小数点以下を切り捨てた値を返します。

`double fmod(double x, double y);`

x を y で割った剰余を返します。y = 0 のときは 0 を返します。

`double pow(double x, double y);`

x の y 乗を返します。

`double fabs(double x);`

x の絶対値を返します。

`double modf(double x, double *ipart);`

x を整数部と小数部に分割し，整数部を ipart に格納し，小数部を返します。

`double sqrt(double x);`

x の平方根を返します。

`double hypot(double x, double y);`

直角三角形の 2 辺の長さ x, y から斜辺の長さを返します。

`double poly(double x, int degree, double *coeffs);`

係数 coeffs[0], coeffs[1], ‥‥, coeffs[degree]をもつ degree 次の x に関する多項式の x における値を返します。

【練習問題 1.1】 if 文

数字を入力し，入力されたのが 1 なら"Yes"を，それ以外なら"No"を表示するプログラムを作成しなさい。

【練習問題 1.2】 for 文

入力した数までの階乗を求めて表示するプログラムを作成しなさい。

【練習問題 1.3】 while 文

入力された数字が 0 になるまで，数字の入力と入力した数字の表示を繰り返すプログラムを作成しなさい。

【練習問題 1.4】 switch 文

1 が入力されたら「One.」と表示，2 が入力されたら「Two.」と表示，その他は「It is neither one nor two.」と表示するプログラムを作成しなさい。

【練習問題 1.5】 コンソール関数

キーボードから入力があるまで"Waiting for keyin."と何度も表示する無限ループ(for 文や while 文)をつくり，キー入

力されたら押されたキーのキーコードを表示して終了するプログラムを作成しなさい。

【練習問題 1.6】 数学関数

以下のものを計算し，表示せよ。

(1) 2^8　(2) sin(67°)　(3) –23.456 の絶対値　(4) 7÷3 の剰余　(5) 0～1 の範囲の乱数を 10 個

【練習問題 1.7】 標準入出力関数

100 個のデータ列 t (整数), x, y, z (0～1 のランダムな実数)を以下の形式でファイルに出力するプログラムを作成しなさい。データはカンマ区切りで出力するものとする。

例

```
t,x,y,z
0,0.567464,0.4544894,0.448468
1,0.164894,0.159618,0.4848975
:
:
99,0.4488,0.97845,0.2712634
```

【練習問題 1.8】 標準入出力関数

練習問題 1.7 で作成したデータファイルを読み込みながら，表示するプログラムを作成しなさい。

【練習問題 1.9】 標準入出力関数

練習問題 1.7 で作成したデータファイルを読み込んで，x に対する y (もしくは z)の線形回帰直線を求め，表示するプログラムを作成しなさい。

[線形回帰]

n 個の観測値 x，y に対して $y = ax + b$ の線形関係があるとすると，観測点 i の自乗誤差 ε_i^2 は，

$$\varepsilon_i^2 = \{y_i - (ax_i + b)\}^2 = y_i^2 + a^2 x_i^2 + b^2 - 2ax_i y_i + 2abx_i - 2by_i$$

となり，観測値全部の二乗誤差和は，

$$S_\varepsilon = \sum \varepsilon_i^2 \text{，} Sxx = \sum x_i^2 \text{，} Syy = \sum y_i^2 \text{，} Sxy = \sum x_i y_i \text{，} Sx = \sum x_i \text{，} Sy = \sum y_i$$

とおくと，

$$S_\varepsilon = Syy + a^2 \cdot Sxx + nb^2 - 2a \cdot Sxy + 2ab \cdot Sx - 2b \cdot Sy$$

となる。この誤差を最小(=0)とする a，b は，

$$\partial S_\varepsilon / \partial a = 2a \cdot Sxx - 2Sxy + 2b \cdot Sx = 0$$

$$\partial S_\varepsilon / \partial b = 2nb + 2a \cdot Sx - 2Sy = 0$$

より，

$$\begin{pmatrix} Sxy \\ Sy \end{pmatrix} = \begin{pmatrix} Sxx & Sx \\ Sx & n \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix}$$

よって，

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{n \cdot Sxx - Sx^2} \begin{pmatrix} n & -Sx \\ -Sx & Sxx \end{pmatrix} \cdot \begin{pmatrix} Sxy \\ Sy \end{pmatrix}$$

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。

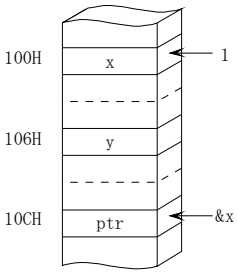


図2.1　ポインタ

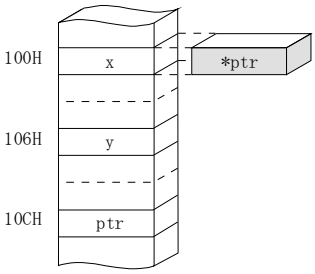


図2.2　エイリアス

```
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。

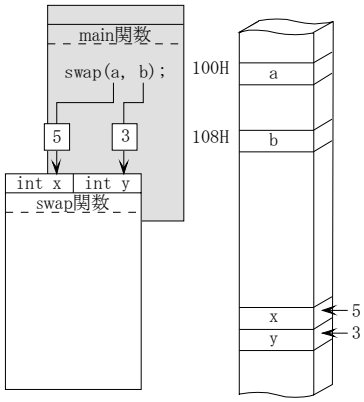


図2.3 関数呼出し1

```
#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。

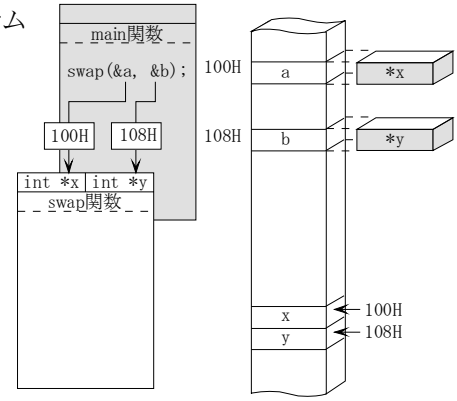


図2.4 関数呼出し2

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。

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

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

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。

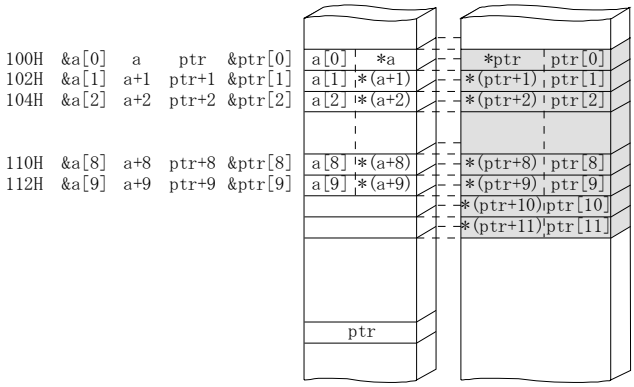


図2.5　配列とポインタ

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。

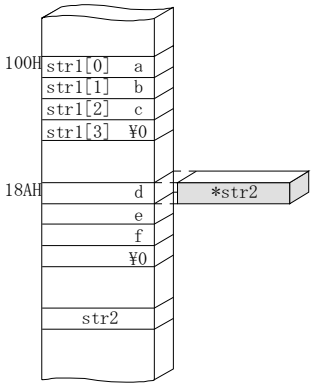


図2.6　文字列

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)
        ;
```

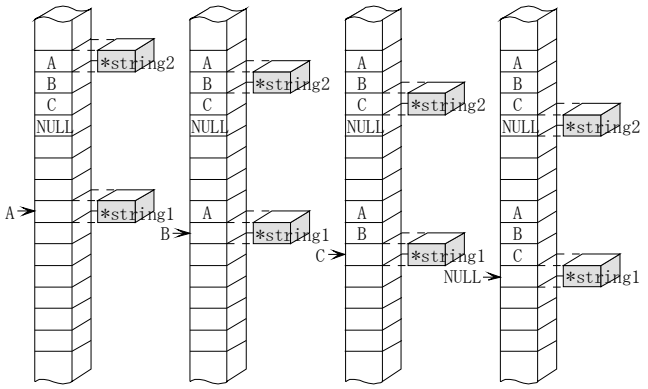


図2.7　文字列のコピー

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

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

list2.6 strcpyらしく

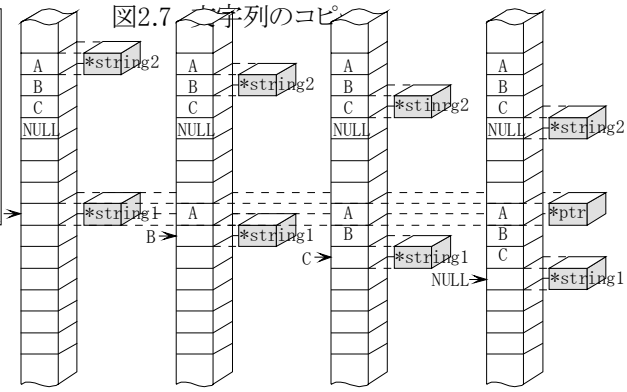


図2.8　strcpyらしく

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。

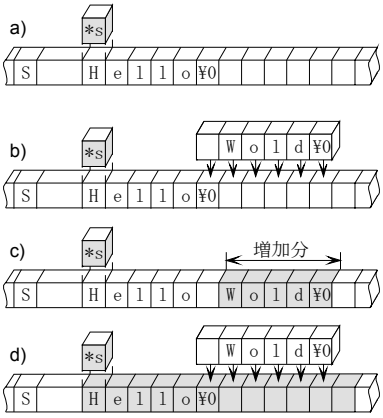


図2.9　文字列の追加

※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。

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。

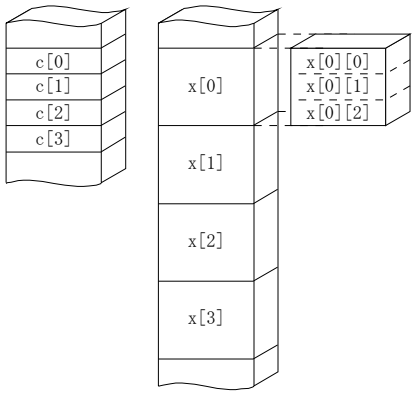


図2.10　多次元配列

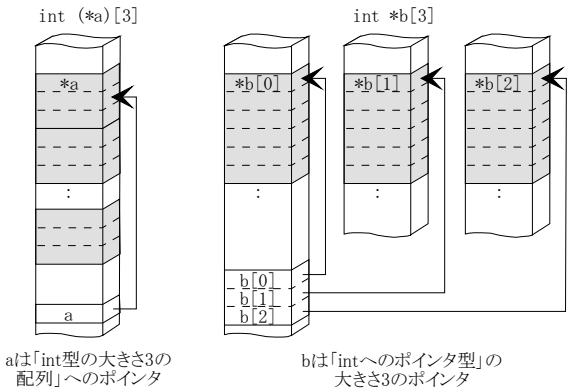


図2.11　int (*a)[3] と int *b[3] の違い

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

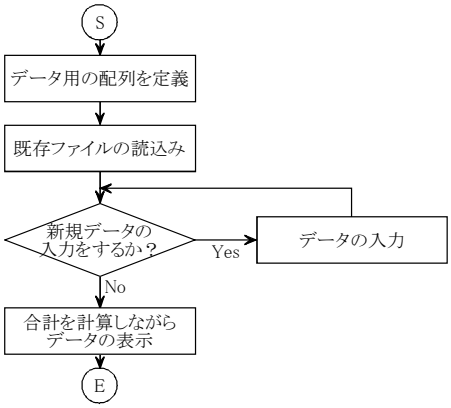
振幅 A，周期 T，波長 λ の x 軸上を進む正弦波の時間 t，位置 x における変位は $y = A \cdot \sin(2\pi(t/T - x/\lambda))$ で表される。$A = 50$，$T = 4$，$\lambda = 10$，$x = 0,2,4,\cdots,20$ の 11 個の点に置かれた上述の調和振動子の任意の時刻（t を適当に与える）における変位をそれぞれ配列に格納し，表示しなさい。

【練習問題 2.2】 ポインタと配列

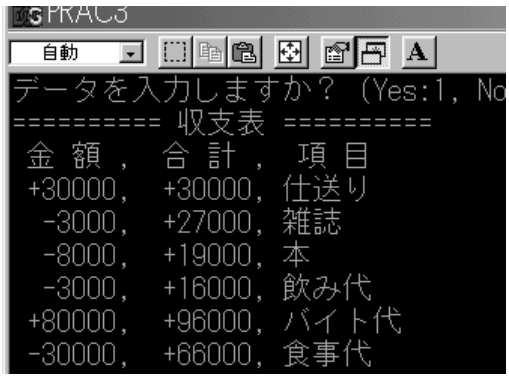
練習問題 1.7 で作成したデータ t, x, y を配列に読込み，一覧に表示するプログラムを作成せよ。

【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。

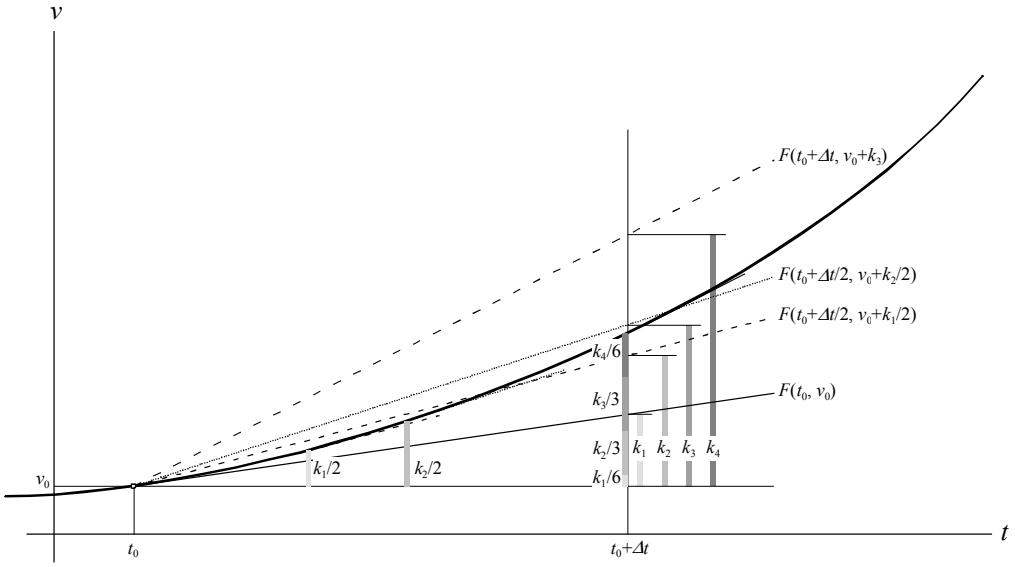


図2.12 ルンゲ・クッタ法

3.　C++言語

3.1.　C++言語とは

C++言語とは，C 言語のスーパーセットとして作られたというよりも，オブジェクト指向のために C 言語をベースに新たに作った言語というのが正しいようです。しかし，実際には C 言語の機能はほとんど C++の中に取り込まれていますし，C の中で問題のあった機能は書き直され，より使いやすくなっています。新しい言語を覚えるというよりも，もっと便利になった C を使うという方が適切なようです。また，C++の特徴となっているクラスというものがあります。よく構造体と比較されますが(比較についてはここでは省略します。興味のあるひとは本でも見てみてください)，実際には構造体にはないとても便利な機能がたくさんあります。

C++でソースを書くには，これまでＣで拡張子を「c」にしていたのを「cpp」にするだけです。あとは特に気を付けることはありません。

3.2.　C++の便利な機能

C++を使うことで，C と比較して便利になる機能を説明します。

3.2.1.　デフォルト引数

C では，プロトタイプ宣言で記述された変数をきちんと書かないとエラーになりました。C++ではあらかじめデフォルトの引数を指定しておくと，引数の省略ができるようになります。

```
int func(double a = 1.2, double b = 3.4);
```

という具合です。これを呼び出すときは，

```
func();
func(5.6);
func(7.8.9.0);
```

と書きます。最初の例では，引数を全部省略しているのでfunc(1.2, 3.4)と同じ意味です。2番目の例では，引数を1つ省略しているのでfunc(5.6, 3.4)と同じ意味になります。最後の例は全部の値を指定しています。注意することは，最初の引数を省略して書くことはできないということです。

3.2.2.　変数の型宣言

C では，変数を関数内で宣言するときは必ず最初の方に書かないとエラーになりました。しかし，実際にソースを書いていくと途中で変数が欲しくなったりします。そういうときに，毎回先頭に戻って書くというのはとても不便です。しかし，C++は変数が必要になったとき，その変数が使われる前だったらどこで宣言してもいいことになっています。

ただし，間違いも起こりやすいので注意が必要です。list3.1 は間違いの例です。while 文の中で宣言した変数 a, n はループから抜けた時点で使えなくなってしまいます。この場合はコンパイラがエラーを出してくれるので気がつくはずですが，エラーの出ないような時は厄介なバグになります。

ちなみに，この例の場合，n の宣言に static を使っていますが，どうしてかわかりますか？ちょっと考えてみてください。

```
#include    <stdio.h>

int main(void)
{
    printf("数字を入力してください. ¥n");
    printf("0入力で終了します. ¥n");
    while(1){
        int a;
        static int n = 0;
        scanf("%d", &a);
        if(a == 0)
            break;
        n++;
        printf("第%d回の入力は%dです. ¥n",
            n, a);
    }
    printf("n = ", n);  // エラー！
    printf("a = ", a);  // エラー！
    return 0;
}
```

list3.1 変数宣言の間違いの例

3.2.3.　関数のオーバーロード

C++では関数のオーバーロードといい，引数が異なる同じ名前の関数を作ることができます。使い方によって，大変便利にも，厄介なバグの原因にもなります。使い方は，list3.2 の例を見れば一目瞭然でしょう。

```
#include    <stdio.h>

void print(int n);
void print(char *s);
void print(double d);

int main(void)
{
    print(1);
    print("Overload");
    print(123.4);
    return 1;
}

void print(int n){
    printf("%d¥n", n);
}

void print(char *s){
    printf("%s¥n", s);
}

void print(double d){
    printf("%lf¥n", d);
}
```

list3.2 関数のオーバーロード

3.2.4.　オペレータオーバーロード

オペレータとは「+, -, *, ... 」のことです。オペレータオーバーロードとはこれをオーバーロードするということです。関数のオーバーロードとほとんど同じです。一般的には後述のクラスと一緒に使いますが，クラス以外にも使いみちはあります。

オペレータオーバーロードの例としては，大抵の参考書が複素数の演算を出しています。ちょっと先走ってしまいますが，複素数クラスを使った例を示しましょう。ただし，list3.3 を見たらわかりますが，数字に double しか使えない，演算も「+」しかない簡略版です。

```
#include    <stdio.h>

class Complex{
    double real;    // 実数部
    double image;   // 虚数部
public:
    Complex(double r, double i);
    void print(void);
    Complex operator + (Complex x);
};

Complex::Complex(double r, double i){
    real = r;
    image = i;
}

void Complex::print(void){
    printf("%lf+%lfi¥n", real, image);
}

Complex Complex::operator + (Complex x){
    Complex temp(0.0, 0.0);
    temp.real = real + x.real;
    temp.image = image + x.image;
    return temp;
}

int main(void)
{
    Complex A(3.0, 2.0), B(5.0, 6.0);
    Complex C(0.0, 0.0);

    printf("A = ");
    A.print();
    printf("B = ");
    B.print();
    C = A + B;
    printf("A + B = ");
    C.print();
    return 1;
}
```

list3.3 オペレータオーバーロードの例

3.2.5.　new と delete

C 言語で配列の説明をしました。配列の宣言をするためにはあらかじめ配列のサイズを指定する必要がありました。しかし，実際に使うときにはプログラムを実行した後で，配列の大きさを決めたいときがあります。そのようなときに使うのがメモリの動的確保です。C 言語でも方法はありますが[1]，いくつか問題点もあるため，あまりお薦めできません。

---

1) 確保には「malloc」や「calloc」を，開放には「free」を使います．使用法が複雑なだけでなく，これらの関数にバグのある OS，コンパイラが多いので，できるだけ使わないようにとなっているものが多いです．

C++でメモリの動的確保を行うには new という演算子を使います。

new データ型;
new データ型[個数]

のように使います。失敗すると 0 を，成功すると領域の先頭アドレスを返します。使い終わったら，delete 演算子を使ってメモリの開放をします。list3.4 の例を見てみましょう。

```
#include    <stdio.h>

int main(void)
{
    int     n;
    printf("データの個数は？ : ");
    scanf("%d", &n);

    int *ptr;
    ptr = new int[n];
    for(int i = 0; i < n; i++){
        printf("data[%d] = ", i);
        scanf("%d", &ptr[i]);
    }
```

```
    printf("入力データの表示¥n");
    for(i = 0; i < n; i++){
        printf("data[%d] = %d¥n",
            i, ptr[i]);
    }
    delete []ptr;

    return 1;
}
```

list3.4 newとdeleteの例

確保したい領域のデータ型へのポインタを宣言し，ptr = new int[]; のように領域を確保します。使用が終わったら，delete [] ptr; のように領域を開放します。この場合のように，ptr が配列の場合は，空の[]を ptr の前に付けます。

3.3.　クラス

C++の一番の特徴としてあげられるのがクラスです。一見するとクラスと構造体の違いは，関数が含まれるかどうかだけのように見えます。実際，使い方によっては全く違いがないときもあります。しかし，クラスにはオブジェクト指向を実現するための，アクセスコントロールとインヘリタンスという強力な機能を持っているという点で構造体と区別されます。

簡単なクラスの構造は，

class クラス名{
アクセスコントロール:
　　クラスメンバ宣言;
};

です。アクセスコントロールとは，クラスメンバがメンバ以外からアクセスできるかどうかを制御します。アクセスコントロールの代表的なものに public と private があります。public メンバは外部からアクセスできるメンバ，private メンバは外部からアクセスできないメンバと考えてください。

```
class test{
    int a;
    char str[32];
public:
    int b;
    void ShowClass(void){
        . . . . . .
        . . . . . .
    }
};
```

＝

```
class test{
    int a;
    char str[32];
public:
    int b;
    void ShowClass(void);
};

void test::ShowClass(void){
    . . . . . .
}
```

左のように定義すると，a や str はプライベートメンバで外部からアクセスは不可能になり，b や関数 ShowClass はパブリックメンバで外部からアクセスが可能となります。また，関数 ShowClass の中身をそのまま書いていますが，これでは不便です。そこで，長い関数は右のように記述されることが多いです。ここで，「::」はスコープ演算子と呼ばれ。

戻値の型 クラス名 :: 関数名（引数，. . . ）

のように使います。

実際に使った例を list3.5 に示します。

```
#include    <stdio.h>

class cl05{
    int                 a;
public:
    int                 b;
    void ShowClass(void);
};

void cl05::ShowClass(void){
    printf("数字を入力してください : ");
    scanf("%d", &a);
    printf("a = %d¥n", a);
    printf("b = %d¥n", b);
```

```
int main(void)
{
    cl05                first;
    cl05                second;

    first.b = 100;
    first.ShowClass();
    second.b = 200;
    second.ShowClass();

    return 1;
}
```

list3.5 クラスの例

3.3.1.　コンストラクタとデストラクタ，メンバ関数

クラスの初期化と終了処理をする関数がコンストラクタとデストラクタです。これらには一般的な関数とは少し違った特徴があります。

1. コンストラクタは，クラスの名前と同じ名前である。
2. コンストラクタは，引数を取ることができる。オーバーロードもできる。
3. コンストラクタは，戻り値がない(void 型でもない)。
4. デストラクタは，クラスの名前の前にチルダ「~」を付ける。
5. デストラクタは，引数がない。戻り値もない。
6. オブジェクトの宣言と同時にコンストラクタが呼ばれ，スコープを失うと，デストラクタが呼び出される。
7. コンストラクタもデストラクタも省略されると暗黙のうちに引数や，中身のない関数が定義される。
8. コンストラクタもデストラクタも public な関数である。

ということです。では，list3.6 の例を見てみましょう。

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

class String{
private:
    int     len;
    char    *str;
public:
    String(char *string);
    ~String();
    void Print(void);
};

String::String(char *string){
    len = strlen(string);
    str = new char[len + 1];
    strcpy(str, string);
    printf("%sのコンストラクタ¥n", str);
```

```
String::~String(){
    printf("%sのデストラクタ¥n", str);
    len = 0;
    delete [] str;
}

void String::Print(void){
    printf("文字列は「%s」です. ¥n", str);
    printf("文字長は%d文字です. ¥n", len);
}

int main(void)
{
    String  Hitotsume("一つ目");
    String  Futatsume("二つ目");
    Hitotsume.Print();
    Futatsume.Print();
    return 1;
```

list3.6 コンストラクタ・デストラクタの例

ここでは文字列を扱う String というクラスを定義することとします。文字列を扱うために，文字列長 len と文字列を保持するための領域へのポインタ str を定義しておきます。コンストラクタ String では，文字列の長さを len に格納し，文字列を保持するための領域を new で確保してから，そこに引数で与えられた文字列を格納しています。デストラクタ ~String では，文字列の長さ len を 0 にして，コンストラクタで確保した領域を解放しています。コンストラクタとデストラク

タの中で表示しているのは，どこで呼ばれているかがわかるようにするためです。

次に，main 関数を見てみると，Hitotume と Futatsume というオブジェクト(クラスで作成した変数をオブジェクトと言います)を作成し，その後 Print 関数でそれぞれのオブジェクトを表示して，終了しています。

さて，このプログラムを実行するとどうなるでしょう。特にどこで，どのようにコンストラクタとデストラクタが呼ばれているかに注意して見てください。

メンバ関数については，中でもう既に使ってしまっています。String::Print がそうです。スコープ演算子がついている以外は，C で出てきた一般的な関数と同じです。

3.3.2.　インヘリタンス

インヘリタンス(inheritance)とは「相続」「継承」という意味の英語で，C++ではクラスを継承することができます。クラスの継承をするためには，クラスの定義で次のように書きます。

```
派生クラス ： アクセスコントロール 基本クラス {
    :
};
```

派生クラスとは，基本クラスを元にして新しく導出されるクラスのことをいいます。ここで注意しなくてはいけないのが，アクセスコントロールです。ここには public，protected，private のいずれかが入ります。このアクセスコントロールと基本クラスメンバのアクセスコントロールによって派生クラスからのアクセスが決定されます。まとめると表 3.1 のようになります。

継承先のクラスには基本クラスのメンバが全て含まれます。また，派生クラスで新たなメンバを追加することもできます。list3.7 のように宣言した基本クラス A，派生クラス B で，関数 ShowA()，関数 ShowB()，関数 main()からアクセスできる変数はどれかを確認してみましょう。

表3.1 インヘリタンスとアクセスコントロール

基本＼派生	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	アクセス不可		

	a	b	c	d	e	f
ShowA()				―	―	―
ShowB()						
main()						

```
class A{
private:
    int a;
protected:
    int b;
private:
    int c;
    void ShowA(void);
};

class B : public A{
private:
    int d;
protected:
    int e;
public:
    int f;
    void ShowB(void);
};
```

```
void A::ShowA(void){
    printf("%d¥n", a);
    printf("%d¥n", b);
    printf("%d¥n", c);
}

void B::ShowB(void){
    printf("%d¥n", a);
        :
    printf("%d¥n", f);
}

int main(void)
{
    B test;
        :
    return 1;
}
```

list3.7 インヘリタンス

3.3.3.　関数のオーバーライド

基本クラスの関数を派生クラスで書き換えてしまうことを関数のオーバーライドと言います。手順は，

1. 基本クラスでオーバーライドされる関数を virtual として宣言する。(仮想関数という)
2. 継承クラスでオーバーライドする関数を宣言する。

です。例を list3.8 に示します。

3.3.4.　コンポジション

クラスはメンバとして(クラス)オブジェクトを持つことができます。この方法により新しいクラスを作ることをコンポジションといいます。オブジェクトを持つというだけではそれほど難しいことはありません。ただ，一つ注意しなくてはいけないのが，引数のあるコンストラクタをもつオブジェクトをもつコンポジションです。もし，基本クラス Kihon のコンストラクタが int

の引数を一つとるクラスの場合，

```
class Composition{
public:
    Kihon kihon(5);
};
```

と書くとどうなるでしょう？実際にソースを書いてみると判りますが，これはエラーになります。では，どうすればいいのでしょうか。このように引数を伴う場合は，

```
class Composition{
public:
    Kihon kihon;
};
```

と，引数なしで書いておき，コンストラクタで，

```
Composition :: Composition() : kihon(5){
    :
}
```

と，書きます。これをメンバイニシャライザといいます。

3.3.5.　多重継承

今までは継承する基本クラスは1つだけでした。基本クラスが複数ある場合はどうなるでしょう。2つのクラス class A と class B があるとき，これらを基本クラスとする派生クラス class C をつくるには，

```
class C : public A, public B {
    :
}
```

と書きます。A，B それぞれのクラスに同名の str というメンバ変数があるとき，C のなかでそれぞれにアクセスするには，A::str，B::str のようにスコープ解決演算子を使用します。

3.4.　ストリーム

3.4.1.　入出力ストリーム

C++でも printf や scanf のような C の関数は使えました。しかし，本来 C++で入出力するには cout (cerr)や cin という入出力ストリームというものを使うようにとなっています。これらはそれぞれ標準出力（標準エラー出力)，標準入力を示すストリームで，

```
cout << a << "文字" << endl; // 画面に「変数 a」，「文字」，「改行」を続けて表示する。
cin >> str;                  // キーボードからの入力を str という変数に格納する。
```

のように使います。また，この入出力ストリームを使うにはあらかじめ iostream.h を読み込んでおく必要があります。

さて，このストリームとは何でしょう。英語で stream とは「流れ」や「小川」という意味ですが，ここでは入出力の通り道を示していて，C++の中では ostream というクラスのオブジェクトとして記述されています。そして，中で使っている「<<」や「>>」はオペレータオーバーロードで定義された演算子です。

cin や cout は書式指定など余計なことをしなくても良い時にはとても便利です。

3.4.2.　ファイルストリーム

ファイルの入出力に関しても，cin や cout と同じようにストリームで管理することができます。ファイルに対するストリームは ofstream というクラスが，fstream.h の中で定義されています。

```
ofstream( );
ofstream( const char* szName, int nMode = ios::out, int nProt = filebuf::openprot );
ofstream( filedesc fd );
ofstream( filedesc fd, char* pch, int nLength );
```

コンストラクタがオーバーロードされています。最初のコンストラクタはオブジェクトを作成するだけです。2 番目のコンス

トラクタは，出力するファイル名を引数にしています。残りの引数はデフォルト引数で省略すると，出力用にオープン(ios::out)，デフォルトモード(ios::openprot)となります。後のコンストラクタはヘルプなどで確認してください。

逆にファイルの入力には，ifstream というのを使います。大体想像ができますね。

【練習問題 3.1】 クラス，デフォルト引数、オペレータオーバーロード

list3.3 の複素数クラス Complex を完成させ，四則演算を行うことができるようにしなさい。コンストラクタにデフォルト引数を設定することで，実数部のみの複素数の指定もできるようになります。

【練習問題 3.2】 new と delete

練習問題 2.2 で作成したプログラムを，読み込むデータ行数によって配列サイズを変更できるように書き換えなさい。

1) ファイルを読み込み行数をカウントする。
2) 読込みに必要な配列を確保する。
3) 確保した配列にデータを読み込む。

【練習問題 3.3】

http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/3/list8.zip

は，学生の氏名と英語，数学の点数を格納するクラスのサンプルです。Student.h，Student.cpp に CStudent クラスとその関数が定義されています。list8.cpp がメインとなっていますので，実際に実行して機能を確認してください。

また，StudentEx.h，StudentEx.cpp は CStudent クラスを基に化学の点数を格納できるようにした CStudentEx クラスのサンプルです。これは list8ex.cpp がメインとなっています。これらのソースから派生クラスの定義方法とその機能を理解してください。

【練習問題 3.4】

http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/3/list9.zip

は，線形リストと呼ばれるデータ構造のサンプルです。List.h，List.cpp が基本クラス，list9.cpp がそのメインとなっています。また，これに先頭要素と末尾要素の検索機能を派生クラスを用いて実装した例を SrchList.h，SrchList.cpp に，メニューを追加したメインを list9s.cpp としていれてあります。これらの定義方法とその機能を理解してください。

[線形リスト]

線形リストとは，要素を一方向に連結したデータ構造です。各要素は次の要素を示すポインタを持っています。配列は異なり，データの移動を伴わずに挿入や削除を行えるというメリットがあります。練習問題 3.4 で示した名前を格納する線形リストは図で表すと図 3.1 のようになります。それぞれ示したデータとポインタを持つ各要素をノードと呼びます。

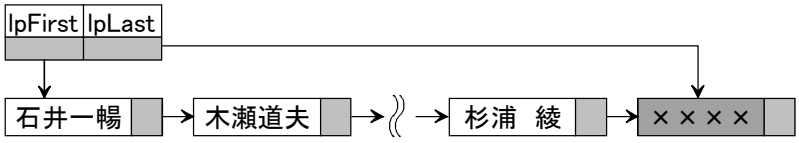


図 3.1　ダミーノードを用いた線形リストの例

線形リストを効率良く表現するために，先頭ノードと末尾ノードへのポインタを記憶する必要があります。また，末尾のノードはリストを管理するためのダミーのノードです。

コンストラクタが起動されるとダミーノード用の要素を1つ確保し，lpFirst と lpLast が共にその要素を示すように設定します。初期化された線形リストは図 3.2 のようになります。

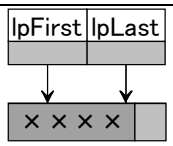


図 3.2　空の線形リスト

先頭への要素の挿入，末尾への要素の追加，先頭要素の削除，末尾要素の削除はそれぞれ図 3.3，3.4，3.5，3.6 のようにポインタの変更と，新規要素の作成（new）または要素の削除（delete）で実装されます。

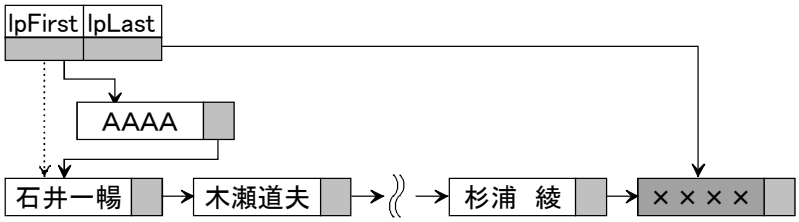


図 3.3　先頭への要素の挿入

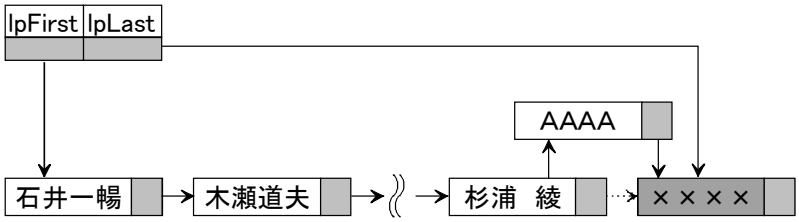


図 3.4　末尾への要素の追加

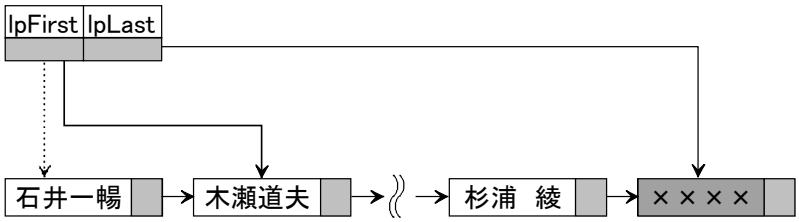


図 3.5　先頭要素の削除

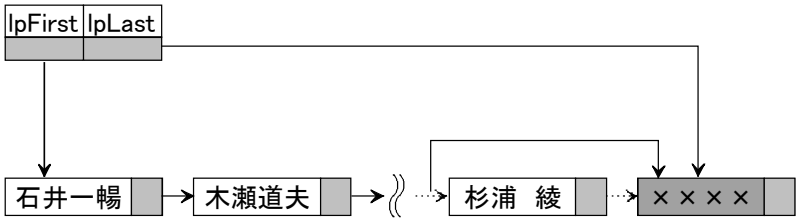


図 3.6　末尾要素の削除

4.　Windows アプリケーション(1)　Windows プログラムの基本，リソース

さて，今回からはWindowsアプリケーションの作成を始めます。Windowsのアプリケーションといと，ちょっと難しいように聞こえますが，実際はパターンが決まっていますので，それさえ押さえてしまえば，それほど苦労もなく作ることができます。また，実際に動作を見ながらプログラミングができますので，作ってみると意外と簡単ということもあります。

4.1.　コンソールアプリケーションとの違い

まず，Windows アプリケーションとコンソールアプリケーションの違いから，確認してみましょう。コンソールアプリケーションを作るためには，

1)　main 関数を書く。
2)　ユーザ定義関数とプロトタイプ宣言をする。
3)　必要なら，変数を定義してアルゴリズム（プログラムの流れ）通りにソースを書く。
4)　必要に応じてヘッダファイルをインクルードする。

といった感じでした。では，Windows アプリケーションではどうでしょうか？実は，基本的にはほとんど変わりません。ただし，

1)　main 関数の代わりに WinMain 関数を使う。
2)　アプリケーションに最低1つインスタンスが必要で，これを最初に作る必要がある。
3)　1つのウィンドウに1つのウィンドウハンドルと呼ばれる識別子が必要である。
4)　必要に応じて，リソーススクリプトを書く必要がある。

という違いがあります。

4.2.　Windows プログラム

4.2.1.　Windows API

API（Application Program Interface）とはOSがアプリケーションに提供する機能（関数）セットのことで，これまでハードウェアなどを操作するためにたくさんのプログラムを書かなくてはいけなかったものを，OS が肩代わりすることで必要最小限の操作で同様の機能を実現するためのものです。その実体はDLL（Dynamic Link Library）と呼ばれるもので，必要なときに読み込まれる（Dynamic Link）C 言語で書かれたライブラリです。これらの機能はアプリケーションだけでなくOSそのものでも数多く使用しているため，実際にはWindowsを使っている間に何度も目にしている機能が数多くあります。

WindowsアプリケーションはこのWindows APIを呼び出すことで作成できます。呼び出す方法は，直接プログラムに書き込むかMFC（Microsoft Foundation Class）というものを通して行います。MFCはWindows APIを機能ごとにまとめて極単純なコードでWindows APIを使用できるようにしています。そのため，Visual C++でもできるだけMFCを使うように推奨しています。しかし，MFC は汎用性を高めるために無駄な処理も多く行っているため処理が遅い，処理の流れが見えなくなるため Windows プログラムの仕組みが理解できないなどの欠点も持っています。ここでは，Windows アプリケーションの作り方をマスターするのが目的ですので，MFC については取り扱わないことにします。

【練習問題 4.1】 Windows API

VC++のヘルプから MessageBox 関数という API の機能を調べなさい。複数選択出きる場合は，「プラットフォーム SDK」の項目を参照すること。また，MFC の AfxMessageBox 関数も参照してその違いを比較しなさい。

4.2.2.　MessageBox だけを使った Windows プログラム

では MessageBox API だけを使った Windows プログラムを作ってみましょう。基本的な作り方はコンソールアプリケーションと同じです。ただし，今回はプロジェクトの種類を「Win32 Application」にします。ただ，メッセージを表示するだけではつまらないので，時刻に応じてあいさつを表示するプログラムを作ってみましょう。必要となる API は

MessageBox と GetLocalTime ですので，あらかじめヘルプを参照しておいてください。list4.1 にソースを示します。

```
#include    <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow){
    SYSTEMTIME          stSystemTime;
    int                 iRet;

    // 現在の時刻を取得します
    GetLocalTime(&stSystemTime);

    // 時刻に応じてメッセージを表示します
    if((stSystemTime.wHour > 0) && (stSystemTime.wHour < 12)){
        iRet = MessageBox(NULL, "おはようございます", "メッセージ", MB_OK);
    }else if(stSystemTime.wHour < 17){
        iRet = MessageBox(NULL, "こんにちは", "メッセージ", MB_OK);
    }else{
        iRet = MessageBox(NULL, "こんばんわ", "メッセージ", MB_OK);
    }

    return 1;
}
```

list4.1 MessageBox を使ったプログラム

ここで使った SYSTEMTIME は GetLocalTime 関数のパラメータに与える構造体で，MB_OK は MessageBox 関数のパラメータに与える定数です。これらは windows.h に定義されています。

4.2.3.　メッセージ

Windows の特徴である GUI(Graphical User Interface)とマルチタスクはメッセージというものを使うことで実現しています。メッセージとは，様々な動作に対して定義された番号で基本動作に関してはWindows によってあらかじめ決められています。プログラムの中では WM_・・・という名前で定義されており，これらをつかまえることで様々な動作を実現します。その動きを見るためには，Spy++というアプリケーションを使います。起動すると，図 4.1 のような画面が出てきます。では，メモ帳を動かしてその動きを見てみます。ウィンドウ1と表示された中には現在動いているアプリケーションの一覧が表示されます。その中から「003D0372“無題－メモ帳” Notepad」を選択してください(003D0372 のところは環境によって変化します)。この 003D0372 という数値はウィンドウを識別するための値でウィンドウハンドルといいます。「スパイ」－「メッセージ」を選択すると「メッセージオプション」というウィンドウがでてきますので，そのまま「OK」を押してください。すると「メッセージ(ウィンドウ 003D0372)」というウィンドウが開きます。ここにメモ帳に送られるメッセージが全て表示されます。「メモ帳」を動かしてその動作を確認してみてください。

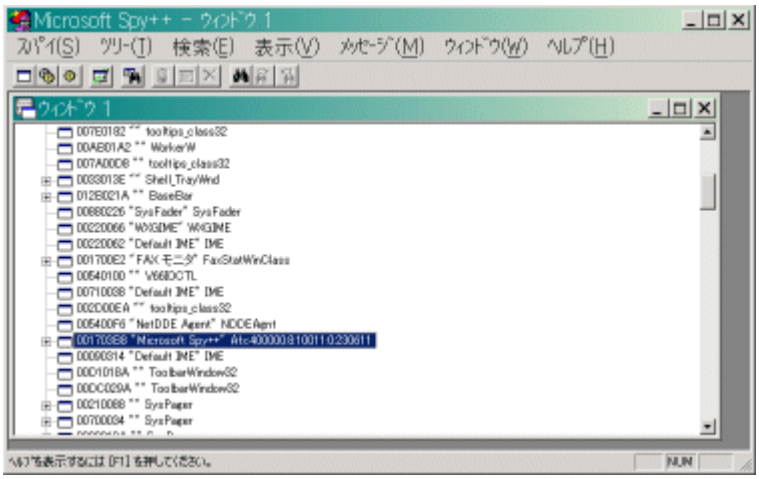


図 4.1　Spy++の起動画面

4.3.　基本的な Windows アプリケーション

では，メッセージを使った Windows アプリケーションの流れを見てみましょう。このプログラムソースはこれから作るプログラムの基本(テンプレート)になります。

4.3.1.　テンプレートとプログラムの流れ

list4.2(http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/4/)に示すソースの中を順に追っていくとこにしましょう。

1)　windows.h を読み込みます。これは，Windows 関連のいろいろな定義をしてあるヘッダなので，必ず読み込むようにしておきましょう。

2)　プリプロセッサ定義 WINNAME はこれから定義する WindowClass の名前になります。これから作るアプリケーショ

```
//////////////////////////////////////////////////////////////
// Win32 Application Template
//////////////////////////////////////////////////////////////
#include    <windows.h>

#define WINNAME "Template"          // ウィンドウクラスの名前

ATOM InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMes, WPARAM wp,
    LPARAM lp);
BOOL QuitMessage(HWND hWnd);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow){

    if(!InitApplication(hInstance))
        return FALSE;

    HWND                hWnd;       // メインウィンドウハンドル
    if(NULL == (hWnd = InitInstance(hInstance, nCmdShow)))
        return FALSE;

    // イベントキューからメッセージを取得する
    MSG                 msg;
    do{
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            if(msg.message == WM_QUIT)
                break;              // while()ループを抜ける
            TranslateMessage(&msg); // キー入力の取得
            DispatchMessage(&msg);  // メッセージを処理する
        }
        if(msg.message == WM_QUIT)
            break;                  // do～while()ループを抜ける
    }while(WaitMessage());

    return (int)msg.wParam;
}

ATOM InitApplication(HINSTANCE hInstance){
    WNDCLASSEX  wcl;

    // ウィンドウクラスを定義する
    wcl.hInstance = hInstance;      // インスタンスのハンドル
    wcl.lpszClassName = WINNAME;    // ウィンドウクラス名
    wcl.lpfnWndProc = WindowFunc;   // ウィンドウ関数
    wcl.style = 0;                  // デフォルトのスタイル
    wcl.cbSize = sizeof(WNDCLASSEX);// WNDCLASSEX構造体サイズ
    wcl.hIcon = NULL;               // ラージアイコン
    wcl.hIconSm = NULL;             // スモールアイコン
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
                                    // カーソルスタイル
    wcl.lpszMenuName = NULL;        // メニューなし
    wcl.cbClsExtra = 0;             // エキストラなし
    wcl.cbWndExtra = 0;             // 必要な情報なし

    // ウィンドウを白くする
    wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);

    // ウィンドウクラスを登録する
    return RegisterClassEx(&wcl);
}
```

```
HWND InitInstance(HINSTANCE hInstance, int nCmdShow){
    HWND        hWnd;

    // ウィンドウの作成
    hWnd = CreateWindow(
        WINNAME,                    // ウィンドウクラスの名前
        "Template",                 // タイトルバー
        WS_OVERLAPPEDWINDOW,        // ウィンドウスタイル
        CW_USEDEFAULT,              // x座標-Windowsに任せる
        CW_USEDEFAULT,              // y座標-Windowsに任せる
        CW_USEDEFAULT,              // 高さ-WIndowsに任せる
        CW_USEDEFAULT,              // 幅-Windowsに任せる
        HWND_DESKTOP,               // 親Windowなし
        NULL,                       // メニューなし
        hInstance,                  // インスタンスハンドル
        NULL                        // 追加引数なし
    );
    if(NULL == hWnd)
        return NULL;

    // ウィンドウを表示する
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return hWnd;
}

// この関数は，Windowsから呼び出されて，メッセージキューから
// メッセージの引き渡しを受ける
LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMsg, WPARAM wp,
                            LPARAM lp){
    static HINSTANCE        hInstance;

    switch(uMsg){
    case WM_CREATE:                 // ウィンドウの作成
        hInstance = (HINSTANCE)GetWindowLong(hWnd,
                                        GWL_HINSTANCE);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);         // WM_QUITを発行する
        break;
    case WM_CLOSE:                  // ウィンドウの終了処理
        if(QuitMessage(hWnd)){      // 終了の確認
            DestroyWindow(hWnd);    // メインウィンドウに
                                    // WM-DESTROYを発行する
        }
        break;
    default:                        // 上記以外はWindowsに
        return DefWindowProc(hWnd, uMsg, wp, lp);
    }
    return 0;
}

BOOL QuitMessage(HWND hWnd){
    if(MessageBox(hWnd, (LPCSTR)"終了しますか？",
        (LPCSTR)"終了確認",
        MB_YESNO | MB_ICONQUESTION) == IDYES)
        return TRUE;
    return FALSE;
}
```

list4.2　Windows アプリケーションテンプレート

ンの基本的な名前にもなります。

3)　4つのプロトタイプ宣言をしていますが，最初の3つは Windows アプリケーションの基本的な動作を決める関数です。毎回ほとんど変わらないので，これらを使いまわすことで，ソースを書く効率がよくなります。最後の関数は，終了するときにいきなり終了しないようにするための関数です。これも毎回入れておくと便利でしょう。

4)　WinMain 関数です。ここで定義している「HWND hWnd」がこのアプリケーションのメインウィンドウハンドルです。まず InitApplication 関数でこれから作成するウィンドウの定義を行い，InitInstance 関数で hWnd を取得します。ついでに取得できなかったら，終了するという処理もしています。メインウィンドウに与えられるメッセージは「MSG msg」と定義し，以下の do～while 文と while 分で2重の無限ループで処理します。PeekMessage というのは Windows メッセージを取得する関数で，メッセージを受け取ると TRUE を返します。もし，受け取ったメッセージが WM_QUIT というメッセージなら2回 break して無限ループを終了，つまりアプリケーションを終了します。メッセージが WM_QUIT でない場合は，内側のループで囲まれた処理をして，WaitMessage()関数のところで次のメッセージが来るまで停止しています。

5)　InitApplication 関数では，このアプリケーションでメッセージを処理する関数（WindowsFunc），アイコン，カーソル，メニュー，ウィンドウの色などを szWinName で定義した名前でウィンドウクラスというクラスに定義しています。

6)　InitInstance 関数は，定義されたウィンドウクラスに基づいてウィンドウを作成します。ShowWindow 関数はここで

作成されたウィンドウを表示します。さらに UpdateWindow 関数を呼び出しているのは，WindowFunc 関数で Window が作成されたときにする処理を画面に反映するためです。

7)　さて，いよいよ WindowFunc 関数です。この関数がプログラムの中心になります。まず，Instance を static で定義しています。これは，Instance を必要とする WindowsAPI が結構たくさんあるためです。実際の取得はこのアプリケーションウィンドウが作成されたとき，すなわち WM_CREATE で行います。次に，switch 文が来ます。WindowFunc 関数は PeekMessage 関数で TRUE が返ってきたとき，すなわちこのアプリケーションに何らかのメッセージが来たときに毎回呼び出されます。どのようなメッセージが与えられたかは uMsg に格納されています。アプリケーションで操作に合わせた処理をするためには uMsg によって適切な処理を行えば良い，ということになります。

例えば，このウィンドウが作成された時には WM_CREATE メッセージが通達されます。アプリケーションはこのとき，Window 内に文字を書く必要があれば，この段階で書くことができますし，メモ帳などのような文字入力部分を作成したいのであればこの段階で作成することになります。ここでは GetWindowLong 関数を使って hInstance の取得を行っています。

WM_CLOSE と WM_DESTROY は終了するときの処理です。ウィンドウの右上の×印が押されるとウィンドウに WM_CLOSE メッセージが通知されます。そこで，WindowFunc では WM_CLOSE が来たら終了して良いかの確認を行い（QuitMessage 関数），TRUE が来たら hWnd を破棄（DestroyWindow，つまり WM_DESTROY の送信）を行っています。WM_DESTROY はこのウィンドウが閉じる時の処理で，PostQuitMessage は WM_QUIT を送れという意味です。この WM_QUIT を WinMain が受け取ると，WinMain の無限ループが終了し，プログラムが終了するということです。

また，最後の DefWindowProc 関数は自分で処理する必要がないものは Windows に任せるという関数です。

8)　QuitMessage 関数は，終了して良いかどうかの確認をするための関数です。MessageBox の引数の hWnd は親 Window ハンドルで，このメッセージに応えるまで親ウィンドウを選択できないようにします。MessageBox では YES が押されたら IDYES が返ってくるので TRUE を返し，それ以外（この場合は NO だけですが）は FALSE を返すということです。ですから，この関数の戻値は TRUE（真）と FALSE（偽）を返す型 BOOL という具合になっています。

4.3.2.　Windows での型と変数名

list4.2 を見ていて，変数の名前と型が変だなと思った人もいるでしょう。実は Windows では変数の型が少し拡張されています。また，変数名についてもその変数の型がわかりやすいように型の識別名がつくようになっています。決まりという訳ではないのでこだわる必要はないですが，Help をみる手助けにもなるでしょうから，簡単に触れておきましょう。

表 4.1　Windows での型拡張

型	表記	意味	表記	意味
文字	CHAR	符号あり 8bit 文字	LPSTR	文字列定数
	UCHAR	符号なし 8bit 文字	LPCSTR	文字列
2 値	BOOL	TRUE と FALSE		
整数	SHORT	16bit 符号あり整数	ULONG	32bit 符号なし整数
	USHORT	16bit 符号なし整数	BYTE	8bit 符号なし整数
	INT	符号あり整数	WORD	16bit 符号なし整数
	UINT	符号なし整数	DWORD	32bit 符号なし整数
	LONG	32bit 符号あり整数	LONGLONG	64bit 符号なし整数
Windows	WINAPI	Win32 API	LRESULT	メッセージ処理の戻値
	WPARAM	32bit メッセージ変数	WNDPROC	Window 関数へのポインタ
	LPARAM	32bit メッセージ変数	VOID	任意の変数型

ハンドル系	HANDLE	オブジェクト	HCURSOR	カーソル
	HINSTANCE	インスタンス	HFILE	ファイル
	HWND	ウィンドウ	HICON	アイコン
	HACCEL	アクセラレータ	HMENU	メニュー
	HBRUSH	ブラシ		
ポインタ	LPBOOL	BOOL へのポインタ	LPDWORD	DWORD へのポインタ
	LPINT	INT へのポインタ	LPLONGLONG	LONGLONG へのポインタ
	LPLONG	LONG へのポインタ	LPVOID	VOID へのポインタ
	LPBYTE	BYTE へのポインタ	LPHANDLE	HANDLE へのポインタ
	LPWORD	WORD へのポインタ		

変数名は例えば，WORD 型なら w･･･，DWORD 型なら dw･･･，ハンドル型なら h･･･，ポインタ型なら lp･･･，文字列型なら lpsz･･･といった感じです。

4.4.　リソース

Windows プログラムの特徴の一つにリソースがあります。リソースとはプログラムとは独立したデータのことで，アイコン，カーソル，文字列，ビットマップ，メニュー，キーボードアクセラレータ，ダイアログボックスなどがこれにあたります。

4.4.1.　アイコンとカーソル

それではプログラムにアイコンとカーソルを追加してみましょう。リソースを書くにはリソースエディタを使います。

1)　まず，テンプレートを基にワークスペースを作ります。

2)　「ファイル」－「新規作成」から新規作成ウィンドウを表示します。「リソーススクリプト」をクリックして選択し，ファイル名を書き込んでから「OK」を押します。すると，FileView の Source Files フォルダの中に，･･･.rc というファイルができているのが確認できます。また，ワークスペースウィンドウに FileView，ClassView の他に ResourceView というのが追加されます。

3)　では，ResouceView に切り替えて新しいリソースを作ってみます（アウトプットウィンドウは閉じておいてください）。フォルダの形をしたアイコンで右クリックしてポップアップメニューから「挿入」を選択します。リソースの挿入ダイアログが出てくるので，「Icon」を選択して「新規作成」ボタンをクリックします。作業領域にアイコンを作成するためのリソースエディタが表示されますので，適当なアイコンを作成してみてください。

　作成できたらワークスペースウィンドウを見ると IDI_ICON1 という名前でアイコンができていることがわかります。この識別子（ID）を使ってプログラムからこのアイコンを呼び出すことになります。

4)　同様にしてカーソルも作成してみましょう。アイコンと異なることは，白黒といくつかのグレーしか使用できないことと，ホットスポット（クリック位置を示すもの）を指定できることです。ホットスポットの指定は上の方にある を押すと指定できます。今度は IDC_CURSOR1 という ID で作成できています。

5)　ここで，「ファイル」－「すべて保存」を選択してから，ファイルのあるフォルダを見ると，今作ったアイコン，カーソルと resource.h というファイルができています。このファイルを開いてみると，

```
#define   IDI_ICON1        101
#define   IDC_CURSOR1      102
```

と書かれているはずです。このようにリソースは実際にはこのような数字で識別されています。

6)　さて，実際のプログラムにこれを反映するには元のプログラムに多少の修正が必要です。

①　resource.h を include する。

②　InitApplication 関数で，

```
wcl.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON1));
```

```
wcl.hIconSm = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON1));
wcl.hCursor = LoadCursor(hInstance, MAKEINTRESOURCE(IDC_CURSOR1));
```

を追加します。

7)　変更が終わったら実際にビルド，実行して変更を確認してみましょう。

4.4.2.　メニュー

Windows のプログラムで使用されるメニューには上のほうに表示される通常のメニューとマウスの右ボタンを押したときに表示されるポップアップメニューがあります。ここでは使用頻度の高い通常のメニューについてのみ説明します。

メニューはアイコンやカーソルと同じように ID が付けられるだけではなく，メニュー内の個々の項目にも ID が付けられます。このIDはWM_COMMANDメッセージを使ってアプリケーションに通知されます。どのメニューが選択されたかはWM_COMMANDを受け取ったときのWindowFuncの引数wpの下位ワード（32bit 整数の下位 16bit）に格納されます。では，「ファイル」－「終了」という項目を持つメニューをテンプレートに付けてみましょう。

1)　まず，テンプレートを基にワークスペースを作成して，リソーススクリプトを追加しておきます。

2)　今回はリソースの追加で「Menu」を選択します。すると，IDR_MENU1 という ID が追加されます。

3)　破線で表示された四角形をダブルクリックすると，メニューアイテムプロパティというウィンドウが表示されます。ここでメニューに関する設定を行います。キャプション欄に「ファイル(&F)」と入力してください。アンパサンド（&）をアルファベットや数字の前に付けるとメニューの中では「ファイル(F)」のようにアンダーライン付きで表示されます。このようにしておくとキーボードの「Alt」と「F」を同時に押すことで，このメニューを選択できるようになります。

4)　今度は下に表示された四角をダブルクリックして，今と同じように「終了(&X)」を作ります。ここで終了すると，このメニューアイテムには自動的に ID_MENUITEM40001 という ID がつきます。このままでもプログラム上はさしつかえないですが，あとでわかりやすいように自分で ID を決めておきましょう。今回は IDM_QUIT という名前にしておきます。ID の欄に「IDM_QUIT」と入力してください。

5)　さて，プログラムにこれを反映させましょう。

①　前回と同様に resource.h を include する。

②　InitApplication 関数で，

```
wcl.lpszMenuName = MAKEINTRESOURCE(IDR_MENU1);
```

を追加します。

③　WIndowFunc に，

```
case WM_COMMAND:
    switch(LOWORD(wp)){
    case IDM_QUIT:
        PostMessage(hWnd, WM_CLOSE, 0, 0L);
        break;
    }
    break;
```

を追加します。

6)　変更が終わったら実際にビルド，実行して変更を確認してみましょう。

4.4.3.　ダイアログボックス

多くの Windows プログラムではメインウィンドウの他に入力や設定を行うためのダイアログボックスを使います。ダイアログボックスには2種類あり，それぞれモーダルダイアログとモードレスダイアログと呼ばれます。モーダルダイアログとはファイルを開くときに出てくるダイアログボックスのように OK や Cancel などを押さない限り他の処理ができないものです。一方，モードレスダイアログとは検索ダイアログのように検索中も文章の編集ができるように，他のウィンドウと並行

して動作するダイアログボックスです。ここでは，モーダルダイアログについてのみ触れます。

ダイアログボックスを作るためには DialogBox 関数を，閉じるには EndDialog 関数を使います。また，ダイアログボックス自身やその上に表示されるコントロール（第5回で解説）の処理はウィンドウプローシージャ（WindowsFunc）と同様のダイアログプロシージャで処理します。

まだ，コントロールの説明をしていませんので，今回はただ文字（バージョン情報）を表示するだけのダイアログを表示してみましょう。

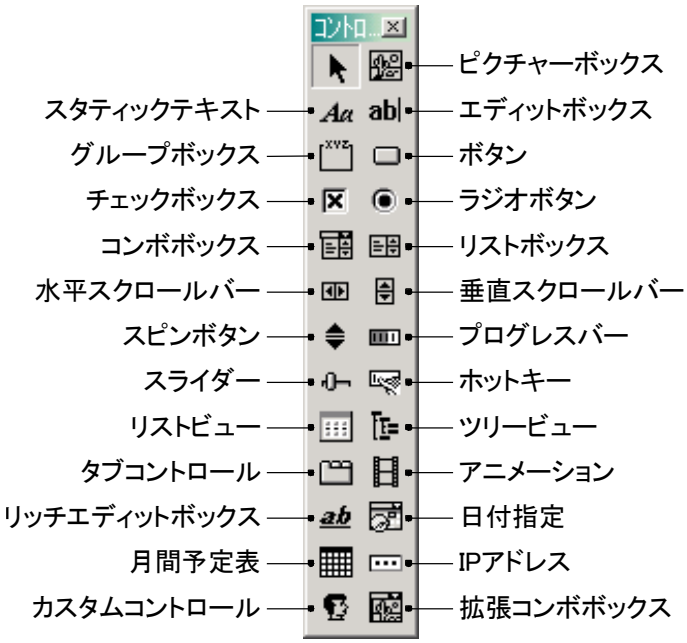


図 4.2　コントロールボックス

1)　まず，テンプレートを基にワークスペースを作成して，リソーススクリプトを追加しておきます。

2)　前回と同様にメニューを作成しておきます。今回はメニュー項目を「ファイル(&F)」－「終了(X)」（IDM_QUIT）と「ファイル(&F)」－「バージョン(&A)」（IDM_ABOUT）の2つにしておきます。

3)　リソースの追加で「Dialog」を選択します。すると，IDD_DIALOG1 という ID が追加されます。また，エディタ画面に図 4.2 のようなコントロールボックスが表示されます。それぞれのボタンは様々なコントロールを示しています。今回はスタティックテキストコントロールだけを使います。ダイアログにはあらかじめ「OK」と「キャンセル」の2つのボタンが張りつけられています。これらにはそれぞれ「IDOK」，「IDCANCEL」の ID があらかじめ付けられています。今回はこれ以外にスタティックテキストで「ダイアログのテスト Ver.1」という文字も追加したりしながら自由に加工してみてください（図 4.3）。

図 4.3　ダイアログボックス

4)　さて，プログラムにこれを反映させましょう。今回は長いので list4.3 に示しました。

```
BOOL QuitMessage(HWND hWnd);
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp,
                         LPARAM lp);
                         ⋮
LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMsg, WPARAM
wp, LPARAM lp){
                         ⋮
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDM_QUIT:
            PostMessage(hWnd, WM_CLOSE, 0, 0L);;
            break;
        case IDM_ABOUT:
            DialogBox(hInstance,
                      MAKEINTRESOURCE(IDD_DIALOG1),
                      hWnd,
                      (DLGPROC)DialogProc);
            break;
        }
        break;
```

```
                         ⋮
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp,
                         LPARAM lp){
    switch(uMes){
    case WM_INITDIALOG:
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            EndDialog(hDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return TRUE;
        }
        break;
    }
    return FALSE;
}
```

list4.3　ダイアログの表示

最初のは今回追加するダイアログプロシージャのプロトタイプ宣言です。戻値が BOOL になっている以外はウィンドウプロシージャと同じです。

次にこのダイアログを出すメニュー項目（IDM_ABOUT）を追加します。これは前回と方法は同じです。この中で

ダイアログボックスを呼び出すことになります。モーダルダイアログボックスを呼び出すにはDialogBox関数を使います。引数は，インスタンスハンドル（hInstance），ダイアログリソース，呼びだし側ウィンドウハンドル(hWnd)，ダイアログプロシージャ名となります。

最後にダイアログプロシージャ本体を追加します。ほとんどウィンドウプロシージャと同じです。ダイアログボックスも一つのウィンドウになるわけですから，ダイアログを作成するとプログラム本体とは別のウィンドウハンドル hDlg が1つ作成されます。

ウィンドウプロシージャと異なる点をまとめると，

① ウィンドウが作成されたときに来るメッセージは WM_INITDIALOG となる。

② DefWindowProc が必要ない。その代わり，処理をしたときは TRUE を，しなかったときは FALSE を返す。

③ ダイアログを閉じるには EndDialog を呼び出す。

というところです。

4.5. ダイアログベースアプリケーション

ダイアログボックスはリソースエディタを使うことで容易にインタフェースを作ることができます。そこで，4.3 で説明した WinMain の最初でモードレスダイアログを作成することで，インターフェースの構築が容易なダイアログベースアプリケーションを作ることができます。list4.4（http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/4/DialogBase.zip，VS6 用は DialogBase6.zip）のソース。基本的な流れは list 4.2 と同じで，①ウィンドウクラスの登録とインスタンスの生成を CreateDialog だけで行う。②WindowFunc の代わりに DialogProc を用いる。③ウィンドウクラスで登録するようなリソースは WM_INITDIALOG で登録する。というところが大きな違いとなります。

```
//=====================================================
// ダイアログボックスベースのアプリケーション
// === モードレスダイアログ版 ===
//=====================================================
#include    <windows.h>
#include    <windowsx.h>
#include    "resource.h"

BOOL CALLBACK DialogProc(HWND hWnd, UINT uMes, WPARAM wp,
LPARAM lp);
BOOL QuitMessage(HWND hDlg);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow){
    // モードレスダイアログボックスを生成する
    // ダイアログボックスが生成されると処理が戻ります
    static HWND         hDlg;
    hDlg = CreateDialog(hInstance,
        MAKEINTRESOURCE(IDD_DIALOG), NULL,
        (DLGPROC)DialogProc);

    // アクセラレータのロード
//  HACCEL              hAccel; // アクセラレータハンドル
//  hAccel = LoadAccelerators(hInstance,
        MAKEINTATOM(IDR_ACCELERATOR));

    // メッセージループを生成する
    MSG                 msg;
    do{
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            if(msg.message == WM_QUIT)
                break;      //while()ループを抜ける
//          if(TranslateAccelerator(hWnd, hAccel, &msg))
//              continue;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        if(msg.message == WM_QUIT)
            break;          // do～while()ループを抜ける
    }while(WaitMessage());

    return msg.wParam;
}
```

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMsg, WPARAM wp,
                         LPARAM lp){
    static HINSTANCE    hInstance;

    switch(uMsg){
    case WM_INITDIALOG:         // ダイアログの生成
        hInstance = (HINSTANCE)GetWindowLong(hDlg,
            GWL_HINSTANCE);
        SetClassLong(hDlg,      // アイコンの設定
            GCL_HICON, (LONG)LoadIcon(hInstance,
            MAKEINTRESOURCE(IDI_ICON)));
        ShowWindow(hDlg, SW_SHOW);  // ウィンドウの表示
        UpdateWindow(hDlg);
        return TRUE;

    case WM_CLOSE:              // ×を押したとき
        if(QuitMessage(hDlg)){  // 終了確認
            DestroyWindow(hDlg);// ダイアログの破棄
            PostQuitMessage(0); // WM_QUITを送る
            return TRUE;
        }
        return TRUE;

    // メニューから呼び出されるメッセージ
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDM_QUIT:          // [ファイル]－[終了]
            PostMessage(hDlg, WM_CLOSE, 0, 0L);
            return TRUE;
        }
    }
    return FALSE;               // 何もしないときはFALSE
}

BOOL QuitMessage(HWND hDlg){
    if(MessageBox(hDlg, "終了してもよろしいですか。",
        "終了確認",MB_YESNO | MB_ICONQUESTION) == IDYES)
        return TRUE;
    return FALSE;
}
```

list4.4 ダイアログベースアプリケーション

5.　Windows アプリケーション(2)　コントロールの実装

5.1.　コントロールの種類と役割

Whidows アプリケーションを使いやすくするための機能としてボタンや入力ボックスがあります。このような様々なアプリケーションで共通に使うことのできる Windows が提供する機能をコントロールと呼びます。Windows が提供するコントロールは Windows のバージョンによって若干異なります。表 5.1 に基本的なコントロールを，図 5.1 にそれらの概観を示します。

表 5.1　基本的なコントロールの種類

クラス名（コントロール）	機能
BUTTON	ボタン，チェックボックス，ラジオボタン，グループボックス
COMBOBOX	コンボボックス
EDIT	エディットボックス
LISTBOX	リストボックス
SCROLLBAR	スクロールバー
STATIC	文字列，ビットマップ画像

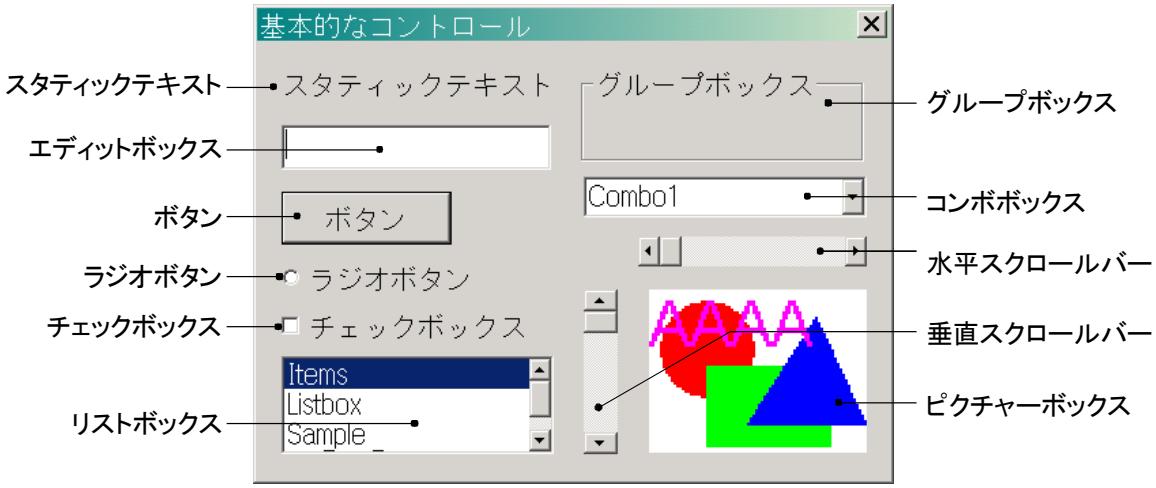


図 5.1　基本的なコントロール

このようなコントロールは基本的にアプリケーションウィンドウやダイアログボックスウィンドウ内の小さなウィンドウ（チャイルドウィンドウ）として動作します。すなわち，これらを作るときはウィンドウクラスを定義（RegisterClass）して，ウィンドウを作成（CreateWindow）するという手順が必要となります。ただし，このように頻繁に使用するクラス全てに対してクラス定義をするというのはとても不便です。そこで，Visual C++ではこのような基本コントロールのクラスをあらかじめ定義してあります（表 5.1 の1列目）。この定義済みクラス名を CreateWindow の引数とすることで簡単にこれらのコントロールを使用できます。また，ダイアログボックスではリソースエディタが使えますので，あらためて CreateWindow を呼ばなくてもリソースエディタでコントロールを貼り付けることができます。それでは，扱いの簡単なダイアログボックスから各コントロールの使い方をマスターしてみましょう。

5.1.1.　スタティックテキスト，ボタン，エディットボックス

図 5.2 のようなダイアログを使ってスタティックテキスト，ボタン，エディットボックスの使い方を見てみましょう。上の白い四角がエディットボックス（IDC_EDIT1），Static と書いてあるところがスタティックテキスト（IDC_TEXT），それと下に2つボタンがあります。今回のプログラムではエディットボックスに入力した文字列を，「表示の更新」ボタンを押すと下のスタティックテキストに表示し，「閉じる」を押すとダイアログボックスを閉じる，ということにします。ス

図 5.2　スタティックテキスト，ボタン，エディットボックス

タティックテキストとは単なる文字表示で，ダイアログ内の文字の表示にも使用しています。ID は自動的に IDC_STATIC がつきます。今回はこのテキストにも操作を加えますので，ID を IDC_TEXT に変更します。また，下の2つのボタンはそれぞれ「OK」と「キャンセル」の表示を変更したもので，それぞれの ID は IDOK と IDCANCEL のままにしておきます。

ソースは前回のバージョンの表示に使用したものが利用できますので，基本的な部分は省略して，重要なダイアログプロシージャだけ示します。

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp, LPARAM lp){
    char              lpszMessage[128] = "";

    switch(uMes){
    case WM_INITDIALOG:
        SetDlgItemText(hDlg, IDC_TEXT, lpszMessage);
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            GetDlgItemText(hDlg, IDC_EDIT1, lpszMessage, 128);
            SetDlgItemText(hDlg, IDC_TEXT, lpszMessage);
            break;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return TRUE;
        }
        break;
    }
    return FALSE;
}
```

list5.1　スタティックテキスト，ボタン，エディットボックス

まず，エディットボックスに入力される文字列のための配列 lpszMessage を 128 字分確保しておき，空の文字列で初期化しておきます。次に，ダイアログが表示されたときに WM_INITDIALOG を受け取るので，スタティックテキストに lpszMessage の内容（最初は空ですが）を表示しておきます。表示には SetDlgItemText 関数を使います。

```
BOOL SetDlgItemText(
    HWND hDlg,          // ダイアログボックスハンドル
    int nIDDlgItem,     // コントロールの ID
    LPCTSTR lpString    // 設定する文字列へのポインタ
);
```

「表示の更新」ボタン（IDOK）ボタンが押されるとメッセージ WM_COMMAND の下位ワードに IDOK が来るので，エディットボックスの文字列を lpszMessage に格納します。エディットボックスの文字列を受け取るには GetDlgItemText を使います。

```
UINT GetDlgItemText(
    HWND hDlg,          // ダイアログボックスハンドル
    int nIDDlgItem,     // コントロールの ID
    LPTSTR lpString,    // 文字列を受け取るバッファへのポインタ
    int nMaxCount       // 文字列の最大文字数
);
```

受け取った文字列 lpszMessage はまた SetDlgItemText を用いてスタティックテキストに表示することができます。

【練習問題　5.1】 SetDlgItemInt，GetDlgItemInt

SetDlgItemText，GetDlgItemText と同様に整数を扱う SetDlgItemInt，GetDlgItemInt があるので，これを使って整数を入力して，整数を表示するダイアログ関数を作成せよ。また，同様に小数の場合はどうすと良いか考えよ。

5.1.2.　ラジオボタン，グループボックス

図 5.3 のようなダイアログを使ってラジオボタンとグループボックスの使い方を見てみましょう。4つの丸いボタンがラジオボタン，性別，学年の枠組みがグループボックスです。ラジオボタンはいくつかのグループ分けがあって，そのグループ内では1つしか選択できないボタンのことです。グループボックスはこのようなときにグループ分けを見やすくするための枠組みで，機能として

図 5.3　ラジオボタン，グループボックス

図 5.4　タブオーダー

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp, LPARAM lp){
    UINT                uSex = BST_CHECKED;
    UINT                uGrade = BST_CHECKED;

    switch(uMes){
    case WM_INITDIALOG:
        CheckDlgButton(hDlg, IDC_RADIO_MALE, uSex);
        CheckDlgButton(hDlg, IDC_RADIO_M1, uGrade);
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            uSex = IsDlgButtonChecked(hDlg, IDC_RADIO_MALE);
            uGrade = IsDlgButtonChecked(hDlg, IDC_RADIO_M1);
            switch(uSex * 2 + uGrade){
            case 3:
                MessageBox(hDlg, "男とM1が押されました．", "確認", MB_OK);
                break;
            case 2:
                MessageBox(hDlg, "男とM2が押されました．", "確認", MB_OK);
                break;
            case 1:
                MessageBox(hDlg, "女とM1が押されました．", "確認", MB_OK);
                break;
            case 0:
                MessageBox(hDlg, "女とM2が押されました．", "確認", MB_OK);
                break;
            }
            EndDialog(hDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return FALSE;
        }
        break;
    }
    return FALSE;
}
```

list5.2　ラジオボタン，グループボックス

は何もありません。さて，それではリソースエディタでダイアログを作成してみましょう。それぞれのラジオボタンにはそれぞれ，IDC_RADIO_MALE，IDC_RADIO_FEMAIL，IDC_RADIO_M1，IDC_RADIO_M2 と割り当てることにします。次にグループ分けの設定をします。ラジオボタンのグループ分けをするためにはまずグループの先頭を決める必要があります。今回は性別と学年の2つのグループがあり，それぞれ「男」と「女」，「M1」と「M2」がありますので，「男」と「M1」をグループの先頭とします。先頭を決めたらそれぞれのコントロールのプロパティで「グループ」のチェックボックスをチェックしておきます。次にタブオーダーの設定をします。タブオーダーとはダイアログ内でタブキーを押したときにアクティブになるコントロールの順番です。リソースエディタで「レイアウト」－「タブオーダー」を選択すると，図 5.4 のようにそれぞれのコントロールに数字が付けられます。この数字がタブオーダーになります。ラジオボタンのグループ分けは「グループ」のチェックボックスがチェックされたラジオボタンからタブオーダーが連続するラジオボタンがグループであると判断します。今回は図のようにタブオーダーを指定してください。タブオーダーの指定は順番にクリックしていくと変化します。

さて，それではプログラムに移りましょう。それぞれのラジオボタンが押されているかを調べるのは IsDlgButtonChecked 関数です。

```
UINT IsDlgButtonChecked(
    HWND hDlg,          // ダイアログボックスハンドル
    int nIDButton       // ボタンの ID
);
```

ボタンが押されていれば BST_CHECKED，押されていなければ BST_UNCHECKED が戻値として返ってきます。それぞれ1，0と定義されています。また，それぞれのラジオボタンのチェック状態を変更するには CheckDlgButton 関数を使います。

```
BOOL CheckDlgButton(
    HWND hDlg,          // ダイアログハンドル
    int nIDButton,      // ボタンの ID
    UINT uCheck         // ボタンの状態
);
```

uCheck に BST_CHECKED や BST_UNCHECKED を設定することで，ボタンの状態を変更できます。

それでは，「OK」を押すとチェックボックスをチェックしてメッセージボックスを表示するプログラムを作成してみましょう。今回もダイアログプロシージャのみ list5.2 に示しておきました。

まず，性別と学年のための変数 uSex と uGrade を定義し，それぞれを BST_CHECKED に設定しておきます。WM_INITDIALOG を受け取ったら，IDC_RADIO_MALE と IDC_RADIO_M1 のチェックボタンを uSex，uGrade の値にセットしておきます。IDOK が押されたら，IDC_RADIO_MALE と IDC_RADIO_M1 のボタンの状態を取得します。それぞれのグループで1回しかチェックしないのはラジオボタンがそれぞれのグループで2つしかないので，一方がチェックされていたらもう一方はチェックされていないからです。3つ以上ある場合は少し複雑になります。チェックが終わったらそれらの状態によって表示を変えて MessageBox を表示します。今回は BST_CHECKED が1であることを利用して，uSex×2＋uGrade を計算して switch 文で分岐しています。ラジオボタンの数が増えてきたらその判断方法を工夫する必要が出てきます。これはそのうちの一つのテクニックです。

5.1.3.　チェックボックス

図 5.4 のようなダイアログを使ってチェックボックスの使い方を見てみましょう。チェックボックスの使い方はグループ分けのないラジオボタンと考えることができます。使用する関数も IsDlgButtonChecked 関数，CheckDlgButton 関数と全くおなじです。

図 5.4　チェックボックス

【練習問題 5.2】 チェックボックス

図 5.4 に示したダイアログボックスを表示し，OK を押したときにチェックされているチェックボックスの値の合計をメッセージボックスで表示するプログラムを作成せよ。

5.1.4.　リストボックス

図 5.5 のようなダイアログボックスを作ってリストボックスの使い方を見てみましょう。上の四角い部分がリストボックス(IDC_LIST1)です。リストボックスはこのように選択できるリストを表示してその中から項目を選択するのに使用します。

図 5.5　リストボックス

それでは，「OK」を押したら選択された番号を表示してダイアログを閉じるプログラムを作成しましょう。これまでと同じようにテンプレートを基に作成します。今回は sprintf を使いますので，stdio.h も include しておいてください。それでは list5.3 にダイアログプロシージャを示します。今回は SendMessage 関数を使用しています。

```
LRESULT SendMessage(
        HWND hWnd,
        // ウィンドウハンドル
        UINT Msg,
        // 送るメッセージ
        WPARAM wParam,
        // メッセージパラメータ
```

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp, LPARAM lp){
    int                 i;
    int                 iListNo = 0;
    LPCTSTR             lpszList[] = {"LIST0", "LIST1", "LIST2", "LIST3"};
    char                lpszMessage[64];

    switch(uMes){
    case WM_INITDIALOG:
        for(i = 0; i <= 4; i++ )
            SendMessage(GetDlgItem(hDlg, IDC_LIST1), LB_INSERTSTRING,
                (WPARAM)i, (LPARAM)lpszList[i]);
        SendMessage(GetDlgItem(hDlg, IDC_LIST1), LB_SETCURSEL,
            (WPARAM)iListNo, 0L);
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            iListNo = (int)(DWORD)SendMessage(GetDlgItem(hDlg, IDC_LIST1),
                LB_GETCURSEL, 0L, 0L);
            sprintf(lpszMessage, "%d番が選択されました。", iListNo);
            MessageBox(hDlg, lpszMessage, "確認", MB_OK);
            EndDialog(hDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return FALSE;
        }
        break;
    }
    return FALSE;
}
```

list5.3　リストボックス

```
        LPARAM lParam
        // メッセージパラメータ
    );
```

hWnd にはそのメッセージを送りたいウィンドウのハンドルを指定します。Msg には送るメッセージを指定します。wParam と lParam にはそれぞれメッセージパラメータを指定します。この SendMessage 関数の引数はちょうどウィンドウプロシージャやダイアログプロシージャの引数と同じようになっています。つまり，この関数を使うことでそれぞれのウィンドウハンドルに割り当てられたプロシージャに自由にメッセージを送ることができるということです。ですから，この関数は様々なところで使用されます。今回はこの関数を使ってリストボックスコントロールのウィンドウハンドル（GetDlgItem 関数で取得）に割り当てられたプロシージャ（実際には見えない）にメッセージを送ることでリストボックスの操作を行います。

まず，リストの番号を格納する整数 iListNo とリストを格納する文字列配列 lpszList を定義します。WM_INITDIALOG を捕まえたらリストボックスに文字列を挿入していきます。

SendMessage(リストボックスウィンドウハンドル, LB_INSERTSTRING, (WPARAM)何番目か, (LPARAM)文字列);

次に初期状態でリストボックスのどの文字列が選択された状態にするかを指定します。

SendMessage(リストボックスウィンドウハンドル, LB_SETCURSEL, (WPARAM)何番目か, 0L);

「OK」ボタンが押されると（IDOK），

iListNo = (int)(DWORD)SendMessage(リストボックスウィンドウハンドル, LB_GETCURSEL, 0L, 0L);

で，上から何番目の文字列が選択されたかを取得しています。あとは，sprintf で文字列に格納して MessageBox で表示しています。

5.1.5.　コンボボックス

コンボボックスはリストボックスの表示を1行にまとめたような形をしています。機能もほとんどリストボックスと同じです。また，使用方法，使用する関数も同じです。

【練習問題 5.3】 コンボボックス

リストボックスで作成したダイアログボックスをコンボボックスを使って作成しなさい。

5.2.　エディットコントロール

今までのプログラムはメインウィンドウを使いませんでした。これではメインウィンドウの意味がほとんどありません。そこで，今度はメインウィンドウにコントロールを貼り付けてみます（貼り付けたものを子ウィンドウと言います）。CreateWindow 関数を使うことになりますので，その使い方を確認しておきましょう。

```
    HWND CreateWindow(
        LPCTSTR lpClassName,        // クラス名へのポインタ
        LPCTSTR lpWindowName,       // ウィンドウ名へのポインタ
        DWORD dwStyle,              // ウィンドウのスタイル
        int x,                      // ウィンドウの x 座標（水平方向）
        int y,                      // ウィンドウの y 座標（垂直方向）
        int nWidth,                 // ウィンドウの幅
        int nHeight,                // ウィンドウの高さ
        HWND hWndParent,            // 親ウィンドウのウィンドウハンドル
        HMENU hMenu,                // メニューハンドル，子ウィンドウの場合はウィンドウ ID
        HANDLE hInstance,           // インスタンスハンドル
        LPVOID lpParam              // WM_CREATE の lParam
    );
```

lpClassName:　親ウィンドウを作るときはRegisterClass関数で登録したクラス名でしたが，コントロールを作るときは表 5.1 に示した定義済みコントロール名を指定します。

lpWindowName:　タイトルバーに表示する名前です。子ウィンドウの場合は指定しないことが多いです。

dwStyle:　ウィンドウのスタイルを指定します。コントロールによって指定できる値が異なります。複数のスタイルを指定する場合は，「|」で連結します。Help を参照してください。

x, y, nWidth, nHeight:　親ウィンドウの時はスクリーン座標系ですが，子ウィンドウのときはクライアント座標系（タイトルバーもしくはメニューの左下を原点とした座標系）になります。

hWndParent:　親ウィンドウのときは NULL でしたが，子ウィンドウのときは親ウィンドウハンドルを指定します。

hMenu:　親ウィンドウの時はメニューハンドルを指定しましたが，子ウィンドウの時はそれぞれのウィンドウに固有の ID を指定します。この ID は親ウィンドウイベントを通知するときの識別に使用します。

hInstance:　親ウィンドウでも子ウィンドウでもインスタンスハンドルを指定します。

lpParam:　ここで指定した値がそのウィンドウが作成されたときの WM_CREATE メッセージの lParam になります。

例えば，親ウィンドウ全面にエディットコントロールを貼り付けるためには，

① エディットウィンドウのウィンドウハンドルを宣言する。このとき static で宣言するのを忘れないように。
② クライアントウィンドウの幅と高さを GetClientRect 関数で調べる。
③ CreateWindow 関数でエディットウィンドウを作成する。

という手順で行います。CreateWindow で指定するクラス名は"EDIT"，ウィンドウスタイルは，WS_CHILD | WS_VISIBLE | WS_VSCROLL | ES_MULTILINE | ES_LEFT | ES_WANTRETURN | ES_AUTOVSCROLL あたりが良いでしょう。WS_･･･は共通ウィンドウスタイル，ES_･･･エディットコントロール用のスタイルでそれぞれ，チャイルドウィンドウ，可視，垂直スクロールあり，複数行，左揃えテキスト，改行の許可，自動垂直スクロールあり，という意味です。これをテンプレートのソースに書き加えると，list5.4 のようになります。先頭で IDC_EDIT の定義をしていますが，resource.h がある場合はその中に書いた方が良いでしょう。その際は，したの方にある resource.h の下の方にある _APS_NEXT_CONTROL_VALUE の定義を変えておくのを忘れないでください。

さて，このプログラムをビルドしてみましょう。いままで灰色だった親ウィンドウの色が白くなっていると思います。ここでカーソルをクリックするとメモ帳のように「I」型のカーソルに変わるはずです。何かキーボードから入力をしてみてください。まるでワープロのように文字入力ができるとおもいます。実際に Windows に標準で入っているメモ帳はこのエディットコントロールを使っています。このプログラムにファイルの保存と印刷の機能を追加すると誰でもメモ帳ができてしまうということです。

```
#include    <windows.h>

#define IDC_EDIT    1000

char    szWinName[] = "Template";   // ウィンドウクラスの名前
                        ⋮
    HINSTANCE           hInstance;
    RECT                Rect;
    static HWND         hEdit;

    hInstance = (HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE);
                        ⋮
    case WM_CREATE:                 // ウィンドウの作成
        GetClientRect(hWnd, &Rect);
        hEdit = CreateWindow("EDIT", "",
            WS_CHILD | WS_VISIBLE
            | ES_MULTILINE | ES_LEFT | ES_WANTRETURN
            | ES_AUTOVSCROLL | WS_VSCROLL,
            0, 0, Rect.right, Rect.bottom,
            hWnd, (HMENU)IDC_EDIT, hInstance, NULL);
        break;
                        ⋮
    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            DestroyWindow(hEdit);   // エディットコントロールの破棄
            DestroyWindow(hWnd);    // メインウィンドウの破棄. WM_DESTROY発行
        }
        break;
```

list5.4　エディットコントロールの貼り付け

5.3.　タイマー

いろいろと計測を行うためにはタイマーが使えるととても便利です。ここでは， Windows で用意されている2つのタイマーの使い方を紹介します。現在の時刻を一定の周期でエディットコントロールに表示するアプリケーションを作ってみましょう。

5.3.1.　SetTimer

API 関数の中に SetTimer という関数があります。まずはこの関数のプロトタイプを見てみます。

```
UINT SetTimer(
    HWND hWnd,              // ウィンドウハンドル
    UINT nIDEvent,          // タイマの ID
    UINT uElapse,           // 設定時間（単位は ms）
    TIMERPROC lpTimerFunc   // マイマープロシージャ
);
```

この関数を実行すると Windows は，設定時間ごとにタイマプロシージャを実行し，WM_TIMER メッセージをアプリケーションに送ります。WndProc で WM_TIMER メッセージを処理するときは最後の引数は NULL にします。使い終わったら KillTimer 関数でタイマを殺します。

```
BOOL KillTimer(
    HWND hWnd,              // タイマを設定したウィンドウハンドル
    UINT uIDEvent           // タイマの ID
);
```

あと必要なのは時間を取得する関数と，エディットコントロールへの表示方法ですね。時間を取得する関数は GetLocalTome 関数でした（4.2.2 参照）。エディットコントロールへの表示には SendMessage を使います。

SendMessage(エディットコントロールハンドル， EM_REPLACESEL，(WPARAM)FALSE, (LPARAM)文字列);

では，これらの関数を使ってプログラムを作ってみましょう。

まず，最初の方でこれから作るタイマーの ID を決めておきます。たくさん作る時は ID を複数作っておけばいいです。WindowFunc では表示する文字列のための lpszMessage と GetLocalTime 関数のための構造体 stSystemTime の定義をしています。あとは WM_CREATE を捕まえたら，SetTimer 関数でタイマーの設定を行います。周期は 1000ms つまり1秒に設定しました。また，今回は WM_CLOSE を捕まえたときにタイマーを殺しています。

タイマーは一定周期ごとに WM_TIMER を送ってきますので，それを捕まえたら GetLocalTime 関数で現在の時刻を取得して lpszMessage に格納します。あとは SendMessage 関数でエディットコントロールに表示するという具合です。複数のタイマーを設定した場合は，

```
#define IDC_EDIT    1000
#define ID_TIMER    0

LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMsg, WPARAM wp, LPARAM lp){
    HINSTANCE           hInstance;
    RECT                Rect;
    char                lpszMessage[128];
    SYSTEMTIME          stSystemTime;
    static HWND         hEdit;

    case WM_CREATE:                 // ウィンドウの作成

        SetTimer(hWnd, ID_TIMER, 1000, NULL);
        break;

    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            KillTimer(hWnd, ID_TIMER);
            DestroyWindow(hEdit);   // エディットコントロールの破棄
            DestroyWindow(hWnd);    // メインウィンドウの破棄. WM_DESTROY発行
        }
        break;
    case WM_TIMER:
        GetLocalTime(&stSystemTime);
        sprintf(lpszMessage, "%02lu:%02lu:%02lu.%03lu¥xd¥xa",
            stSystemTime.wHour,    stSystemTime.wMinute,
            stSystemTime.wSecond, stSystemTime.wMilliseconds);
        SendMessage(hEdit, EM_REPLACESEL, FALSE, (LPARAM)lpszMessage);
        break;
```

list5.5　SetTimer を使ったタイマー

WPARAM にタイマーの ID が格納されていますので，if 文か switch 文でそれぞれの処理を行うということになります。

さて，それでは実際に実行してみましょう。タイマーの周期を変えて実行してみてください。実行してみるといくつか問題点が出てくると思います。一つはあまり周期の精度が良くないということです。単純な時間表示だけでしたらそれほど問題にもなりませんが，これが計測用となると話は別です。もう一つは長時間実行するとエディットコントロールの表示が止まってしまうということです。実はエディットコントロールの最大サイズは 64kbyte（英数半角文字で 64×1024 文字）までという制限があります。これを回避するためには，①リッチエディットコントロールを使う，②制限文字数を越えたら表示をクリアする，の2通りの方法があります。①の方法については各自勉強していただくことにして，ここでは②の方法について説明します。ログの表示だけであればこれで十分です。

エディットコントロールのバッファがいっぱいになると，WM_COMMAND メッセージが親ウィンドウに通知されます。このとき WPARAM の上位ワードに EN_MAXTEXT が，下位ワードにエディットコントロールの ID が格納されます。また，LPARAM にはエディットコントロールのウィンドウハンドルが格納されます。そこで，WindowFunc でこのメッセージを監視し，メッセージが来たらエディットコントロールをクリアする処理を行えばいいことになります。つまり，list5.6 のようにしておけばいいことになります。ここで使っている SetWindowText 関数は，指定したウィンドウハンドルのタイトルバーを変更する関数ですが，ハンドルがコントロールの場合はコントロール内のテキストを変更する関数です。Help で確認してみてください。

```
case WM_COMMAND:
    if(HIWORD(wp) == EN_MAXTEXT)
        SetWindowText((HWND)lp, NULL);
    break;
```

list5.6　エディットコントロールのクリア

5.3.2.　マルチメディアタイマー

Windows には SetTimer 関数を用いたタイマーの他にもタイマーが用意されています。これはマルチメディアタイマーと呼ばれる機能で本来はビデオやサウンドなどのマルチメディアデバイスを操作するために用意されたものです。マルチメディア系では高速で正確な時間管理が必要となるので，これを利用することでかなり正確なタイマー管理が可能となります。まず，使用する関数から見ていきましょう。

```
MMRESULT timeBeginPeriod(
    UINT uPeriod        // 最小のタイマー解像度を ms 単位で指定します。
);
MMRESULT timeEndPeriod(
    UINT uPeriod        // timeBeginPeriod で指定した最小のタイマー解像度を ms 単位で指定します。
};
MMRESULT timeSetEvent(
    UINT uDelay,        // イベントを発生させる時間間隔を ms で指定します。
    UINT uResolution,   // タイマー解像度を指定します。timeBeginPeriod で指定した値以上にします。
    LPTIMECALLBACK lpTimeProc, // タイマープロシージャを指定します。
    DWORD dwUser,       // タイマープロシージャに与えられるユーザ定義データです。
    UINT fuEvent,       // タイマーイベントのタイプを指定します。
};
MMRESULT timeKillEvent(
    UINT uTimerID       // timeSetEvent の戻値で得たタイマーイベント ID を指定します。
);
```

timeBeginPeriod 関数と timeEndPeriod 関数はタイマーをシステムで動作させるタイマーの解像度を指定します。timeSetEvent 関数が実際にタイマーを開始する関数で，戻値は作成されたタイマーイベントの ID になります。この ID は timeKillEvent 関数で必要になります。uDelay で時間間隔を指定します。uResolution はイベントを発生させるかどうかのタイマーチェックに行くタイマー解像度です。timeBeginPeriod で指定した値以上にする必要があります。

lpTimeProc にはタイマプロシージャを指定します。タイマプロシージャのプロトタイプは，

```
void CALLBACK TimeProc(
    UINT uID,          // タイマイベントの ID
    UINT uMsg,         // 使用できません。
    DWORD dwUser,      // timeSetEvent の dwUser で指定したユーザ定義データです。
    DWORD dw1,         // 使用できません。
    DWORD dw2          // 使用できません。
);
```

となっています。タイマプロシージャの引数は自由に設定できません。そこで，dwUser という引数が用意されています。複数の変数を引数として与えたい場合は，構造体を用意してそのポインタを dwUser として与えるというのが一般的です。fuEvent にはタイマーイベントのタイプを指定します。1度きりの場合は TIME_ONESHOT，繰り返す場合は TIME_PERIODIC を指定します。

前述のタイマーをマルチメディアタイマーに置き換えてみましょう。list5.7 のようになります。マルチメディアタイマを使うためには，mmsystem.h を include することと，winmm.lib をライブラリに追加する必要があります。ライブラリの追加は，「プロジェクト」－「設定」でプロジェクトの設定ダイアログを表示し，リンクタブから「オブジェクト／ライブラリ モジュール」に追加するか，FileView に右ボタンクリックでファイル追加を行います。

それではソースを見ていきましょう。まず，mmsystem.h を include します。続けてタイマープロシージャに渡すユーザ定義データとなる構造体を定義しておきます。今回は hEdit だけなので構造体にする必要はありませんが，後々のためにもこうしておきましょう。次にタイマープロシージャのプロトタイプ宣言をしておきます。これは決まった形なので，自由に決められるのは関数名だけです。WindowFunc の中ではタイマーID の uTimerID とユーザ定義データ構造体 stTimer を定義します。両方とも値を保持する必要があるので static で宣言します。さて，WM_CREATE を捕まえたらエディットコントロールとタイマーの作成を行います。今回は周期を 1000ms つまり1秒にしました。ユーザ定義データには stTimer のアドレス(&stTimer)を与えます。これは dwUser が DWORD (32bit)となっているので，構造体が大きくなると全部を渡すことができなくなるためです。Windows のアドレス(ポインタ)は 32bit となっているため，アドレスを渡すようにすると，どんなデータでも受け渡しすることがかのうとな

```
#include    <mmsystem.h>
                    ⋮
#define IDC_EDIT    1000
typedef struct{
    HWND                hEdit;
}   TIMERSTRUCT;
                    ⋮
void CALLBACK TimerProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1,
                        DWORD dw2);
                    ⋮
    RECT                Rect;
    static UINT         uTimerID;
    static TIMERSTRUCT  stTimer;
                    ⋮
    case WM_CREATE:                 // ウィンドウの作成
        GetClientRect(hWnd, &Rect);
        stTimer.hEdit = CreateWindow("EDIT", "",
            WS_CHILD | WS_VISIBLE
                    ⋮
        timeBeginPeriod(1);
        uTimerID = timeSetEvent(1000, 2, TimerProc,
            (DWORD)&stTimer, TIME_PERIODIC);
        break;
                    ⋮
    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            timeKillEvent(uTimerID);
            timeEndPeriod(1);
            DestroyWindow(stTimer.hEdit);
            DestroyWindow(hWnd);    // メインウィンドウの破棄. WM_DESTROY発行
        }
        break;
                    ⋮
void CALLBACK TimerProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1,
                        DWORD dw2){
    char                lpszMessage[128];
    SYSTEMTIME          stSystemTime;
    TIMERSTRUCT *       lpstTimer = (TIMERSTRUCT *)dwUser;

    GetLocalTime(&stSystemTime);
    sprintf(lpszMessage, "%02lu:%02lu:%02lu.%03lu¥xd¥xa",
        stSystemTime.wHour,    stSystemTime.wMinute,
        stSystemTime.wSecond, stSystemTime.wMilliseconds);
    SendMessage(lpstTimer->hEdit, EM_REPLACESEL, FALSE, (LPARAM)lpszMessage);
}
```

list5.7　マルチメディアタイマー

ります。DWORD にキャストすることも忘れないようにしてください。WM_CLOSE ではタイマーの終了処理を行います。最後にタイマープロシージャを書きます。まず，必要な変数と併せてユーザ定義データ構造体へのポインタを宣言して dwUser を代入しておきます。キャストするのを忘れないでください。これで WindowFunc で作成したエディットコントロールに文字を書きこむことが可能となります。あとは，SetTimer 関数を使ったときとほとんど同じですね。

さて，実行してみましょう。SetTImer 関数を使ったときと比べて時間の精度が上がっていることが確認できると思います。

5.4.　ファイル操作，コモンダイアログ

エディットコントロールとタイマーを使ったアプリケーションに A/D 変換ボードや RS232C 用の関数などを組み合わせると様々な実験計測ができるようになります（ボードの扱いはボード毎に異なるのでここでは扱いません。それぞれのボードのマニュアルを参照してください）。このように計測ができるようになると計測したデータをファイルに保存したくなります。ここではファイルを保存する方法とファイル保存のときによく見かける「ファイル保存ダイアログ」の使い方を紹介します。

5.4.1.　ファイル操作

Windows でオブジェクトを操作するためにはそのオブジェクトを判別するためのハンドルというものが必要になります（ウィンドウハンドルやインスタンスハンドル，コントロールハンドルもその一つです）。ファイルを操作すためのハンドルはファイルハンドルと言い，CreateFile 関数で取得することができます。閉じるときは CloseHandle 関数を用います。

```
HANDLE CreateFile(
    LPCTSTR lpFileName,                 // ファイル名文字列へのポインタを指定します。
    DWORD dwDesiredAccess,              // ファイルへのアクセスモードを指定します。
    DWORD dwShareMode,                  // オブジェクトの共有方法を指定します。
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                                        // 子プロセスへの継承方法（構造体）を指定します。
    DWORD dwCreationDisposition,        // ファイルが存在するとき，しないときの動作を指定します。
    DWORD dwFlagsAndAttributes,         // ファイルの属性およびフラグを指定します。
    HANDLE hTemplateFile                // その他のフラグを指定します。
);
```

lpFileName にはファイル名文字列へのポインタを指定します。ファイル名の代わりに”COM1”などを指定すると通信リソースなどにもアクセスできます（詳しくは Help を参照）。dwDesiredAccess は読込みなら GENERIC_READ，書き込みなら GENERIC_WRITE を指定します。dwShareMode はファイルの共有モードで，ここでオープンしたオブジェクトを他のオープン操作に対して許可するかを指定します。読込みを許可するなら FILE_SHARE_READ，書込みを許可するなら FILE_SHARE_WRITE を指定します。lpSecurityAttributes には通常 NULL を指定します。dwCreationDisposition にはファイルが存在するとき，しないときの動作を指定します。表 5.2 を参照してください。dwFlagsAndAttributes にはファイルの属性およびフラグを指定します。属性には表 5.3 のようなものを指定しますが，通常は FILE_ATTRIBUTE_NORMAL を指定します。フラグはほとんど使うことはないと思います。hTemplateFile は NULL を指

表 5.2　dwCreationDisposition

CREATE_NEW	新しいファイルを作成します。すでに存在する場合は失敗します。
CREATE_ALWAYS	新しいファイルを作成します。すでにある場合は上書きします。
OPEN_EXISTING	ファイルをオープンします。ファイルが存在しない場合は失敗します。
OPEN_ALWAYS	ファイルをオープンします。ファイルがない場合は新たに作成します。
TRUNCATE_EXISTING	ファイルをオープンし，サイズを 0 にします。指定ファイルがない場合は失敗します。

表 5.3　dwFlagsAndAttributes

FILE_ATTRIBUTE_ARCHIVE	アーカイブファイル
FILE_ATTRIBUTE_COMPRESSED	圧縮ファイル
FILE_ATTRIBUTE_HIDDEN	隠しファイル
FILE_ATTRIBUTE_READONLY	リードオンリーファイル
FILE_ATTRIBUTE_SYSTEM	システムファイル
FILE_ATTRIBUTE_TEMPORARY	テンポラリファイル

定してください。関数の戻値がオブジェクトへのハンドルとなります。

```
BOOL CloseHandle(
    HANDLE hObject                  // 閉じるオブジェクトへのハンドルを指定します。
);
```

hObject には閉じるオブジェクトへのハンドルを指定します。

あと必要になるのはファイルからの読み込みと書き込みですすね。それぞれ ReadFile 関数と WriteFile 関数を使います。

```
BOOL ReadFile(
    HANDLE hFile,                   // ファイルハンドルファイルハンドルを指定します。
    LPVOID lpBuffer,                // データを受け取るバッファへのポインタを指定します。
    DWORD nNumberOfBytesToRead,     // 読み込むバイト数を指定します。
    LPDWORD lpNumberOfBytesRead,    // 読み取ったバイト数を格納する変数のアドレスです。
    LPOVERLAPPED lpOverlapped       // FILE_OVERLAPPED を指定したときに必要です。
);
BOOL WriteFile(
    HANDLE hFile,                   // ファイルハンドルを指定します。
    LPCVOID lpBuffer,               // 書き込むデータバッファへのポインタを指定します。
    DWORD nNumberOfBytesToWrite,    // 書き込むバイト数を指定します。
    LPDWORD lpNumberOfBytesWritten, // 書き込んだバイト数を格納する変数のアドレスです。
    LPOVERLAPPED lpOverlapped       // FILE_OVERLAPPED を指定したときに必要です。
);
```

コメントを読めば大体の意味はわかると思います。

5.4.2.　コモンダイアログ

コモンダイアログとは，Windows があらかじめ用意した良く使うと思われるダイアログのことです。代表的なコモンダイアログには表 5.4 のようなものがあります。多くのコモンダイアログはその設定用の構造体を持っているため，それも併記してあります。今回は保存するファイル名の選択 GetSaveFileName 関数のみを取り上げて，その扱い方を見てみま

表 5.4　コモンダイアログ

機能	関数名	構造体名
色の選択	ChooseColor	CHOOSECOLOR
フォントの指定	ChooseFont	CHOOSEFONT
テキストファイルにおける検索／置換	FindText／ReplaceText	FINDREPLACE
開く／保存するファイル名の選択	GetOpenFileName／GetSaveFileName	OPENFILENAME
印刷	PrintDlg	PRINTDLG
印刷のページ設定	PageSetupDlg	PAGESETUPDLG

す。他の関数についてはヘルプを参照してください。また，コモンダイアログを使うためには，①commdlg.h を include する，②comdlg32.lib を追加する，必要があるので忘れずに行ってください。

では，GetSaveFileName 関数のプロトタイプを見てみましょう。

```
BOOL GetSaveFileName(
    LPOPENFILENAME lpofn     // OPENFILENAME 構造体へのポインタを指定します。
);
```

OPENFILENAME 構造体へのポインタ lpofn のみが引数となっています。つまり，この構造体へ必要な情報を入力してポインタを渡せばいいことになります。では，構造体を見てみましょう。

```
typedef struct tagOFN { // ofn
    DWORD           lStructSize;
    HWND            hwndOwner;
    HINSTANCE       hInstance;
    LPCTSTR         lpstrFilter;
    LPTSTR          lpstrCustomFilter;
    DWORD           nMaxCustFilter;
    DWORD           nFilterIndex;
    LPTSTR          lpstrFile;
    DWORD           nMaxFile;
    LPTSTR          lpstrFileTitle;
    DWORD           nMaxFileTitle;
    LPCTSTR         lpstrInitialDir;
    LPCTSTR         lpstrTitle;
    DWORD           Flags;
    WORD            nFileOffset;
    WORD            nFileExtension;
    LPCTSTR         lpstrDefExt;
    DWORD           lCustData;
    LPOFNHOOKPROC   lpfnHook;
    LPCTSTR         lpTemplateName;
} OPENFILENAME;
```

lStructSize はこの構造体のサイズです。hwnOwner はこのダイアログの親ウィンドウのハンドルです。hInstance は lpTemplateName を指定しない場合は無視します。lpstrFilter はフィルタです。

″CSV ファイル(*.csv)¥0*.csv¥0 すべて(*.*)¥0*.*¥0¥0″

のように指定します。文字列の区切りにはヌル文字（¥0）をいれます。最後はヌル文字２つ（¥0¥0）を入れます。lpstrFile に選択されたファイルのフルパス（F:¥USER¥DATA.CSV など）が入ります。nMaxFile は lpstrFile の大きさを指定します。lpstrFileTitle には選択されたファイル名のみが入ります。nMaxFileTitle は lpstrFileTitle の大きさを指定します。Flags にはダイアログ作成時の細かい設定を指定します。読込みのときは OFN_FILEMUSTEXIST | OFN_EXPLORER（存在しないファイルのときは警告を出す），書き込みのときは OFN_OVERWRITEPROMPT | OFN_EXPLORER（上書きするときは警告を出す）を指定すると良いでしょう。必要なメンバだけセットすれば十分なので，ZeroMemory 関数で初期化しておくと便利です。

では，5.3.2 で作ったプログラムにファイル書き込みの機能を追加しましょう。タイマを開始する前にファイル名を取得してファイルを開きます。画面に表示しているのと同じ内容をファイルに保存し，×を押したらファイルを閉じて終了するプログラムにします。

list5.8 にソースを示します。まず，先頭で commdlg.h の include を行います。次にタイマプロシージャの中でファイルの書き込みを行うことになるので TIMERSTRUCT 構造体に hFile ハンドルを追加しておきます。WM_CREATE を捕まえたら，タイマを作成する前にファイル保存コモンダイアログを表示する準備をします。ファイル名バッファ lpszFileName と構造体 stOfn を宣言して，それぞれ初期化しておきます。ファイル名も初期化しておかないと実行する際にエラーを起します。構造体の設定は一般的なものにして，拡張子のデフォルトを csv（カンマ区切りデータファイル）にしておきました。この形式は Excel 等で標準で読み込むことができます。準備ができたら GetSaveFileName 関数でファイル名を取得します。成功したらこのファイル名を使って CreateFile 関数でファイルを書き込みモードで作成します。ファイル名の読込みとファイルの作成で失敗したらメッセージを表示して終了処理もしています。あとはタイマーを作成しています。WM_CLOSE にはファイルを閉じる操作も加えておきましょう。

タイマープロシージャでは表示用に作成した文字列をそのままファイルに WriteFile 関数を使って書き込んでいます。書き込んだバイト数が間違っていたときはエディットコントロールに「Zero bytes write.」というメッセージを表示しているのもわかると思います。

```
#include    <commdlg.h>
                    ⋮
typedef struct{
    HWND                hEdit;
    HANDLE              hFile;
}   TIMERSTRUCT;
                    ⋮
    case WM_CREATE:                 // ウィンドウの作成
                    ⋮
        char                lpszFileName[MAX_PATH] = "";
        OPENFILENAME        stOfn;

        ZeroMemory(&stOfn, sizeof(OPENFILENAME));
        stOfn.lStructSize   = sizeof(OPENFILENAME);
        stOfn.hwndOwner     = hWnd;
        stOfn.lpstrFilter
            = "CSV File(*.csv)¥0*.csv¥0All Files(*.*)¥0*.*¥0¥0";
        stOfn.lpstrFile     = lpszFileName;
        stOfn.lpstrDefExt   = "csv";
        stOfn.nMaxFile      = MAX_PATH;
        stOfn.Flags         = OFN_OVERWRITEPROMPT;

        if(GetSaveFileName(&stOfn) == TRUE){
            stTimer.hFile = CreateFile((LPCTSTR)stOfn.lpstrFile,
                GENERIC_WRITE,
                FILE_SHARE_WRITE,
                NULL,
                CREATE_ALWAYS,
                FILE_ATTRIBUTE_NORMAL,
                (HANDLE)NULL);
            if(stTimer.hFile == (HANDLE)-1){
                MessageBox(hWnd, "File open faild.", NULL, MB_OK);
                PostMessage(hWnd, WM_CLOSE, 0, 0L);
                break;
            }
        }else{
            MessageBox(hWnd, "File name get faild.", NULL, MB_OK);
            PostMessage(hWnd, WM_CLOSE, 0, 0L);
            break;
        }
        timeBeginPeriod(1);
        uTimerID = timeSetEvent(1000, 2, TimerProc,
            (DWORD)&stTimer, TIME_PERIODIC);
        break;
                    ⋮
    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            CloseHandle(stTimer.hFile);
            timeKillEvent(uTimerID);
                    ⋮
void CALLBACK TimerProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1,
                    ⋮
    DWORD               dwBytesWrite;
    WriteFile(lpstTimer->hFile, (LPVOID)lpszMessage, strlen(lpszMessage),
        &dwBytesWrite, NULL);
    if(dwBytesWrite == 0){
        SendMessage(lpstTimer->hEdit, EM_REPLACESEL, FALSE,
            (LPARAM)"Zero bytes write.¥xd¥xa");
    }
}
```

list5.8　ファイル保存コモンダイアログ

5.5.　最後に

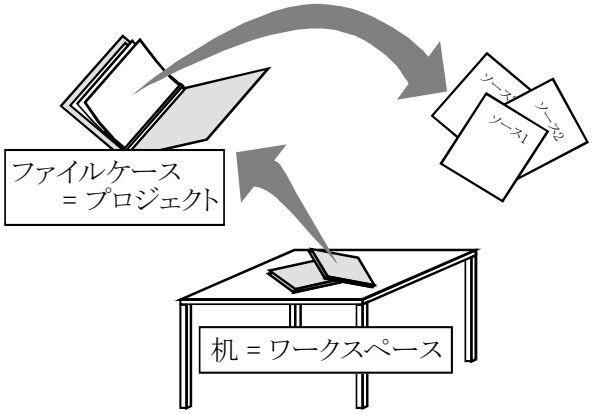
Windows プログラムを駆け足でやってきましたが，ここで掲載したのはそのごく一部に過ぎません。ここで紹介しきれなかったコントロールや関数がまだまだ数多くあります。付録に参考資料を載せておきますので，これを参考にいろいろなプログラムに挑戦してみてください。

A.　Visual C++ Ver.6 の使い方

1.　Visual C++の構造

Visual C++ではファイルを図 A.1 のように3段階に分けて管理しています。Visual C++を立ち上げただけでは何もない状態になっています。そこで，プログラムを作るためにの中心になる作業場（ワークスペース）を作ることになります。このワークスペースの中にはいくつかのファイルケース（プロジェクト）を置くことができます。このプロジェクトが最終的にはそれぞれのプログラムになります。そして，ファイルケースの中にはプロジェクトを行うために必要な手順書（ソース）や資料（リソース）を入れる，といことです。

では実際に list1.1 のプログラムを作るまでの手順を順に説明していきます。



図A.1　ワークスペース・プロジェクト・ソースの関係

2.　ワークスペースを作る

初めに作業場となるワークスペースを作ってみましょう。[ファイル(F)]－[新規作成(N)...]を選ぶと，図 A.2 のようなダイアログが出てきます。そこで，図の丸で囲った[ワークスペース]というタグを選択します。「ブランクワークスペース」という項目が見えると思います。これは「プロジェクトも何もないワークスペースをこれから作成する」ということです。基本となる[位置](ディレクトリ)と[ワークスペース名]を入力して，OK を押すと新しい空のワークスペースが作成されます。例えば，位置を「F:¥USER¥」，ワークスペース名を「CHAP1」として，ワークスペースを作ってみましょう。（「CHAP1」と入力すると「F:¥USER¥」の後ろに，自動的に「CHAP1」が追加されます）

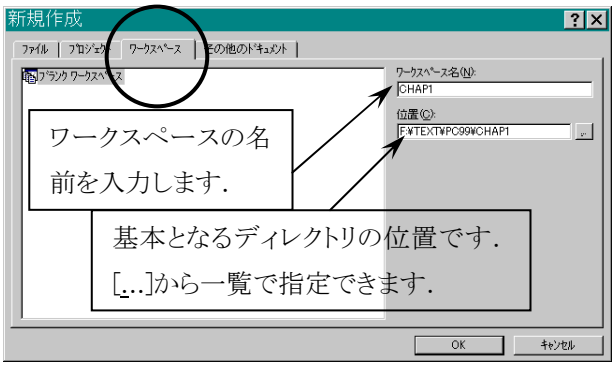


図 A.2　新規作成ダイアログ（ワークスペース）

図 A.3 は作成された新規ワークスペースです。「ワークスペース'CHAP1':0 プロジェクト」と書いてあるのは，今はまだプロジェクトがないといことを示しています。では，この図で各部の名称と機能を確認しておきましょう。

2.1.　FileView

これはワークスペース内のファイル構成を表示します。ここに新しいプロジェクトを作成していくことになります。プロジェクトが作成されるとその中に「Source File」，「Header File」，「Resource File」というフォルダが作成され，その中に実際にプロジェクトを構成するファイルを入れていくことになります。

2.2.　ClassView

これはワークスペース内のクラス構成を表示します。C 言語では関係ないです。C++言語を使うようなアプリケーションを作成する時は，クラスの構成，関係などが樹形図で表示されます。

2.3.　ResourceView

これはワークスペース内のリソース構成を表示します。リソースとはアイコン，ダイアログ，メニュー，カーソル，ビットマップ，ツールバー，アクセラレータ（ショートカットキー），ストリングテーブルなど，Windows アプリケーションに必要な構成要素を示します。

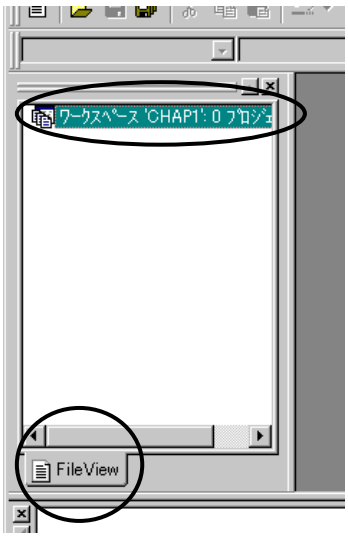


図 A.3　新規ワークスペース

3.　プロジェクトを作る

ワークスペースができたので，その中にプロジェクトを作ってみましょう。[ファイル(F)]－[新規作成(N)...]で新規作成ダイアログを出します。今度は，プロジェクトを作成するので，[プロジェクト]タグを選択します。たくさんの項目が並んでいますが，これがこれから作成するアプリケーションの性質を決定します。良く使うものは以下のものです。

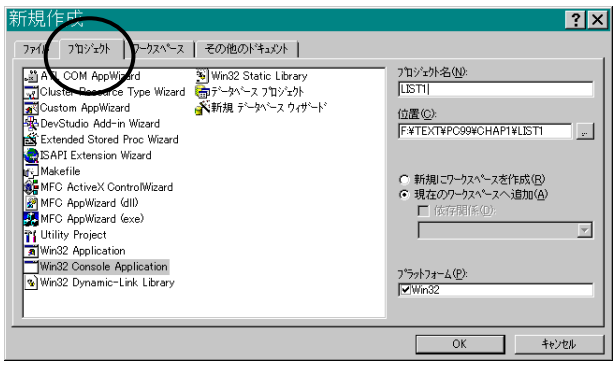


図 A.4　新規作成ダイアログ（プロジェクト）

Win32 Application　－　一般的な Windows アプリケーションです。

Win32 Console Application　－　DOS 窓で動くアプリケーションです。

Win32 Dynamic-Link Library　－　DLL と呼ばれるライブラリ（関数の集合体）です。必要な時に動的に読み込まれ，要らなくなったらメモリから開放されます。読み込むアプリケーションと DLL のバージョンがあわないと正常動作しない時があります。

Win32 Static-Link Library　－　リンク時にプログラム中に取り込まれるライブラリです。プログラムと一体化しているため，バージョンの不一致などを気にする必要がなくなりますが，アプリケーションそのものが大きくなり，メモリの消費も大きくなります。

今回はコンソール用のアプリケーションを作成するので，[Win32 Console Application]を選択します。ここで[位置]を確認しておきましょう。ワークスペースを作成したディレクトリ（今回の場合なら「F:¥USER¥CHAP1¥」）になっていればOK です。なっていない場合は[...]で指定しておきましょう。次に，[プロジェクト名]を入力します。例えば，「LIST1」と入力します。すると，[位置]で指定したディレクトリに「LIST1」を追加したディレクトリ名が自動的に入力されます。あとは，せっかく作ったワークスペースに追加するわけですから，[現在のワークスペースに追加(A)]を選択して，[OK]を押します。すると，図 A.5 の「Win32 Console Application」というダイアログが出てきます。

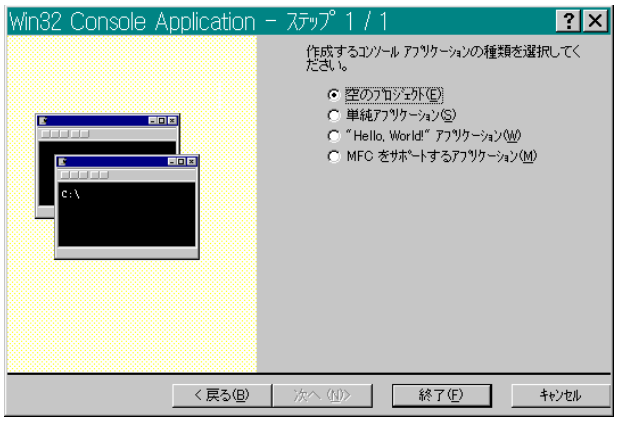


図 A.5　Win32 Console Application の種類

空のプロジェクト　－　プロジェクトのみを作成します。

単純アプリケーション　－　プロジェクトと共にソースのひな形を作ってくれます。

"Hello, World!"アプリケーション　－　"Hello, World!"と表示するアプリケーションのプロジェクトとソースを作ってくれます。基本構造を知るにはよいでしょう。それ以外の使い道はありません。

MFC をサポートするアプリケーション　－　MFC(Microsoft Foundation Class)を使うことのできるアプリケーションのプロジェクトとソースのひな形を作ってくれます。MFC の詳細は Help でも見てください。

今回は基本を知るためにも「空のプロジェクト」を選択しましょう。普段は「単純アプリケーション」の方が使いやすいです。「終了」を押すと，ワークスペースが図 A.6 のようにプロジェクト数が 1 になり，LIST1 ファイルというプロジェクトが作成されます。LIST1 ファイルの下の3つのフォルダが見えない時は，左にある「+」を押すと，見えるようになります。ここまできたら，あとはソースファイルを書くだけです。

図 A.6　新規プロジェクト

4.　ソースファイルを作る

さて，下準備が終わったので，実際にソースを書いてプログラムらしくしましょう。ソースファイルを作るには，ワークスペースの「Source Files」を選択して，[ファイル(F)]－[新規作成(N)...]です。すると今度は，図 A.7 のような新規作成ダイアログが現れます。ソースファイルを書くのですから，「C++ソースファイル」を選択して，[ファイル名]を入力します。(C じゃなく C++になっているのは気にしなくてもかまいません。気になる人は「LIST1.C」のようにファイル名に拡張子も付けて書いてください。)[OK]を押すと，「LIST1cpp」というファイルが作成され，右側のウィンドウが真っ白になります。

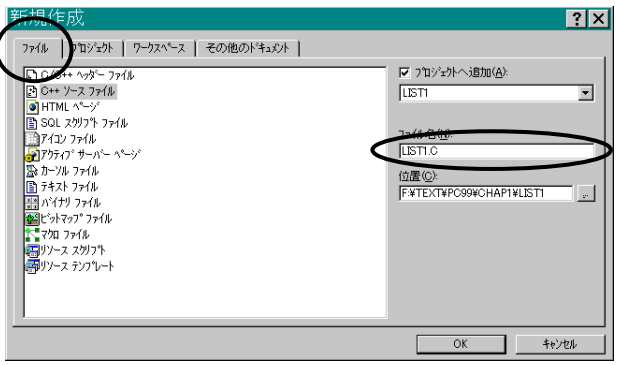


図 A.7　新規作成ダイアログ(ファイル)

ここにプログラムのソースを書いていくことになります。list1.1 を書くと図 A.8 のような感じになります。

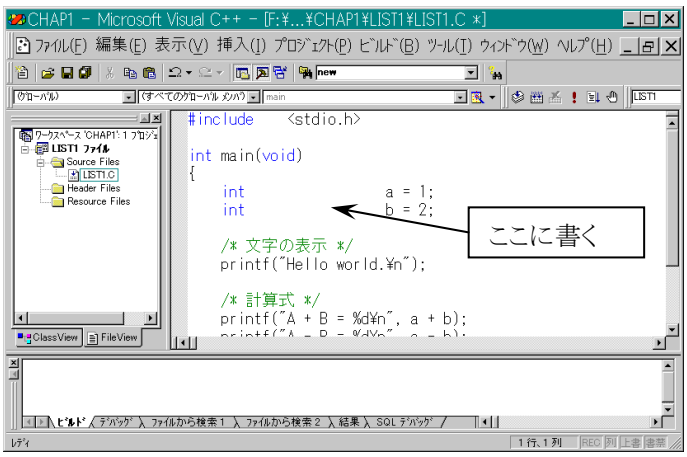


図 A.8　ソースを書いたところ

C/C++言語はスペースや改行を無視します。そのため，これらをうまく使うことで見やすいソースを書くことができます。見やすいソースはバグ(プログラムの問題点)を見つけやすくし，開発の効率を挙げるために役立ちます。一般的なスタイルを挙げると，

1)　インデント(字下げ)する。

「{」から「}」で囲まれたブロックや関数の範囲を明確にすることで，有効範囲やループの始端・終端をはっきりさせます。

2)　処理の区切りに改行を入れる。

これもプログラム全体の流れをつかみやすくします。

3)　「,」の後ろ(場合によっては「(」,「)」の前後)にスペースを入れる。

式が繋がっているとどこからどこまでが変数かわかりづらくなります。適当にスペースを入れることで，それを解消します。

4)　長い行は適当なところで改行する。

ソースを書いている画面の横幅はそれほど大きくありません。横にはみ出してしまうと，スクロールした時にバグを見落としてしまう原因となります。

5)　一つの関数は 20～30 行程度に納め，適度に関数に分ける。

あまり長い関数だと初めの方で行った処理を忘れたりして，新たなバグを生む原因となります。うまく関数を利用して見通しの良いソースを作るようにしましょう。

6)　コメントをうまく利用する

適度にコメントを入れるようにしましょう。後で見た時の理解の手助けになります。

5.　ビルドする

作成したプロジェクト内のソースをそれぞれコンパイルし，リンクすることを「ビルド」と言います。ビルドすることで，実行ファイルが作成されます。今回作成した list1.1 もビルドして，きちんと書けているか確認をしてみます。

ビルドするには図 A.9 のビルドメニューから，[ビルド(B)]－[ビルド(B) List1.exe]を選択します。すると，下の方の「アウトプットウィンドウ」に途中経過が表示されます。もし，

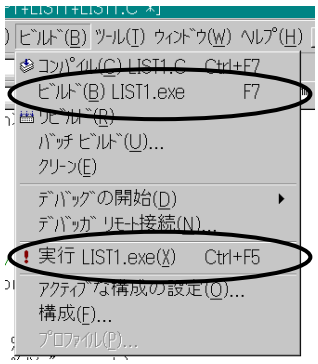


図 A.9　ビルドメニュー

うまくいくと図 A.10 のように表示されます。書き間違いやエラーがあるとアウトプットウィンドウにエラーあった行番号とエラーの内容が表示されます。エラーをダブルクリックすると，そのエラーのあった行にジャンプしてくれるので，修正して再度ビルドしてみましょう。

```
--------------------構成: LIST1 - Win32 Debug--------------------
ｺﾝﾊﾟｲﾙ中...
LIST1.C
ﾘﾝｸ中...

LIST1.exe - ｴﾗｰ 0、警告 0
```

図 A.10　アウトプットウィンドウ

6.　実行する

さて，ビルドがうまくいったら実行してみましょう。[ビルド(B)]－[実行 LIST1.exe(E)]を選択すると，Win32 コンソールが開いて，図 A.11 のような実行結果が表示されます。予想通りの答えになっているでしょうか？予想通りの答えになっていない時はソースのどこかに間違いがあるということです。このように，実行段階で間違いを発見するとその原因がどこなのか見つけるのがとても困難になります。また，結果が予想のつかない場合などは，間違いそのものを見つけることが困難になります。できるだけ，見やすいソースを書くことがバグを減らす一番のポイントです。

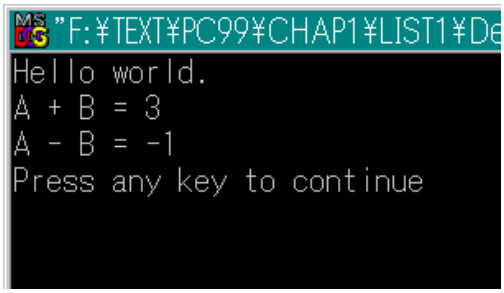


図 A.11　実行結果

7.　さらに

7.1.　Debug モードと Release モード

図 A.10 のアウトプットウィンドウの表示で Win32 Debug と出ています。これは Debug(バグ取り)モードでコンパイルしているという意味です。この Debug モードとは実行ファイルの中に Debug 用のコードを埋めこみ，実行途中の状態をモニタできるようにしているということです。大きなプログラムでバグがあった時などに有効です。しかし，プログラムサイズが大きくなるという欠点もあります。そこで，実際に実行するための実行ファイルを作るためのモードが Release モードです。Release モードに変更するには，[ビルド(B)]－[アクティブな構成の設定(O)...]を選択して，「プロジェクトの標準構成ダイアログ」から Release と書いてあるものを選択します。

7.2.　ワークプレースに複数のプロジェクトを共存させる

練習問題毎にワークプレースを作成するのは不便なので，1つのワークプレースに複数のプロジェクトを共存させましょう。方法は簡単です。すでにプロジェクトが 1 つあるワークプレースに，「3.プロジェクトを作る」の方法でプロジェクトを追加するだけです。

7.3.　デバッグ

ひとたび実行ファイルになってしまったら，バグを探すのは大変だということはすでに書きましたが，では実際にそのようなときはどうすればいいのでしょうか。そのような時のデバッグ法を2つほど紹介します。

1）値の表示

怪しいところの値を表示してみるという方法です。どのあたりでおかしくなっているか判っている時に便利です。

2）デバッガを使う

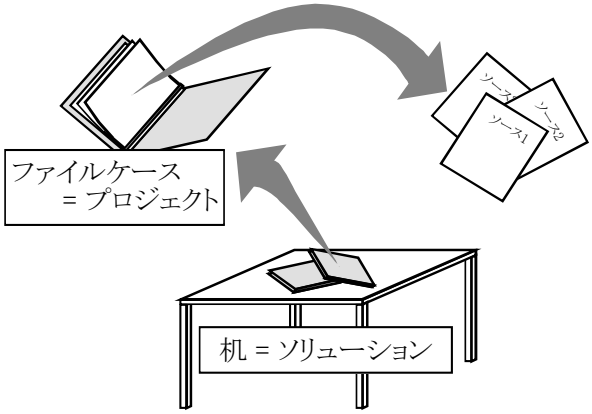
[ビルド(B)]－[デバッグの開始(D)]－[ステップイン(I)]からアプリケーションを実行すると，デバッグモードでの実行になります。1 行ごと実行したり，ブレークポイント(停止位置)まで実行したりしながら，値を見ることができます。

B.　Visual Studio .NET の使い方

1.　Visual Studio .NET の構造

Visual Studio .NET ではファイルを図 A.1 のように3段階に分けて管理しています。Visual Studio .NET を立ち上げただけでは何もない状態になっています。そこで，プログラムを作るためにの中心になる作業場（ソリューション）を作ることになります。このソリューションの中にはいくつかのファイルケース（プロジェクト）を置くことができます。このプロジェクトが最終的にはそれぞれのプログラムになります。そして，ファイルケースの中にはプロジェクトを行うために必要な手順書（ソース）や資料（リソース）を入れる，といことです。

では実際に list1.1 のプログラムを作るまでの手順を順に説明していきます。



図B.1　ソリューション・プロジェクト・ソースの関係

2.　ソリューションを作る

初めに作業場となるソリューションを作ってみましょう。[ファイル(F)]－[新規作成(N)]－[空のソリューション(B)...]を選ぶと，図 B.2 のようなダイアログが出てきます。基本となる[位置](ディレクトリ)と[プロジェクト名]にソリューション名を入力して，OK を押すと新しい空のソリューションが作成されます。例えば，位置を「F:¥USER¥」，ソリューション名を「CHAP1」として，ソリューションを作ってみましょう。（「CHAP1」と入力すると「F:¥USER¥」の後ろに，自動的に「CHAP1」が追加されます）

図 B.3 は作成された新規ソリューションです。「ソリューション’CHAP1’:0 プロジェクト」と書いてあるのは，今はまだプロジェクトがないといことを示しています。では，この図で各部の名称と機能を確認しておきましょう。

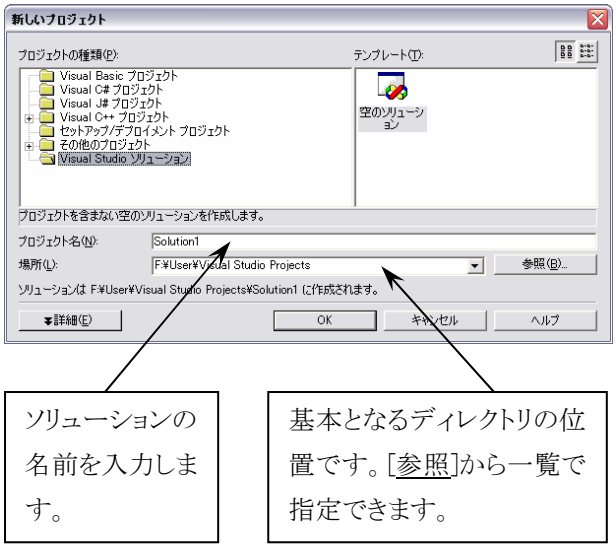


図 B.2　新規作成ダイアログ（ソリューション）

2.1.　ソリューションエクスプローラ

これはソリューション内のファイル構成を表示します。ここに新しいプロジェクトを作成していくことになります。プロジェクトが作成されるとその中に「参照設定」，「ソースファイル」，「ヘッダーファイル」，「リソースファイル」というフォルダが作成され，その中に実際にプロジェクトを構成するファイルを入れていくことになります。

2.2.　クラスビュー

これはソリューション内のクラス構成を表示します。C 言語では関係ないです。C++言語を使うようなアプリケーションを作成する時は，クラスの構成，関係などが樹形図で表示されます。

2.3.　リソースビュー

これはソリューション内のリソース構成を表示します。リソースとはアイコン，ダイアログ，メニュー，カーソル，ビットマップ，ツールバー，アクセラレータ（ショートカットキー），ストリングテーブルなど，Windows アプリケーションに必要な構成要素を示します。

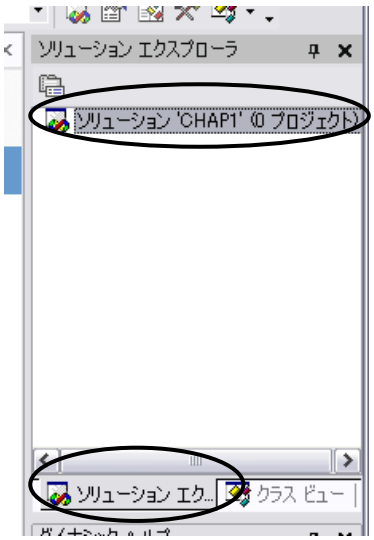


図 B.3　新規ソリューション

3.　プロジェクトを作る

ソリューションができたので，その中にプロジェクトを作ってみましょう。ソリューションエクスプローラの CHAP1 ソリューションでマウスの右ボタンをクリックし，[追加(D)]－[新しいプロジェクト(N)...]で新しいプロジェクトの追加ダイアログを出します。たくさんの項目が並んでいますが，これがこれから作成するアプリケーションの性質を決定します。良く使うものは以下のものです。

図 B.4　新しいプロジェクトの追加ダイアログ

Win32 プロジェクト ― 一般的な Windows アプリケーションです。

Win32 コンソールプロジェクト ― コマンドプロンプトで動くアプリケーションです。

今回はコンソール用のアプリケーションを作成するので，[Win32 コンソールプロジェクト]を選択します。次に，[プロジェクト名]を入力します。例えば，「LIST1」と入力します。すると，[位置]で指定したディレクトリに「LIST1」を追加したディレクトリ名が自動的に入力されます。[OK]を押すと，アプリケーションウィザードダイアログが出てきます。ここで少し設定をしておきます。ダイアログ左側の[アプリケーションの設定]を押すと図 B.5 のような表示になります。アプリケーションの種類に注意しましょう。上の2つ，Windows アプリケーションとコンソールアプリケーションは先ほどの説明の通りです。その下の2つは，

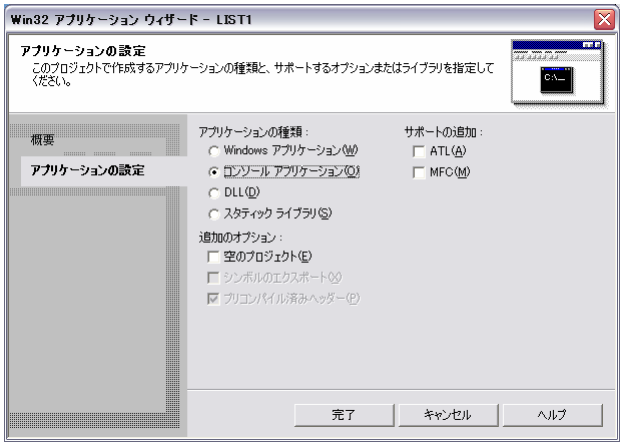


図 B.5　Win32 Console Application の種類

DLL（Dynamic Link Library） ― 実行時の必要なときに動的に読み込まれるライブラリ（関数の集合体）です。必要に応じて入れ替えが可能で，複数のアプリケーションで共通の機能を共有することができます。

スタティックライブラリ ― リンク時にプログラム中に取り込まれるライブラリです。プログラムと一体化しているため，バージョンの不一致などを気にする必要がなくなりますが，アプリケーションそのものが大きくなり，メモリの消費も大きくなります。

です。追加のオプションは[空のプロジェクト(E)]をチェックしないと，プロジェクトと共にソースのひな形を作ってくれますが，今回は基本を知るためにも「空のプロジェクト(E)」をチェックしましょう。プロジェクトが図 B.6 のようにプロジェクト数が 1 になり，LIST1 ファイルというプロジェクトが作成されます。実際にこれから作成するファイルはこれらのフォルダに分類されます。必要に応じてフォルダを追加することもできます。ここまできたら，あとはソースファイルを書くだけです。

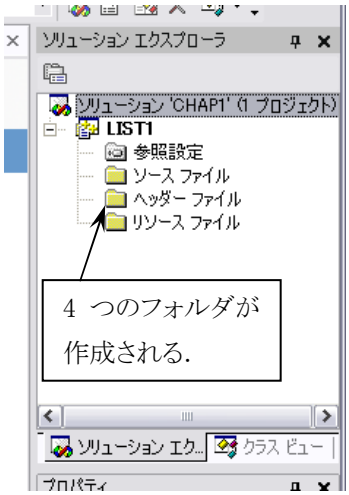


図 B.6　新規プロジェクト

3.1.　プロジェクトのプロパティ

プロジェクトのプロパティを用いて作成するアプリケーションに関する様々な設定をすることができます。ソリューションエクスプローラ内の LIST1 プロジェクトを選択し，マウスの右ボタンメニューからプロパティを選択します。Visual Studio.NET は 64bit システム対応となっていますが，今回の授業では特に必要ないので，[構成プロパティ]－[C/C++]－[全般]－[64ビット移植への対応]を[いいえ]に変更しておきましょう。

4.　ソースファイルを作る

さて，下準備が終わったので，実際にソースを書いてプログラムらしくしましょう。ソースファイルを作るには，ソースファイルフォルダでマウスの右ボタンクリックをして，[追加(D)]－[新しい項目の追加(W)...]です。すると今度は，図 B.7 のような新しい項目の追加ダイアログが現れます。ソースファイルを書くのですから，「C++ファイル」を選択して，[ファイル名]を入力します。(C じゃなく C++になっているのは気にしなくてもかまいません。気になる人は「LIST1.C」のようにファイル名に拡張子も付けて書いてください。)[OK]を押すと，「LIST1cpp」というファイルが作成され，LIST1 という真っ白いウィンドウが現れます。ここにプログラムのソースを書いていくことになります。list1.1 を書くと図 B.8 のような感じになります。

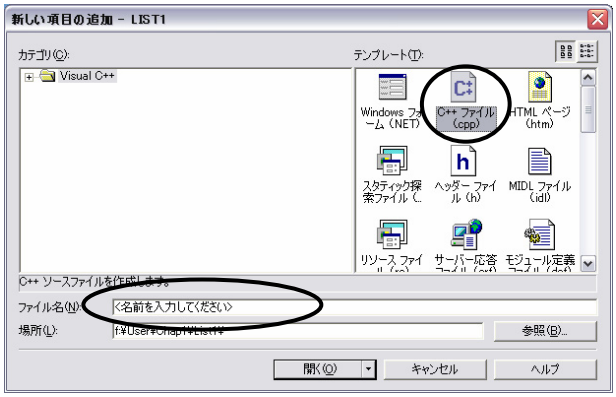


図 B.7　新規作成ダイアログ（ファイル）

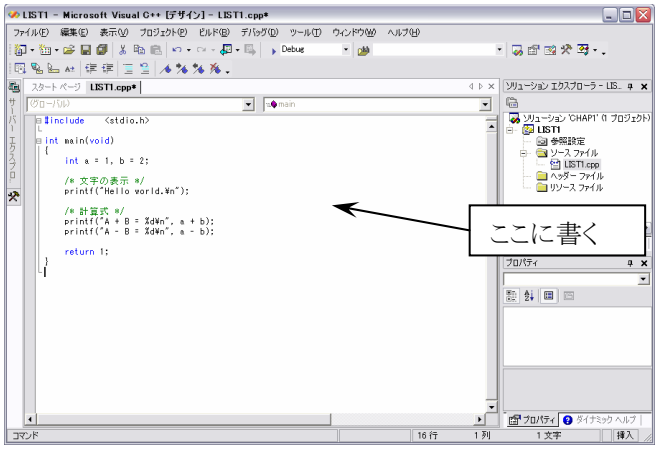


図 B.8　ソースを書いたところ

C/C++言語はスペースや改行を無視します。そのため，これらをうまく使うことで見やすいソースを書くことができます。見やすいソースはバグ（プログラムの問題点）を見つけやすくし，開発の効率を挙げるために役立ちます。一般的なスタイルを挙げると，

1）インデント（字下げ）する。

「{」から「}」で囲まれたブロックや関数の範囲を明確にすることで，有効範囲やループの始端・終端をはっきりさせます。入力には Tab キーを使います。

2）処理の区切りに改行を入れる。

これもプログラム全体の流れをつかみやすくします。

3）「,」の後ろ（場合によっては「(」，「)」の前後）にスペースを入れる。

式が繋がっているとどこからどこまでが変数かわかりづらくなります。適当にスペースを入れることで，それを解消します。

4）長い行は適当なところで改行する。

ソースを書いている画面の横幅はそれほど大きくありません。横にはみ出してしまうと，スクロールした時にバグを見落としてしまう原因となります。

5）一つの関数は 20～30 行程度に納め，適度に関数に分ける。

あまり長い関数だと初めの方で行った処理を忘れたりして，新たなバグを生む原因となります。うまく関数を利用して見通しの良いソースを作るようにしましょう。

6）コメントをうまく利用する

適度にコメントを入れるようにしましょう。後で見た時の理解の手助けになります。

5.　ビルドする

作成したプロジェクト内のソースをそれぞれコンパイルし，リンクすることを「ビルド」と言います。ビルドすることで，実行ファイルが作成されます。今回作成した list1.1 もビルドして，きちんと書けているか確認をしてみます。

ビルドするには図 B.9 のビルドメニューから，[ビルド(B)]－[LIST1 のビルド(U)]を

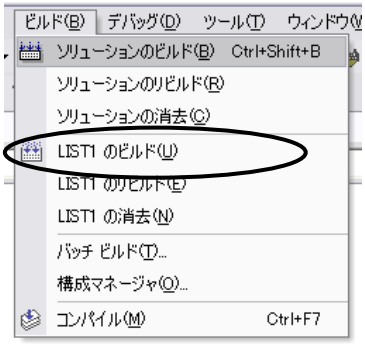


図 B.9　ビルドメニュー

選択します。すると，下の方の「アウトプットウィンドウ」に途中経過が表示されます。もし，うまくいくと図 B.10 のように表示されます。書き間違いやエラーがあるとアウトプットウィンドウにエラーあった行番号とエラーの内容が表示されます。エラーをダブルクリックすると，そのエラーのあった行にジャンプしてくれるので，修正して再度ビルドしてみましょう。

```
------ ビルド開始 : プロジェクト : LIST1, 構成 : Debug Win32 ------

コンパイルしています...
LIST1.cpp
リンクしています...

ビルドログは "file://f:¥User¥Chap1¥List1¥Debug¥BuildLog.htm" に保存されました。
LIST1 - エラー 0、警告 0

---------------------- 終了 ----------------------

    ビルド : 1 正常終了、0 失敗、0 スキップ
```

図 B.10　アウトプットウィンドウ

6.　実行する

さて，ビルドがうまくいったら実行してみましょう。[デバッグ(D)]－[デバッグなしで開始(G)]を選択すると，Win32 コンソールが開いて，図 B.11 のような実行結果が表示されます。予想通りの答えになっているでしょうか？予想通りの答えになっていない時はソースのどこかに間違いがあるということです。このように，実行段階で間違いを発見するとその原因がどこなのか見つけるのがとても困難になります。また，結果が予想のつかない場合などは，間違いそのものを見つけることが困難になります。できるだけ，見やすいソースを書くことがバグを減らす一番のポイントです。

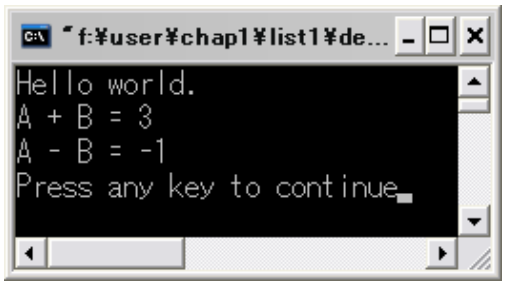


図 B.11　実行結果

7.　さらに

7.1.　Debug モードと Release モード

図 B.10 のアウトプットウィンドウの表示で Debug Win32 と出ています。これは Debug(バグ取り)モードでコンパイルしているという意味です。この Debug モードとは実行ファイルの中に Debug 用のコードを埋めこみ，実行途中の状態をモニタできるようにしているということです。大きなプログラムでバグがあった時などに有効です。しかし，プログラムサイズが大きくなるという欠点もあります。そこで，実際に実行するための実行ファイルを作るためのモードが Release モードです。Release モードに変更するには，[ビルド(B)]－[構成マネージャ(O)...]を選択して，「アクティブソリューション構成(A)」から Release と書いてあるものを選択します。

7.2.　ソリューションに複数のプロジェクトを共存させる

練習問題毎にソリューションを作成するのは不便なので，1つのソリューションに複数のプロジェクトを共存させましょう。方法は簡単です。すでにプロジェクトが 1 つあるソリューションに，「3.プロジェクトを作る」の方法でプロジェクトを追加するだけです。

7.3.　デバッグ

ひとたび実行ファイルになってしまったら，バグを探すのは大変だということはすでに書きましたが，では実際にそのようなときはどうすればいいのでしょうか。そのような時のデバッグ法を2つほど紹介します。

1）値の表示

怪しいところの値を表示してみるという方法です。どのあたりでおかしくなっているか判っている時に便利です。

2）デバッガを使う

[デバッグ(D)]－[ステップイン(I)]からアプリケーションを実行すると，デバッグモードでの実行になります。1 行ごと実行したり，ブレークポイント(停止位置)まで実行したりしながら，値を見ることができます。

C.　Visual C++ Toolkit 2003 と Platform SDK の使い方

1.　Visual C++ Toolkit 2003 と Platform SDK

Visual C++ Toolkit（以下，VCTK）は Microsoft が無償で提供している開発環境で，これと Platform SDK を組み合わせる（以下，VCTK+PSDK）ことで C/C++による開発環境を構築することができます。ただし，VCTK+PSDK では Visual Studio Ver.6 や Visual Studio .NET のような統合開発環境がありませんので，マウスで感覚的に操作するということはできません。また，多少コマンドプロンプト操作の知識を必要とします。

VCTK+PSDK をインストールして開発環境を設定して，list1.1 のプログラムを作るまでの手順を順に説明していきます.

2.　開発環境を作る

2.1.　Visual C++ Toolkit 2003 のインストール

参考資料のリンクから VCTK をダウンロードします。VCTK にはコンパイラ，リンカ，C/C++のヘッダファイルとライブラリが含まれますので，これだけでも基本的なプログラムを作成できます。インストールはダウンロードした VCToolkitSetup.exe を実行するだけです。

2.2.　Platform SDK のインストール

PSDK には Windows プログラム作成に必要なヘッダファイル，ライブラリファイルとリソースコンパイラが含まれています。また，複数ソースファイルをまとめてコンパイル・リンクするためのメイクも含まれています。参考資料のリンクから PSDK をダウンロードします。ダウンロードできるファイルは，AMD64bit 用，Intel64bit 用，Intel 汎用とフルバージョンの分割ファイルおよび CD イメージ（ISO 形式）がありますので，自分の環境に合ったものを選択します。インストールはダウンロードした実行ファイルを実行するだけです。

2.3.　環境設定用のバッチファイルを作る

VCTK と PSDK のインストールが済んだら実行ファイルへのパス，インクルードファイルへのパス，ライブラリファイルへのパスを環境変数に設定するために list C.1 のバッチファイルをメモ帳などで作成します。ファイル中の VCTK と PSDK はそれぞれインストールした場所に応じて書き換えて下さい。このファイルをパスの通ったフォルダもしくは windows フォルダに vctk.bat のような名前をつけて保存して下さい。これで開発環境は完成です。VCTK+PSDK では開発にコマンドプロンプトを使用しますが，その際は必ず初めに"vctk"を実行して，環境変数の設定をするようにしてください。うまくコンパイルなどが実行できないときはこのバッチファイルを見直して下さい。

```
@echo off

set VCTK=C:¥Program Files¥Microsoft Visual C++ Toolkit 2003
set PSDK=C:¥Program Files¥Microsoft SDK

set PATH=%VCTK%¥bin;%PSDK%¥Bin;%PSDK%¥Bin¥Win64;%PATH%
set INCLUDE=%VCTK%¥include;%PSDK%¥Include
set LIB=%VCTK%¥lib;%PSDK%¥Lib
```

list C.1　環境設定用バッチファイル

2.4.　コマンドプロンプトの使い方

MS-DOS を使ったことのある人であればコマンドプロンプトは容易に使うことができます。使ったことのない人はエクスプローラと併用すると良いでしょう。その場合，覚えておいた方が良いコマンドは，ドライブの移動とフォルダの移動，ファイルの一覧表示の3つだけで充分です。

1) ドライブの移動　－　ドライブレターを入力します。（ex.　A:, C:, . . . ）
2) フォルダの移動　－　"cd 〈フォルダ名〉"を入力します。スペースの入ったフォルダの場合はフォルダ名を""で囲みます。また1階層上のフォルダに移動するときは"cd .."と入力します。（ex.　cd program, cd "Documents and Settings", ...）
3) ファイルの一覧表示　－　"dir"と入力すると現在のフォルダのファイル一覧を表示します。

また，参考資料にある Open Command Window Here というツールをインストールしておくと，好きなフォルダを右クリ

ックするだけで，そのフォルダでコマンドプロンプトが開くので便利です。

3.　ソースファイルを作る

さて，下準備が終わったので，実際にソースを書いてプログラムを作成しましょう。VCTK+PSDK でソースファイルを書くにはエディタを使用します。使うエディタは Windows 標準のメモ帳，秀丸エディタ，などどのようなものでもかまいません。自分の使い慣れたものを選ぶようにしましょう。では，エディタを使って list1.1 を書いて LIST1.cpp のような名前で保存してみましょう。作成中に作業ファイルなどができるので C:¥Program¥CHAP1 のようにプログラム毎にフォルダを作っておくと便利でしょう。

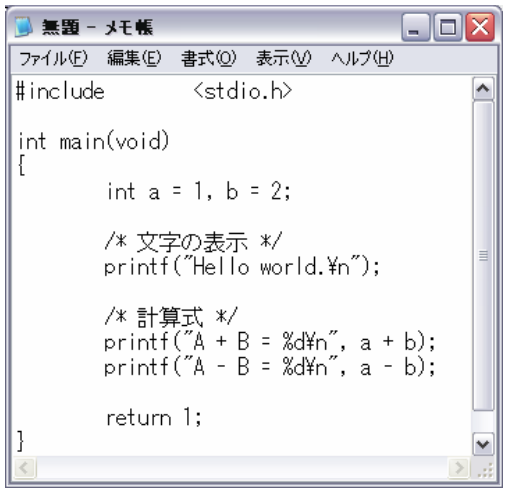


図 C.1　メモ帳でソースを書いたところ

C/C++言語はスペースや改行を無視します。そのため，これらをうまく使うことで見やすいソースを書くことができます。見やすいソースはバグ（プログラムの問題点）を見つけやすくし，開発の効率を挙げるために役立ちます。一般的なスタイルを挙げると，

1）インデント（字下げ）する。

「{」から「}」で囲まれたブロックや関数の範囲を明確にすることで，有効範囲やループの始端・終端をはっきりさせます。インデントには Tab キーを使います。

2）処理の区切りに改行を入れる。

これもプログラム全体の流れをつかみやすくします。

3）「,」の後ろ（場合によっては「(」，「)」の前後）にスペースを入れる。

式が繋がっているとどこからどこまでが変数かわかりづらくなります。適当にスペースを入れることで，それを解消します。

4）長い行は適当なところで改行する。

ソースを書いている画面の横幅はそれほど大きくありません。横にはみ出してしまうと，スクロールした時にバグを見落としてしまう原因となります。

5）一つの関数は 20～30 行程度に納め，適度に関数に分ける。

あまり長い関数だと初めの方で行った処理を忘れたりして，新たなバグを生む原因となります。うまく関数を利用して見通しの良いソースを作るようにしましょう。

6）コメントをうまく利用する。

適度にコメントを入れるようにしましょう。後で見た時の理解の手助けになります。

4.　ビルドする

作成したプロジェクト内のソースをそれぞれコンパイルし，リンクすることを「ビルド」と言います。ビルドすることで，実行ファイルが作成されます。今回作成した list1.1 もビルドして，きちんと書けているか確認をしてみます。

VCTK+PSDK でビルドするにはコマンドプロンプトを使用します。プログラムメニューからコマンドプロンプトを起動して，ソースファイルを保存したフォルダに移動するか，Open Command Window Here を使って保存したフォルダでコマンドプロンプトを開きます。移動したら，まず環境変数を設定するために準備で作成したバッチファイル（vctk）を実行します。

```
C:¥Program¥CHAP1> vctk↵
```

ビルドするには，コマンドラインでコンパイラ cl.exe を実行します。オプションを付けなければ自動的にリンカも呼び出してくれます。

```
C:¥Program¥CHAP1> cl list1.cpp↵
```

ビルドに成功すると,

```
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for
80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

list1.cpp
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:list1.exe
list1.obj
```

のように表示されます。間違いがあった場合はエラーと行番号が表示されますので，それに合わせて修正しましょう。複数のソースファイルに分かれている場合には後に続けてソースファイル名を列記すれば自動的にビルドしてくれます。

5.　実行する

さて，ビルドがうまくいったら実行してみましょう。

```
C:¥Program¥CHAP1> list1↵
Hello world.
A + B = 3
A - B = -1
```

予想通りの答えになっているでしょうか？予想通りの答えになっていない時はソースのどこかに間違いがあるということです。このように，実行段階で間違いを発見するとその原因がどこなのか見つけるのがとても困難になります。また，結果が予想のつかない場合などは，間違いそのものを見つけることが困難になります。できるだけ，見やすいソースを書くことがバグを減らす一番のポイントです。

6.　さらに

6.1.　Make ファイルを使う

Windows アプリケーションのように複数のソースファイルを使うようになると，コマンドラインで毎回全てのソースを書いてビルドするのは不便になってきます。そのようなときは Make ファイルを使うと便利です。Make ファイルの書式は，list C.2 のように

ターゲット: 依存関係

　　コマンド

となっており，「依存関係」のファイルから「ターゲット」を作成するには「コマンド」を実行する。「依存関係」のファイルが存在しない場合はそのファイルが「ターゲット」になっている行を探して実行するという形式になっています。このファイルを他のソースファイルと同じフォルダに”Makefile”という名前で保存し，コマンドラインで，

```
all: list1.exe

list1.exe: list1.obj
    link list1.obj

list1.obj: list1.cpp
    cl /c list1.cpp
```

list C.2　Makefile

```
C:¥Program¥CHAP1> nmake↵
```

と実行すると，コマンドラインに直接記述したときと同様の結果が得られます。

6.2.　フリーの開発環境を使う

VCTK+PSDK にフリーの開発環境を追加することでコマンドラインを使わなくても C/C++を使ったプログラミング環境を作ることができます。フリーの開発環境もいくつかありますが，ここでは CPad for Borland C++ Compiler（以下，BCPad）の設定法を設定について説明します。この開発環境はもともとは Borland C++というコンパイラ用に開発されたものですが，設定を変更することで VCTK+SDK の環境でも使用することが可能です。他の開発環境もほぼ同様なので使いやすい環境を探してみて下さい。

1)　ダウンロードとインストール

参考資料のリンクから BCPad をダウンロードします。ダウンロードしたファイルは圧縮ファイルになっており，適当な

フォルダに Lhasa や LHMelt などを用いて解凍するだけでインストール作業なしに使用することができます。解凍したフォルダの中の BCPad.exe を実行してみましょう。図 C.2 の様な画面が出てくるはずです。上がエディタ画面，下がコンパイル時のメッセージウィンドウとなっています。BCPad.exe へのショートカットなどを作っておくと便利でしょう。

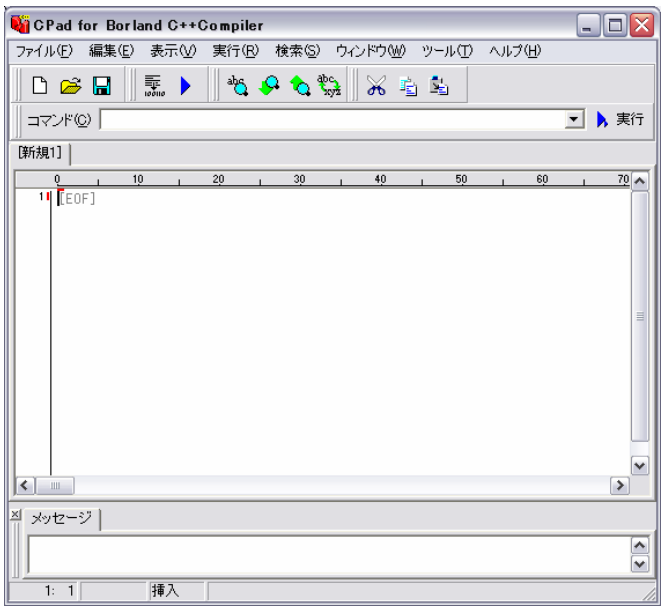


図 C.2　BCPad

2)　環境設定

[実行(R)]－[設定(S)...]から VCTK+SDK に関する設定をしておきましょう。まず，[基本設定]タブのコンパイラのパス(W)にはインストールした VCTK フォルダの bin にある cl.exe を指定します。あと，[高度な設定]タブを開き，[以下の設定を変更する(C)]をチェックして，

[追加する環境変数(C)]

```
VCTK=C:¥Program Files¥Microsoft Visual C++ Toolkit 2003
PSDK=C:¥Program Files¥Microsoft SDK
INCLUDE=%VCTK%¥include;%PSDK%¥Include
LIB=%VCTK%¥lib;%PSDK%¥Lib
```

[追加するパス(P)]

```
%VCTK%¥bin
%PSDK%¥Bin
%PSDK%¥Bin¥Win64
```

のように設定しておきましょう。

3)　使い方

エディタウィンドウにプログラムのソースを書いて，[実行(R)]－[コンパイル(C)]でコンパイル，[実行(R)]－[実行(R)]で実行することができます。エラーなどはメッセージウィンドウに表示されます。

D.　参考資料

<書籍>

柴田望洋：秘伝 C 言語問答，SOFTBANK，ISBN4-89052-172-0

C 言語のうち，ポインタにだけ注目して1冊の本にまとめたもの。ある程度，C 言語が使えるようになったら一度読んでおくとポインタに対する疑問が解けます。

柴田望洋：C プログラマのための C++入門，SOFTBANK，ISBN4-89052-339-1

C++の入門書です。C 言語を理解している人が C++言語を使って疑問に思うところを的確に説明してくれます。

山本信雄：VisualC++①，翔泳社，ISBN4-88135-821-9

Visual C++を使って Windows プログラムを作る手順を順を追って説明しています。①は基本から Win32Application の作り方を，②は MFC（Microsoft Foundation Class）を使ったプログラミングの基礎を，③は MFC を使った実践的なプログラミングの説明があります。

北山洋幸：Win32API システムプログラミング withVisualC++6.0，カットシステム，ISBN4-87783-021-9

DLL やマルチスレッドなど Windows を使った実用アプリケーションを作成する際に必要となるプログラミング技法について解説しています。基本的な Windows アプリケーション作成法を覚えたら一度読んでおいた方が良いでしょう。

<Web>

粂井康孝：猫でもわかるプログラミング，http://www.kumei.ne.jp/c_lang/

C，C++，Win32，MFC すべての説明が網羅されているホームページです。特に Win32 に関しては膨大なサンプルとともに丁寧な解説が書かれています。このホームページを見ればほとんどのことは足りるでしょう。

宍戸輝光：創作プログラミングの街，http://www.sm.rim.or.jp/~shishido/

グラフィック処理系プログラムに重点を置いたホームページです。Windows（C/C++）だけでなく JAVA，Linux，数学系処理についても触れています。

Codeguru，　http://www.codeguru.com/

海外のプログラマーがそれぞれのプログラムコードを寄せ合う集会場です。様々な分野のアルゴリズムがあるので，コード作成の参考に。

<Visual C++ Toolkit 2003 + Platform SDK>

Visual C++ Toolkit 2003，　http://msdn.microsoft.com/visualc/vctoolkit2003/

Windows® Server 2003 SP1 Platform SDK Web Install，　http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en

Windows® Server 2003 SP1 Platform SDK ISO Install，　http://www.microsoft.com/downloads/details.aspx?familyid=D8EECD75-1FC4-49E5-BC66-9DA2B03D9B92&displaylang=en

CPad，　http://hp.vector.co.jp/authors/VA017148/pages/cpad.html

WideStudio，　http://www.widestudio.org/

Visual C++ Toolkit ＋ Platform SDK のインストールとテスト，　http://homepage1.nifty.com/kazubon/progdoc/poor/vctoolkit.html

Visual C++ Toolkit 2003，　http://www.02.246.ne.jp/~torutk/cxx/vc/vctoolkit2003.html

C/C++言語による実験・計測アプリケーションの作成

2000 年 5 月 23 日　初版印刷
2005 年 5 月 6 日　３版発行
著　者　石井　一暢
発行者　石井　一暢
発行所　農用車両システム工学研究室

Printed in Japan