2「オブジェクト指向」以外でのC言

2–1 コメント

- /* から */ までは、C言語の場合と同様にコメント。(人間向け)
- // から その行の最後までは注釈。

補足 (TODOコメント):

既存のプログラムを眺めていると、

```
//TODO ...
```

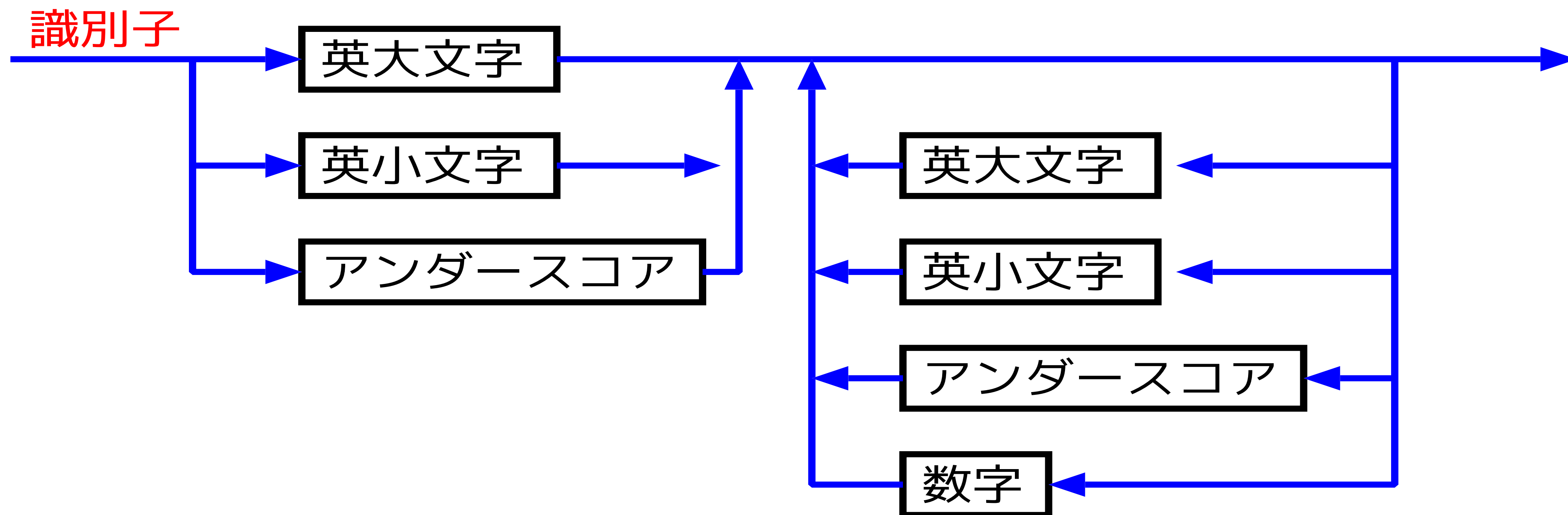
という風に “**TODO**” や “**FIXME**”, “**XXX**” で始まるコメントに出会うことがある。これらの意味は次の通りである。

`//TODO` 作業	··· 作業 ということをやらなければならない。 **(to do)**
`//FIXME` 修正作業	··· 正しく動いてないので 修正作業 に記述されている様な修正が必要。 **(fix me)**
`//XXX` 間違い	··· とりあえず動いてはいるが 間違い に記述されている様に正しくない。

2–2 識別子とキーワード

識別子：

変数等に付ける名前としては、次のものが許される。



但し、キーワード (下記) は識別子として使うことはできない。

また、次の 2 つは識別子としての使用は避けるべき

- 連続したアンダースコア (下線, _) を含む文字列、
- 1 文字目がアンダースコアで 2 文字目が英大文字の文字列

キーワード：次の通り。

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	templat	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

コーディング規約：

一般に、コードの可読性·保守性を高めたり不具合を招く可能性を抑制したりするために、コーディング規約が設定されていることがある。

Java言語の場合 は...

名称	Webページのアドレス
電通国際情報サービス版	http://www.objectclub.jp/community/codingstandard/JavaCodingStandard2004.pdf
オブジェクト倶楽部版	http://www.objectclub.jp/community/codingstandard/CodingStd.pdf
Java言語コーディング規約 (Sunコーディング規約)	http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html (原文), http://www.oracle.com/technetwork/java/codeconvtoc-136057.html (原文), http://numata.designed.jp/javacodeconv/ (和訳)
頑健なJavaプログラムの書き方	http://www.alles.or.jp/~torutk/oojava/codingStandard/writingrobustjavacode.html

これらのコーディング規約の中には 、
◇変数名に対する命名の仕方、
◇書式、
◇不具合を招くコードと回避例、
等が示されている。
実際には、互いに矛盾した指針もある。
しかし、こういった規約にも十分留意すべきである。

例えば、
識別子に対する名前の付け方として 次の様な指針が一般的である。
◇ 英単語など、意味のある単語を並べて名前を付ける。
◇「意味のある単語」としては、不可解な短縮形は使わない。

2–3 基本データ型

- C言語の基本データ型に加えて、bool と wchar_t

- bool型... 論理値(i.e.{真,偽})を値域とするデータ型
  bool型の定数値を表すキーワード... true(真) とfalse(偽)
  (C言語では int型で代用)

- wchar_t型... 255文字を超える文字セット(e.g.Unicode)
  内の文字を保持するためのデータ型

- どちらも整数型の一種として分類される。

		整数としての値
false	...	0
true	...	1

  また、0 以外の整数値... true(真) と見做される。
  (C言語の場合と同様)

2–4 リテラル(定数)

C++では定数のことをリテラルと呼んでいる。

整数型リテラル：

- **10**進, **16**進, **8**進の表記が可能。(**C**言語と同様。)

- `int`で表せれば`int`型，さもなければ`long`型。但し、最後に `L` または `l` を付けると`long`型になる。

浮動小数点数型リテラル：

- **C**言語の場合とほぼ同じ。(標準は`double`型で、…… 。)

文字リテラル：

- C言語の場合と同じで、`'A'` の様に文字を1重引用符で囲んで表す。
- データ型は `char`。(C言語では `int` 型だった。)
- コンピュータ上で採用されている文字符号体系に従って、各文字には文字コード(長さ8のビット列)が割り当てられている。

  文字リテラルの値... 文字コードの表す整数値(文字番号)
  (C言語の場合と同様、`char` は整数型の一種。)

論理型リテラル：

- `true` と `false` の2つ。

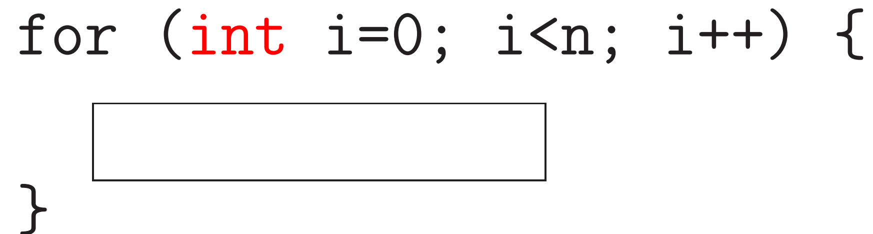
2–5 変数の宣言

- C 言語と同様の書き方。
- 局所変数の場合は、C 言語の場合と同様に、
  宣言によって(暗黙値に)初期化される訳ではない。
- C++の場合は、変数の宣言も文(statement)の一種と考え、
  プログラムのどこに変数宣言を置いてもよい。

但し、使う前に宣言しておく。
(変数の)名前の範囲に気を付ける。

C 言語では、
変数宣言は文の扱いではなく、
ブロックの最初に固めて置くことになっていた。

- for文中のループ制御変数については、
  値を初期設定する場所で宣言することもできる。
  例えば、



```
for (int i=0; i<n; i++) {

}
```

と書くことができ、この場合、
ループ制御変数 i はこのループの中だけで有効になる。

const修飾子：

- 初期設定を伴う変数宣言の前に const というキーワードを置くと、その変数は値の変更を行えなくなる。　(行うとコンパイルエラー)

- const宣言された変数は
定数式の一部として使える。　…定数だけから構成される式
⟶配列宣言の際の要素数指定に使える。　(C言語では 、不可)

⇨ (引数なし)マクロの代わりに
const宣言された変数を用いることができる。

- 関数の外でconst宣言された変数の名前
◇(デフォルトでは)そのソースファイル内だけで有効
◇他のソースファイルからも見える様にするためには、
const 修飾子の前に明示的にキーワード extern を置く

(参考) クラス,局所変数,定数変数に対する命名規則：

Java 言語における Sun コーディング規約

- クラス名,フィールド名,変数名の付け方 (定数値領域を除く)：
  - ◇ 主として英小文字を用いる。
  - ◇ 2 つ以上の「意味のある単語」から構成する場合は、2 つ目以降の単語の頭文字を大文字にする。
  - ◇ クラス名の場合 は、先頭文字を大文字にし、名詞 (句) を表す単語列にする。
  - ◇ フィールド名,変数名の場合 は、先頭文字を小文字にする。

- 定数値を保持するフィールド名,変数名の付け方：
  - ◇ 「意味のある単語」を構成する全ての英文字を大文字にする。
  - ◇ 2 つ以上の「意味のある単語」から構成する場合は、(単語の区切を明らかにするために) 単語間にアンダースコア (_, 下線記号) を入れる。

2–6 型変換

算術計算の際の自動型変換：　　…C言語と同様

① 演算前に

bool型, char型, short型, 列挙型 ⟶ int型　　に変換

int型で表せない値 ⟶ unsigned型

② 異なる型のデータ間で演算する場合、⟶ 型を揃える。そのために、データ型間の次の順序に従って、演算前に下位(i.e.左)の方の型が上位の型に変換される。(変換後の型が演算結果の型になる。)

int < unsigned < long < unsigned long
< float < double < long double

⇨ char型同士の加算結果はcharではなくint型。

キャスト(型変換)：式の値をデータ型という型に変換したい時、次の書き方も可。

- C言語の場合と同様に、
  (データ型)式
- 関数記法(古いC++):
  データ型(式)

古いキャスト記法では、
◇ 安全で何の問題もない型変換も、的確でなくコンパイラによる正当性チェックも難しい型変換も、1種類の演算子／関数に押し込められていて、
◇ 過度の利用がエラーの大きな原因となることがある、
という指摘

⇨ 次の名前付きキャストが導入された。

⇨ ◇ 型変換を目に付き易くし、
◇ プログラマがキャストの意図を表現できる
ようにするために、次の名前付きキャストが導入された。

- 安全な型変換の場合　(i.e. 暗黙の型変換も可能な様な場合)

 `static_cast<データ型>(式)`

- 安全でない場合
 (e.g. 整数 ↔ ポインタ間の様に変換結果がシステムに依存)

 `reinterpret_cast<データ型>(式)`　(出来れば使用を避ける)

- const 性,volatile 性 (揮発性) の属性を付けたり外したりしたい場合

 `const_cast<データ型>(式)`

- クラス階層で上位クラス (基底クラス) のオブジェクトを下位クラス (派生クラス) のオブジェクトと見做したい場合

 `dynamic_cast<データ型>(式)`

2–7 スコープ解決演算子

スコープ：プログラム内に現れる名前は、それぞれに固有の有効範囲(スコープ)をもつ。⟶ 次の範囲設定だけが可能

- 局所スコープ… ブロック中の、局所名の宣言以降の部分。
- クラススコープ… クラスや構造体の定義を記述した範囲
- 名前空間スコープ… 名前空間の定義中で、所属する名前の宣言以降の部分。
- 大域スコープ(ファイルスコープ)
  … ソースファイル中で、
  関数定義やクラス定義、列挙クラス定義、名前空間定義の外に置かれた、大域的な名前の宣言以降の部分。

・・

スコープを指定した名前の参照：

スコープ解決演算子(対象特定演算子) :: により可能。

例えば、

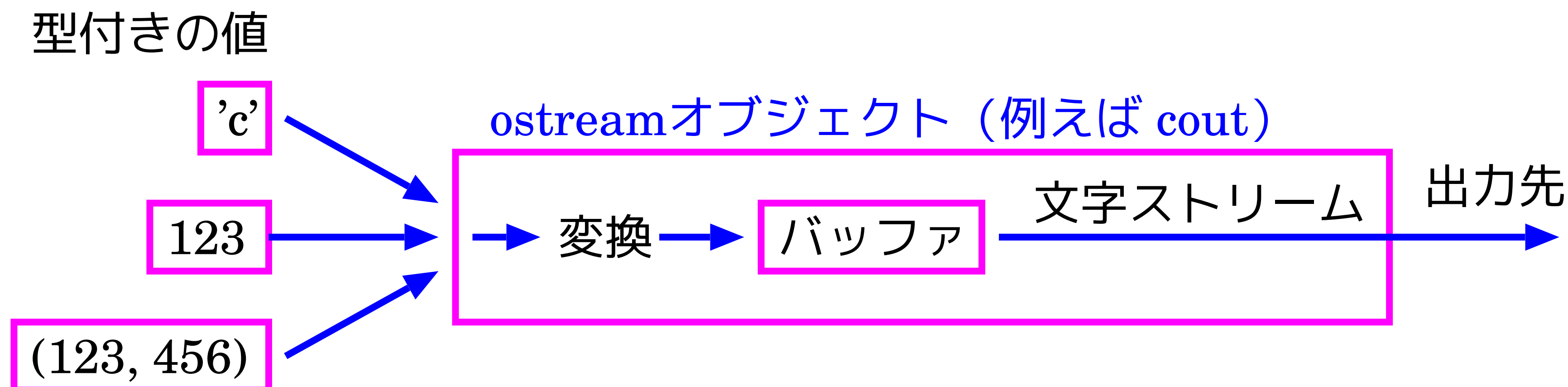
- :: 名前 … ソースファイル中の大域的な 名前 (最大のファイルスコープ)
- 名前空間名 :: 名前 … 名前空間 名前空間名 中に登録された関数, クラス, オブジェクト, 等の 名前
- クラス名 :: 名前 … クラス クラス名 のスコープ中の要素の 名前

2–8 ストリーム入出力

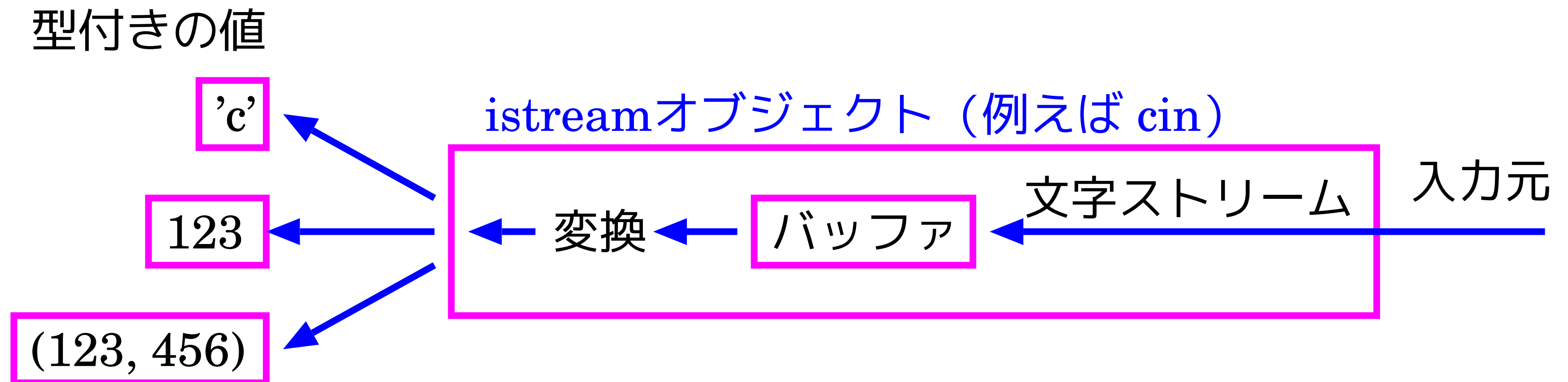
C言語と違って
ストリームをモデル化したクラス群を標準ライブラリに用意して、入出力機能を実現。

C言語の場合は 入出力用の種々の関数が標準ライブラリ内に...

- `ostream`オブジェクトは、
  (言語処理系によって識別されたオブジェクトの型情報に従って)
  オブジェクトを適切な文字(バイト)ストリームに変換する。



また、istreamオブジェクトは、
文字(バイト)ストリームを適切な型のオブジェクトに変換する。



⇨ 型安全である。すなわち、入出力対象のデータ型を理解した上で常に適切な変換処理が為される。

C言語のprintf()やscanf()では
式の型に合わない書式を指定して
不可解な実行結果に至る可能性がある。

- C言語の printf() や scanf() との併用も可。

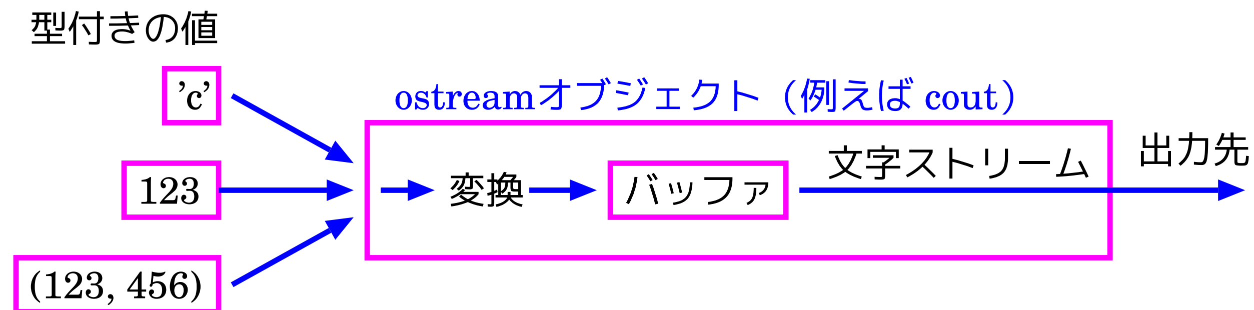
> そのためには
>
> C言語の cstdio ライブラリと C++言語の iostream ライブラリのそれぞれが保有しているバッファを同期させる必要があり、
>
> ```
> ios::sync_with_stdio();
> ```
>
> という関数呼び出しをライブラリを使う前に行なっておく。

- iostream ライブラリ内には、次の標準ストリームが用意されている。

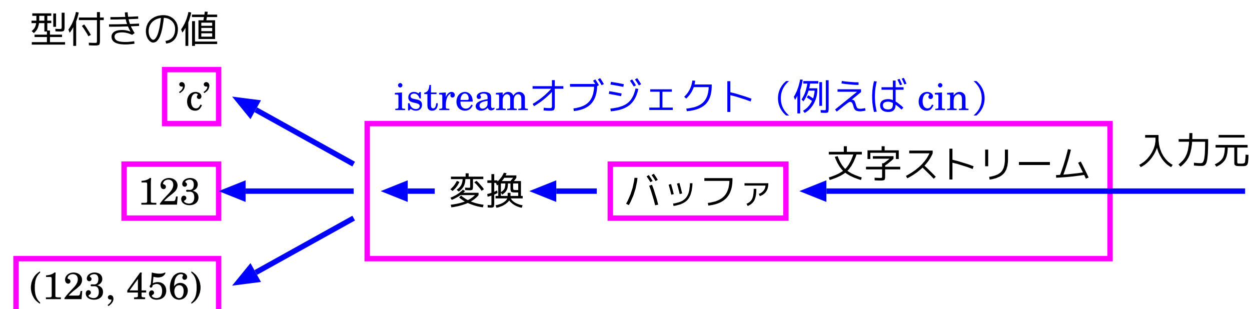
  cin ・・・ 標準入力
  cout ・・・ 標準出力
  cerr ・・・ 標準エラー出力(バッファリングされない)
  clog ・・・ 標準エラー出力(バッファリングされる)

- iostream ライブラリ内では 、
通常はシフト演算子として使われる << と >> を「多重定義」し、

◇ << の左側に ostream オブジェクト (e.g.cout) が来た時には
ostream オブジェクトが持っている出力機能を利用、



◇ >> の左側に istream オブジェクト (e.g.cin) が来た時には
istream オブジェクトが持っている入力機能を利用



書式を指定した出力：

演算子 << を用いて`ostream`オブジェクト(e.g.`cout`)に流し込まれたデータは、デフォルトでは表示に必要な最短の文字列となって出力される。

⇨ 出力の際の書式を設定したい場合は、
設定したい書式に関連したマニピュレータ(操作子) を<<演算子の右側に置く

> 演算結果は 、
> 第1オペランドとして指定された`ostream`オブジェクト。
>
> マニピュレータを受けた`ostream`オブジェクトは、
> 内部の動作設定を変える。

具体的なマニピュレータ(操作子)としては 、
次の様なもの。(他にも多数)

マニピュレータ	機能	マニピュレータが定義されているヘッダファイル
endl	バッファに溜まったデータと改行コードを順に出力先に吐き出す。	iostream
ends	ヌル文字を出力(バッファに追加)。	iostream
flush	バッファに溜まったデータを出力先に吐き出す。	iostream
left	左寄せで出力する様に ostream オブジェクトの動作設定を変える。	iostream
right	右寄せで出力する様に ostream オブジェクトの動作設定を変える。	iostream
setw(n)	次の出力時に少なくとも n 桁で出力する様に ostream オブジェクトの動作設定を変える。(注意： 次の次以降の出力については、何の動作設定も行わない。)	iomanip
setfill(c)	空いた桁に文字 c を埋めて出力する様に ostream オブジェクトの動作設定を変える。	iomanip

(次ページに続く)

マニピュレータ	機能	マニピュレータが定義されているヘッダファイル
dec	整数の入出力を 10 進表記で行う様に ostream オブジェクトの動作設定を変える。	iostream
oct	整数の入出力を 8 進表記で行う様に ostream オブジェクトの動作設定を変える。	iostream
hex	整数の入出力を 16 進表記で行う様に ostream オブジェクトの動作設定を変える。	iostream

(次ページに続く)

マニピュレータ	機能	マニピュレータが定義されているヘッダファイル
fixed	浮動小数点数を指数部なしの表記で出力する様にostreamオブジェクトの動作設定を変える。	iostream
scientific	浮動小数点数を小数点の前に1桁,指数部付きの表記で出力する様にostreamオブジェクトの動作設定を変える。	iostream
setprecision(n)	浮動小数点数を小数点以下 n 桁の桁数で出力する様にostreamオブジェクトの動作設定を変える。	iomanip
boolalpha	bool型の値を true または false という形で出力する様にostreamオブジェクトの動作設定を変える。	iostream
noboolalpha	bool型の値を 1 または 0 という形で出力する様にostreamオブジェクトの動作設定を変える。	iostream

例2. 1 (sin(),cos(),tan()の表) それぞれの$k \in \{0,1,2,3,\ldots,12\}$に対して$\sin(k\pi/12)$, $\cos(k\pi/12)$, $\tan(k\pi/12)$の値を計算して、それらの結果を表の形に出力するC++プログラムを次に示す。

```
[motoki@x205a]$  cat -n printTableOfSinCosTan.cpp
     1 /* k=0,1,2,...,12として                          */
     2 /* sin(k*PI/12), cos(k*PI/12), tan(k*PI/12) */
     3 /* の表を出力するC++プログラム                 */
     4
     5 #include <iostream>
     6 #include <iomanip>
     7 #include <cmath>
     8 using namespace std;
     9
    10 const double PI = 3.141592653589793;   // 円周率
    11
```

```
12 int main()
13 {
14   cout << " k  sin(k*PI/12)  cos(k*PI/12)  tan(k*PI/12)
15        << "--  ------------  ------------  ------------
16   for (int k=0; k<=12; k++) {
17     cout << right
18          << setw(2) << k
19          << fixed << setprecision(9)
20          << setw(14) << sin(k*PI/12.0)
21          << setw(14) << cos(k*PI/12.0)
22          << scientific << setprecision(5)
23          << setw(14) << tan(k*PI/12.0) << endl;
24   }
25 }
[motoki@x205a]$  g++ printTableOfSinCosTan.cpp
[motoki@x205a]$  ./a.out
```

```
 k  sin(k*PI/12)  cos(k*PI/12)  tan(k*PI/12)
--  ------------  ------------  ------------
 0   0.000000000   1.000000000   0.00000e+00
 1   0.258819045   0.965925826   2.67949e-01
 2   0.500000000   0.866025404   5.77350e-01
 3   0.707106781   0.707106781   1.00000e+00
 4   0.866025404   0.500000000   1.73205e+00
 5   0.965925826   0.258819045   3.73205e+00
 6   1.000000000   0.000000000   1.63312e+16
 7   0.965925826  -0.258819045  -3.73205e+00
 8   0.866025404  -0.500000000  -1.73205e+00
 9   0.707106781  -0.707106781  -1.00000e+00
10   0.500000000  -0.866025404  -5.77350e-01
11   0.258819045  -0.965925826  -2.67949e-01
12   0.000000000  -1.000000000  -1.22465e-16
[motoki@x205a]$
```

istream オブジェクト (e.g. cin) のメンバ関数： 内部に次の関数も...

関数プロトタイプ ・・・ 説明
`istream& get(char& c)` ・・・ 入力ストリームから次の1文字を取って来て変数 c に格納する。 (補足： 引数部の"&"は、実引数に c という別名を付ける (i.e. 参照呼出)、という指示。)
`istream& get(char* s, int n, char c='\n')` ・・・ 引数 c で指定された区切り記号又は EOF までの文字の並び (但し最長 n-1 文字で打ち切り) を読み込み、最後に空文字 `\0` を付けて char 型配列 s に格納する。
`istream& getline(char* s, int n, char c='\n')` ・・・ 上の引数 3 個の `get(char*, int, char ='\n')` とほぼ同じ。こちらの関数の場合は、最後の区切り記号は捨てられる。
`istream& read(char* s, int n)` ・・・ 入力ストリームから、長さ n の文字の並び (但し途中で入力終端に至ればそこまで) を読み込み、char 型配列 s に格納する。

演算子>>を通した入力では読み飛ばされる

⟶ 場合によっては上記の関数も有用

入力の場合は、正しく入力されない可能性も...

→次のメンバ関数も有用

関数プロトタイプ ・・・ 説明
`int good()` ・・・ 何の問題も起きてない (i.e. 状態を表す変数に `goodbit` がセットの) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
`int eof()` ・・・ 入力終端に至っている (i.e. 状態を表す変数に `eofbit` がセットの) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
`int fail()` ・・・ 入力時に予期しない事態が発生した (i.e. 状態を表す変数に `failbit` がセットの) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
`int bad()` ・・・ 入力時に予期しない深刻な事態が発生した (i.e. 状態を表す変数に `badbit` がセットの) 場合は非ゼロの値を返し、それ以外の場合は 0 を返す。
(続く)

関数プロトタイプ ・・・ 説明
`int operator!()` ・・・ 入力時に予期しない事態(もしくは深刻な事態)が発生した場合は非ゼロの値を返し、それ以外の場合は **0** を返す。 正式の関数名は `operator!()` であるが、この関数は通常 ! istreamオブジェクト という風に演算子の形で使う。
`int rdstate()` ・・・ istreamオブジェクト内の状態を表す変数の値を返す。
`void clear(int st=0)` ・・・ 状態を表す変数の値を st に設定する。 (st のデフォルト値である **0** は正常値(goodbit)を表すので、istreamオブジェクト.clear() と書けば状態リセットになる。)
`istream& ignore(streamsize n = 1, int delim = EOF)` ・・・ バッファに溜まっている文字を、delimというコード番号の文字が現れるまで、もしくは文字数がnに達するまで読み捨てる。

例2. 2 (円錐の体積)　入力ミスにも対処するようにして2つの実数データ r, h を読み込み、底面の半径が r、高さが h の円錐の体積を計算して出力するC++プログラムを次に示す。

```
[motoki@x205a]$  cat -n calculateVolumeOfCone.cpp
     1 /* 2つの実数データ r と h を読み込み、             */
     2 /*       底面の半径が r、高さが h の円錐の体積   */
     3 /* を出力するC++プログラム                        */
     4
     5 #include <iostream>
     6 #include <climits>
     7 using namespace std;
     8
     9 const double PI = 3.141592653589793;   // 円周率
    10
    11 int main()
    12 {
    13   double radius, height;
```

```
14
15    cout << "底面の半径，高さ： ";
16    cin >> radius >> height;
17    while (!cin) {
18      cin.clear();
19      cin.ignore(INT_MAX, '\n');
20      cout << "[入力ミス --> 再度入力] 底面の半径，高さ: ";
21      cin >> radius >> height;
22    }
23
24    cout << "底面の半径が " << radius
25         << "，高さが "      << height << " の円錐の体積"
                                                    << endl
26         << "     = " << PI*radius*radius*height/3.0
                                                   << endl;
27 }
[motoki@x205a]$  g++ calculateVolumeOfCone.cpp
```

```
[motoki@x205a]$  ./a.out
底面の半径，高さ：  2.0 5.0
底面の半径が 2，高さが 5 の円錐の体積
     = 20.944
[motoki@x205a]$  ./a.out
底面の半径，高さ：  2.0 a
[入力ミス --> 再度入力] 底面の半径，高さ：  a
[入力ミス --> 再度入力] 底面の半径，高さ：  2.0 5.0
底面の半径が 2，高さが 5 の円錐の体積
     = 20.944
[motoki@x205a]$
```

2–9 演算子

C言語からの変更：

- C言語の拡張。 (C言語で許されC++言語で許されないものはない。)

- C言語には無かったがC++言語で導入されたものは次の通り。

◇ スコープ解決演算子 `::`

… これを使うと局所スコープで隠れた場所から大域変数にアクセスできる。⟶ **2.7**節を参照

◇ 空き領域演算子 `new`

… C言語の標準ライブラリ関数`malloc()`の様に、空き領域から動的にメモリを確保する時に使う。⟶ **2.14**節

◇ 空き領域演算子 `delete`

… C言語の標準ライブラリ関数`free()`の様に、動的に確保したメモリを空き領域に返す時に使う。⟶ **2.14**節

◇ キャスト演算子 データ型(式)

… 古いC++で導入された関数記法のキャスト。⟶ **2.6節**

◇ キャスト演算子 static_cast<データ型>(式)

… 安全な型変換の場合に使う名前付きキャスト。⟶ **2.6節**

◇ キャスト演算子 reinterpret_cast<データ型>(式)

… 安全でない型変換の場合に使う名前付きキャスト。⟶ **2.6節**

◇ キャスト演算子 const_cast<データ型>(式)

… const性を変更したい場合に使う名前付きキャスト ⟶ **2.6節**

◇ キャスト演算子 dynamic_cast<データ型>(式)

… クラス階層上位のオブジェクトを下位オブジェクトと見做したい場合に使う名前付きキャスト。⟶ **2.6節**

◇ メンバポインタ演算子 .*

… クラス内の局所的なポインタ(オフセットの様なもの, メンバポインタという)を通してオブジェクト内の要素にアクセスする時に使う。具体的に x.*y と書いた時、これは「オブジェクトxの中でメンバポインタyを辿った先のメンバ」を表す。

◇ メンバポインタ演算子 ->*

… クラス内の局所的なポインタ(i.e. メンバポインタ) を通してオブジェクト内の要素にアクセスする時に使う。具体的に x->*y と書いた時、これは「xの指すオブジェクトの中でメンバポインタyを辿った先のメンバ」を表す。

◇ typeid演算子 typeid( )

… 引数の型が表すオブジェクトを作成する。
typeid(引数).name() と書くと、これは引数の「型を表す文字列」(処理系に依存)の意味になる。

- C言語の演算子を拡張(多重定義)したものは次の通り。

  ◇ 挿入演算子 <<

  ··· 演算子の左側に`ostream`オブジェクトが来た時はストリーム出力を表す様にした。$\longrightarrow$ **2.8**節を参照。

  ◇ 抽出演算子 >>

  ··· 演算子の左側に`istream`オブジェクトが来た時はストリーム入力を表す様にした。$\longrightarrow$ **2.8**節を参照。

- C言語と少し違っているものは次の通り。

◇ 関係演算子 `<`, `>`, `<=`, `>=`

　　… `bool`型が設けられたことに伴い演算結果が **0** または **1** ではなく `false` または `true` で表されるという違いのみ。本質的な違いはない。

◇ 論理演算子 `&&`, `||`, `!`

　　… `bool`型が設けられたことに伴う違いのみ。本質的な違いはない。

◇ 条件演算子 条件 ? 式 : 式

　　… `bool`型が設けられたことに伴う違いのみ。本質的な違いはない。

◇ 後置の増分,減分演算子 `++`, `--`

　　… 優先順位が **1** つ上がった。

演算子の優先順位と結合性： 次の通り。

優先順位高 ↑

演算子	結合性
スコープ解決演算子 ::	左から右
関数の引数をくくる丸括弧 ( ) 配列添字をくくる四角括弧 [ ] メンバアクセス演算子 -> . ++ (後置) -- (後置) 関数表記のキャスト 型( ) static_cast< >( ) const_cast< >( ) dynamic_cast< >( ) reinterpret_cast< >( ) typeid( )	左から右
+ (単項) - (単項) ++ (前置) -- (前置) sizeof( ) ! ビット反転~ 間接演算子* 番地演算子& C言語表記のキャスト 空き領域演算子 new delete	右から左
.* ->*	左から右
* / %	左から右
+ -	左から右
左シフトとストリームへの挿入 << 右シフトとストリームからの抽出 >>	左から右
(次ページに続く)	

優先順位高

<table>
<tr><th colspan="2">(前ページからの続き)</th></tr>
<tr><td>< <= > >=</td><td>左から右</td></tr>
<tr><td>== !=</td><td>左から右</td></tr>
<tr><td>ビット積&</td><td>左から右</td></tr>
<tr><td>ビット排他的和^</td><td>左から右</td></tr>
<tr><td>ビット和|</td><td>左から右</td></tr>
<tr><td>&&</td><td>左から右</td></tr>
<tr><td>||</td><td>左から右</td></tr>
<tr><td>条件演算子 条件 ? 式 : 式</td><td>右から左</td></tr>
<tr><td>= += -= *= /= %= >>= <<= &= ^= |=</td><td>右から左</td></tr>
<tr><td>コンマ演算子,</td><td>左から右</td></tr>
</table>

2–10 制御構造

C言語の場合と同様の、次の制御構造が使える。

- `if`文
  `if-else`文
- `while`文
- `for`文　　(`for`ループの中の初期化式, 繰り返し式は コンマで区切った式のリストでもよい。)
- `do-while`文
- `break`文
- `continue`文
- `switch`文
- `return`文
- 条件演算子　（ 式1 ？ 式2 ： 式3 ）
- `goto`文

2–11 関数

参考 関数に対する命名規則：

Java言語においては 次の指針が一般的

◇ 主として英小文字を用いる。

◇ 2つ以上の「意味のある単語」から構成する場合は、
2つ目以降の単語の頭文字を大文字にする。 Sunコーディング規約

◇ メソッド名の場合 (関数名に相当) は、先頭文字を小文字にし、動詞(句)を表す単語列にする。
Sunコーディング規約

(構造体要素に相当)

◇ フィールドの値を調べるメソッドの名前 は「“get”＋フィールド名」にする。(e.g. getNum) 電通国際情報サービスコーディング規約

◇ フィールドの値を設定するメソッドの名前 は「“set”＋フィールド名」にする。(e.g. setValue) 電通国際情報サービスコーディング規約

—

main() の戻り値に関するC言語との違い：

- C++言語 · · · 暗黙の戻り値 **0** を仮定
- C言語 · · · 暗黙の戻り値 なし

関数引数リストが空の場合のC言語との違い：

- C++言語 · · · 引数なしの意味
- C言語 · · · 引数の個数が不明で「引数チェックは行わない」という意味

関数の暗黙の実引数：　　関数定義の際、

- デフォルト値を指定したい仮引数部については次の形で書く。

  データ型　仮引数名 = デフォルト値

- デフォルト値を指定する仮引数は、引数リストの後方に固める。

  関数名(デフォルト値指定なし , デフォルト値指定あり)

  これにより、関数呼び出しを行った場合、
  ① 前から順に実引数と仮引数の対応が取られ、
  ② 対応する実引数がない仮引数についてはデフォルト値が設定される
  という風に、曖昧さなく引数間の対応付けを行うことができる。

- 同じスコープ内でデフォルト値指定を繰り返し書くことは出来ない。

- ソースファイル上で、それを利用する関数呼び出しの前に配置する。

関数のinline宣言：

関数定義の先頭に inline というキーワードを挿入することにより、
コンパイラに

この関数の呼び出し場所に(関数呼び出しのコードではなく)
関数本体の処理コードを埋め込んでもらいたい
(インラインに展開)

という依頼を出すことができる。これに関して、

- inline宣言されている関数であっても、
コンパイラは必ずしもインラインに展開するとは限らない。

- inline宣言によるインライン展開は型安全である。すなわち、
コンパイラは引数の(構文や)データ型を認識し、
必要に応じて引数の適切な型変換を行った上でインライン展開を行う。

- inline宣言によるインライン展開は型安全である。すなわち、

  ・・・・・・・・・・・・・・・・・・・

一方、C言語で利用できる
引数付きマクロを用いたマクロ展開は 型安全ではなく、
使い方を間違えると不可解な結果がもたらされることがある。例えば、

```
#define square(x) x*x
```

という定義では、

```
1.0/square(x)  ⟶  1.0/x*x,
square(x+1)    ⟶  x+1*x+1,
square(x++)    ⟶  x++*x++
```

と展開されてしまう。また、

```
#define square(x) ((x)*(x));
```

という風に定義を間違えると、コンパイル時に使用場所での文法エラー

⇨ 引数付きマクロの使用は避け、
代わりにインライン関数を使用するのが望ましい。

関数の多重定義：

C言語 ...本質的に同じ機能でも関数ごとに別々の名前が必要。不都合

⇨ C++言語では 、次の条件を満たす場合に、
定義する複数の関数の名前を同じに設定できる。
(条件) 仮引数の型のリスト(シグネチャという)が互いに異なる。

コンパイラは、プログラム中の関数呼び出しに対して、
実引数の型のリストと定義された関数の仮引数の型のリストを見比べ、
最も適切な関数を割り出す。具体的には、

① 実引数の型のリストと合致する仮引数の型のリストが見つかれば、
その仮引数の型のリストをもつ関数を選ぶ。

② 標準的な格上げ (e.g. `float`→`double`, `bool`→`int`, `char`→`int`,...)
による型リストの合致を試みる。

③ 標準的な型変換 による型リストの合致を試みる。

④ ユーザ定義の型変換 による型リストの合致を試みる。

⑤ デフォルト実引数 の可能性を探る。

例2. 3 (多重定義,inline宣言)　引数で指定された値の最大値を返す **max()関数群を多重定義** したC++プログラムの例を次に示す。

```
[motoki@x205a]$ cat -n overloadMaxFunctions.cpp
     1 // max関数群を多重定義した例
     2
     3 #include <iostream>
     4 using namespace std;
     5
     6 inline int max(const int x, const int y);
     7 inline int max(const int x, const int y, const int z);
     8 double max(const double a[], const int size);
     9
    10 int main()
    11 {
    12   double a[5]={1.1, 9.9, 7.7, 3.3, 5.5};
    13
```

```
14    cout << "max(1,8)=" << max(1,8) << endl;
15    cout << "max(1,33,8)=" << max(1,33,8) << endl;
16    cout << "max(a,5)=" << max(a,5) << endl;
17    cout << "::max(1.1,8.8)=" << ::max(1.1,8.8) << endl;
18 }                       std内のmax(double,double)を避けるため
19                    double→intの標準型変換をしてmax(int,int)
20 // 引数で指定された値の最大値を返す関数群
21 inline int max(const int x, const int y)
22 {
23    return (x<y) ? y : x;
24 }
25
26 inline int max(const int x, const int y, const int z)
27 {
28    return max(max(x, y), z);
29 }
```

```
30
31 double max(const double a[], const int size)
32 {
33   double max = a[0];
34   for (int i=1; i<size; i++) {
35     if (a[i] > max)
36       max = a[i];
37   }
38   return max;
39 }
[motoki@x205a]$ g++ overloadMaxFunctions.cpp
[motoki@x205a]$ ./a.out
max(1,8)=8
max(1,33,8)=33
max(a,5)=9.9
::max(1.1,8.8)=8
[motoki@x205a]$
```

2–12 名前空間

- 複数の所で作られたプログラムを合わせて使おうとした時、(関数名,構造体名,クラス名,等の) 大域的な名前が衝突することがある。

⇨ 局所的でない個々の宣言を特定の名前空間に所属させ、非局所的な名前を

その名前の宣言された名前空間の名前 :: その名前

という風に明示的に表す方式が導入された。

- 標準ライブラリの名前空間 `std` を暗黙の名前空間として扱いたい場合は、プログラムの最初の方に、

```
using namespace std;
```

という using 指令を置けば良い。

- 個々のプログラマは、独自の名前空間を定義し、その中に種々の関数や変数定義, 構造体定義, クラス定義を入れることができる。例えば、

```
namespace English {
  int result;
  void printResult() {
    cout << "calculated value = " << result << endl;
  }
}
namespace Japanese {
  int result;
  void printResult() {
    cout << "計算結果 = " << result << endl;
  }
}
```

という風に名前空間 `English` と `Japanese` を定義しておけば、必要に応じて `English::result`, `English::printResult()`, `Japanese::result`, `Japanese::printResult()` を区別して使うことができる。

- 無名の名前空間の中で関数や外部変数を定義／宣言すれば、
  それらの関数や外部変数は、
  ◇ それが定義された同一ソースファイル内に限定される。
  ◇ 同一ソースファイル内では名前空間の指定無しで使用できる。

例えば、

```
namespace {
  int result;
  void printResult() {
    cout << "calculated value = " << result << endl;
  }
}
```

と定義すれば、変数resultと関数printResult()は
名前空間の指定無しで
同一ソースファイル内の残りの場所から(のみ)参照できる。

- 無名の名前空間の中で関数や外部変数を定義／宣言すれば、
  それらの関数や外部変数は、
  ◇ それが定義された同一ソースファイル内に限定される。
  ◇ 同一ソースファイル内では名前空間の指定無しで使用できる。
  例えば、

…………………

⇨ 外部変数や関数の有効範囲を同一ファイル内に制限したい場合、
- 宣言の前に`static`修飾子を付ける他に、
- 無名の名前空間を用いる

方法がある。

補足: `static`修飾子には様々な使い方がある。
その中の
「外部変数や関数の有効範囲を制限する」用法は、
他の使い方と全く異なる効果をもたらすので、
C++言語では推奨されていない。

2–13 参照宣言

- 変数宣言できる場所で、特定の変数領域に別名(エイリアス, 参照名という)を付け、使うことができる。

補足: 新たなメモリ領域の確保ではない。
⇨ 「変数」という単語を避ける。

- ブロック内、および関数外の変数宣言できる場所では 、参照名の宣言は次の形で行う。

データ型 & 参照名 = 参照先のメモリ領域を表す名前や式 ;

- 仮引数を宣言できる場所では 、
  参照先の指定はせず参照名の宣言は次の形で行う。

  データ型 & 参照名

  この場合、
  参照名の表すメモリ領域は関数呼び出し時に動的に定まる。

  ⇨ 仮引数を参照名として宣言することにより参照呼び出しとなる。

> 補足: 仮引数を参照名として宣言する場合は、
> 参照先の動的な決定を実現するために、
> 裏では対応する実引数へのポインタが関数に渡される。
>
> ⇨ 関数内で参照仮引数の値の変更を行わないなら 、
> その参照仮引数の宣言については
> `const` 修飾子も付けた方が良い。

例2. 4 (参照宣言)

```
[motoki@x205a]$  cat -n useReferenceDeclaration.cpp
     1 // 参照名を使用した例
     2
     3 #include <iostream>
     4 using namespace std;
     5
     6 int main()
     7 {
```

```
 8    int a[5] = {0, 10, 20, 30, 40};
 9    int& top = a[0];
10    int* p = a;
11                                    a[0]+2 と同等
12    cout << "top+2=" << top+2 << endl;
13    cout << "*(p+2)=" << *(p+2) << endl;
14    cout << "*(&top+2)=" << *(&top+2) << endl;
15 }                                      *(&a[0]+2) と同等
[motoki@x205a]$  g++ useReferenceDeclaration.cpp
[motoki@x205a]$  ./a.out
top+2=2
*(p+2)=20
*(&top+2)=20
[motoki@x205a]$
```

例2. 5 (参照宣言を用いたswap()関数の実装)　引数で指定された変数の内容を交換する関数swap()を実装した例

```
[motoki@x205a]$  cat -n implementSwapByReference.cpp
     1  // 参照名を用いたswap(,)の実装
     2
     3  #include <iostream>
     4  using namespace std;
     5
     6  void swap(int& x, int& y);
     7
     8  int main()
     9  {
    10    int a=0, b=5;
    11
```

```
12     cout << "a=" << a << ", b=" << b << endl;
13     swap(a, b);
14     cout << "a=" << a << ", b=" << b << endl;
15 }
16
17 void swap(int& x, int& y)
18 {
19     int temp = x;
20     x = y;
21     y = temp;
22 }
[motoki@x205a]$  g++ implementSwapByReference.cpp
[motoki@x205a]$  ./a.out
a=0, b=5
a=5, b=0
[motoki@x205a]$
```

参照呼出し

2–14 空き領域演算子newとdelete

空き領域演算子new：

空き領域(i.e. ヒープ領域) からメモリを確保するために、
C言語のmalloc(), calloc() の代わりになるものとして、
単項演算子 new が用意されている。これに関して、

- この演算子は通常次のいずれかの形で使う。
  - ◇ new データ型
  - ◇ new データ型 (確保した領域に設定する初期値)
  - ◇ new データ型 [配列の要素数を表す式]

- 上記 1∼2番目の書き方 をした場合は、
  指定されたデータ型の領域1個に相当するメモリが確保され、
  この確保領域へのポインタが演算結果の値として返される。

空き領域演算子new：

空き領域(i.e.ヒープ領域) からメモリを確保するために、
C言語のmalloc(), calloc() の代わりになるものとして、
単項演算子 new が用意されている。これに関して、

- この演算子は通常次のいずれかの形で使う。
  - ◇ new データ型
  - ◇ new データ型 (確保した領域に設定する初期値)
  - ◇ new データ型 [配列の要素数を表す式]

- 上記 2番目の書き方 をした場合は
  確保領域への初期設定も行われる。

- 上記 3番目の書き方 をした場合は、
  指定されたデータ型と配列サイズをもつ配列領域が確保され、
  この配列の先頭要素へのポインタが演算結果の値として返される。

空き領域演算子new：

空き領域(i.e.ヒープ領域) からメモリを確保するために、
C言語のmalloc(), calloc() の代わりになるものとして、
単項演算子 new が用意されている。これに関して、

- この演算子は通常次のいずれかの形で使う。
  - ◇ new データ型
  - ◇ new データ型 (確保した領域に設定する初期値)
  - ◇ new データ型 [配列の要素数を表す式]

- メモリ確保に失敗すると 、
  ◇bad_alloc型の「例外オブジェクト」が発生したり、
  ◇0 が演算結果として返されたり
  する。

補足: 例外オブジェクトが発生した場合、
発生した例外を処理するコードに制御が移り、
そこでエラー処理が為される。

空き領域演算子delete：

空き領域(i.e.ヒープ領域)から確保したメモリを開放するために、
C言語のfree() の代わりになるものとして、
単項演算子 delete が用意されている。これに関して、

- この演算子は次のいずれかの形で使う。
  - ◇ delete 確保領域へのポインタを保持する式
  - ◇ delete [ ] 確保領域へのポインタを保持する式

- 上記
  1番目の書き方 ...配列領域以外を開放する時、
  2番目の書き方 ...配列領域を開放する時

- 演算結果の値はない。

例2. 6 (空き領域演算子new,delete)　newとdeleteを使用した例

[motoki@x205a]$ cat -n useFreeStoreOperations.cpp

```
     1 // 空き領域演算子 new と delete を使用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <cassert>
     6 using namespace std;
     7
     8 int main()
     9 {
    10   int* p = new int(333);
    11   int* data;
    12   int size;
    13
    14   cout << "*p = " << *p << endl;
    15
```

変数の使い方としては不自然

```
16  //-----------------------------
17  cout << "配列サイズ： ";
18  cin >> size;
19  assert(size > 0);             指定条件を満たさないと強制終了
20
21  data = new int[size];           sizeが大き過ぎると確保失敗
22  assert(data != 0);            指定条件を満たさないと強制終了
23
24  for (int i=0; i<size; i++) {
25    data[i] = i;
26  }
27
28  cout << "配列dataの内容：";
29  for (int i=0; i<size; i++) {
30    if (i%5 == 0)
31      cout << endl;
32    cout << "  " << right << setw(5) << data[i];
```

```
33    }
34    cout << endl;
35
36    delete[] data;
37 }
[motoki@x205a]$  g++ useFreeStoreOperations.cpp
[motoki@x205a]$  ./a.out
*p = 333
配列サイズ：  12
配列dataの内容：
      0       1       2       3       4
      5       6       7       8       9
     10      11
[motoki@x205a]$
```

2–15 抽象データ型stringとvector<>

C++言語には様々な種類のライブラリが追加されている。例えば、

- ベクトルや連結リスト、集合、マップ(i.e.写像のグラフ)、スタック、待ち行列、行列(matrix)を
  - ◇ 抽象データ型のデータとして扱ったり、
  - ◇ 標準的なアルゴリズムで効率良く処理する

  ためのライブラリ、
- エラー処理をサポートするためのライブラリ、
- 数値計算のためのライブラリ、
- 並行処理のためのライブラリ、

ここでは、基本的なものを2つ選んで簡単に紹介する。

文字列を表すための抽象データ型string：

- C言語 ... 文字列を表すために、
  '\0' を最後に付加した文字の並びを char 型配列に保持。

  これに関しては、

  ◇ 標準的な演算子を用いて処理できない。
  例えば、`char s[80];` と宣言されていた時、
  `s="string";` も `"abc"+"def"` も許されない。

  ◇ 安全性が保証されない。
  例えば、`strcpy()`, `strcat()` は
  格納先配列に十分な容量があるかどうかのチェックを省略。

⇨ C++言語では 、文字列を表すための抽象データ型string

- 使う場合 は `#include <string>` とする。

- 実体は `basic_string<char>` というクラス。

例2. 7 (抽象データ型string) 抽象データ型stringを使用した例

```
[motoki@x205a]$ cat -n use_stringType.cpp
     1 // 抽象データ型stringを利用した例
     2
     3 #include <iostream>
     4 #include <string>
     5 #include <cctype>
     6 using namespace std;
     7
     8 int main()    ヌル終端文字配列
     9 {                  ↓
    10   string s1 = "abc", s2;
    11
```

```
12    cout << "Enter a string: ";
13    cin >> s2;
14    s1 += s2;                                              連接
15
16    cout << "s1 = " << s1 << endl;
17    for (int i=0; i<s1.length(); i++) {                    長さ
18      s1[i] = toupper(s1[i]);  添字番号のチェックはしない
19    }         大文字に変換
20    cout << "s1 = " << s1 << endl;
21 }
[motoki@x205a]$  g++ use_stringType.cpp
[motoki@x205a]$  ./a.out
Enter a string:  _def123
s1 = abc_def123
s1 = ABC_DEF123
[motoki@x205a]$
```

オブジェクトの列を表すための抽象データ型vector<>：

- 通常の配列では 、

  ◇ 配列の要素数はコンパイル時、もしくはnew演算子実行時に確定し、それ以降の実行の途中で変更出来ない　不便に感じることも

  ◇ 配列の要素数は属性として配列自身が保持している訳ではなく、別途パラメータ等で与えられるので、要素数の指定間違いといった単純なミスも起こり易い。

⇨ C++言語では 「動的配列」
(i.e. サイズを動的に変えられる配列)
を実現した抽象データ型として vector<要素の型>

- 使う場合 は `#include <vector>` とする。

例2. 8 (抽象データ型vector<int>)　使用例

```
[motoki@x205a]$  cat -n use_vectorType.cpp
     1 // 抽象データ型vector<int>を利用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <vector>
     6 using namespace std;
     7
     8 void showContentsOf(vector<int> v);   大きさと内容を表示
     9
    10 int main()
    11 {
    12   vector<int> v;                      要素数0の動的配列
    13
    14   showContentsOf(v);
    15
```

```
16    for (int i=0; i<10; i++)
17      v.push_back(i);                          末尾に要素を追加
18    showContentsOf(v);
19
20    for (int i=0; i<5; i++)
21      v.pop_back();                            末尾要素を削除
22    showContentsOf(v);
23
24    for (int i=0; i<v.size(); i++)           保有する要素数
25      v[i] *= v[i];
26    showContentsOf(v);
27 }
28
29 void showContentsOf(vector<int> v)
30 {
31    cout << "要素数 = " << v.size() << endl;
32    cout << "内容   = (";
```

```
33    if (v.size() > 0)
34      cout << right << setw(2) << v[0];
35    for (int i=1; i<v.size(); i++)
36      cout << ", "<< right << setw(2) << v[i];
37    cout << " )" << endl;
38 }
[motoki@x205a]$  g++ use_vectorType.cpp
[motoki@x205a]$  ./a.out
要素数 = 0
内容   = ( )
要素数 = 10
内容   = ( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9 )
要素数 = 5
内容   = ( 0,  1,  2,  3,  4 )
要素数 = 5
内容   = ( 0,  1,  4,  9, 16 )
[motoki@x205a]$
```