Ⅰ プログラミングにかかる前の基礎知識

1．プログラミング

（1）プログラミングの定義

・「プログラミング」：ＪＩＳでは、プログラムの設計、記述、修正及び試験。

・「プログラム」「計算機プログラム」：動作を指定する一覧表又は計画表。処理作業の手続きの一覧。ＪＩＳではモジュールの論理的集まりとしている。

「計算機プログラム」という用語は、テレビ放送のプログラム等一般用語と区別したいときに使用する。「プログラム設計」は、計算機の処理作業の手続きの一覧である「計算機プログラム」の設計、すなわち「プログラミング」と同じ意味である。

（2）プログラミング

プログラミングは、「計算機プログラム」（以下、単に「プログラム」と記述）の構成単位であるモジュール（プログラム単位）の①設計②コーディング③単体テストのことである。このモジュール設計からモジュールの単体テストまでを行う人をプログラマと呼ぶ。

（3）モジュール

プログラムを機能によって分割したプログラム単位がモジュールである。モジュール毎に開発を進めてゆくことにより、①考える範囲が小さいので作成しやすい、②複数のプログラマが並行して同時に作業を進めることができる、③モジュール毎の単体テストを行うことによりデバッグが容易になる、等のメリットがある。このように要求されている機能を明確にし、洗い出した機能単位毎にモジュールを対応させ、モジュール構造を明確にした上で行うプログラミング方法がモジュラープログラミングである。

プログラミングに際し、顧客の要求を明確にした上で、要求されている機能をモジュール単位に分割する工程が「上流工程」である。上流工程では適切な仕様記述言語を導入しプログラミング段階を自動化する試みも着実に進んでいる。しかし、仕様記述言語も人工言語の一種であることにかわりはなく、将来的にもプログラミングの技能がソフトウエア技術者の基本的な技能であることにかわりはない。

（4）コーディング

モジュールに組み込む機能の構成・構造を記述したプログラム仕様書の内容をプログラム言語の命令で記述する作業がコーディングである。プログラム言語の命令で記述されたプログラムが原始プログラムである。

（5）プログラム言語

プログラム言語はコンピュータに指示を与えるために作られた人工言語である。プログラム言語は低水準言語と高水準言語に分類される。低水準言語と高水準言語の差異はハードウエアに密着した言語であるか人にとってわかりやすい言語であるかの差である。

低水準言語はＣＰＵのハードウエアに密着しているため基本的にＣＰＵメーカによって作成される。ＣＰＵに対して直接指示を与えることが可能であり、ＣＰＵが直接理解するのはメモ

リ上に展開された機械語だけである。ビットパターンとして表現される機械語と、機械語と一対一に対応するニモニックの集合であるアセンブラ言語が低水準言語である。ニモニックはＣＰＵの一動作を簡略化された英語様のキーワードで表したものである。低水準言語はＣＰＵのハードウェアに依存し、異なるＣＰＵ間での互換性はない。低水準言語を低レベル言語と表現することもある。

高水準言語は、多くの場合簡略化された英語的な命令でコンピュータに対する指示を記述する。低水準言語と異なりある一定レベル以上の機能を持つＣＰＵであれば特にＣＰＵのハードウエアに依存しない。高水準言語の多くは低水準言語と同様に処理の手順を記述していく手続き言語である。しかし処理の手順よりは「何がしたいのか」の記述に重点を置いた非手続き的な言語も生まれている。ＦＯＲＴＲＡＮ、ＣＯＢＯＬ、ＰＬ／ＩやＣは手続き的な高水準言語でPrologやSmalltalkは非手続き的な側面を持つ高水準言語である。特にSmalltalkに代表されるオブジェクト指向言語は注目を浴びつつあり、開発の現場ではＣをオブジェクト指向言語として拡張したＣ＋＋が開発の主流となりつつある。

（６）原始プログラム

アセンブラやＣ等のプログラム言語で記述されたプログラムが原始プログラムで、ソースプログラムとも言う。紙の上に通常の筆記用具で書かれたものも原始プログラムである。最終的に、原始プログラムはコンピュータ上で実行可能な目的プログラムに置き換えられる。目的プログラムはオブジェクトプログラムとも言う。

目的プログラムはコンピュータの構成要素であるＣＰＵとオペレーティングシステムによってその形態が決定される。目的プログラムは多くの場合オペレーティングシステムから起動される。オペレーティングシステムから起動可能な目的プログラムはファイルとして存在し、目的ファイルないしオブジェクトファイルと呼ばれる。

オブジェクトプログラムやオブジェクトファイルのオブジェクトとオブジェクト指向言語のオブジェクトは意味が異なる。

原始プログラムを目的プログラムに翻訳する道具、ソースプログラムからオブジェクトプログラムを生成する道具を言語プロセッサと総称する。オブジェクトプログラムを生成せずにソースプログラムを逐一解釈しながらコンピュータに対する指示に置き換えて動作するタイプの言語プロセッサがインタプリタである。

インタプリタに対し、原始プログラムを一括して目的プログラムに翻訳するのがトランスレータである。インタプリタもトランスレータもコンピュータ上で動作するプログラムである。トランスレータを翻訳プログラム、インタプリタを解釈実行プログラムということもある。これらのプログラムに原始プログラムを処理させるには原始プログラムもコンピュータ上のファイルとして存在する必要がある。ファイルとしての原始プログラムを原始ファイル、ソースプログラムのことをソースファイルという。

ソースファイルの記述はワードプロセッサでも可能である。ただプログラミングに際してワードプロセッサの大部分の機能は不要であり、場合によってはトラブルの種になるので、通常

はソースファイルの記述にはエディタを用いる。エディタはプログラマにとって日常的に使用する道具である。高級なエディタとしては元々ＵＮＩＸ上で動作していたＥｍａｃｓやＶｉエディタが、パーソナルコンピュータの高機能化に伴いパーソナルコンピュータ上でも動作するようになってきている。Ｗｉｎｄｏｗｓに付属のメモ帳もエディタであるからソースファイルの作成は可能であるが最終的には自分の気に入ったエディタを決め自由自在に操作可能になるようにすべきである。

（７）言語プロセッサ

トランスレータはコンパイラと表現することもある。インタプリタとトランスレータの差異は一行一行翻訳して実行するかまとめて翻訳しオブジェクトファイルを生成するかにあり、言語の文法の差異を表しているのではない。Ｂａｓｉｃのようにインタプリタに向いた文法を持つ言語も確かにも存在するが、若干の向き不向きがあるにすぎない。Ｂａｓｉｃのコンパイラも存在すれば、コンパイラが主であるＣのインタプリタも存在する。

低水準言語であるアセンブラ言語の言語プロセッサがアセンブラである。言語プロセッサはプリプロセッサ、コンパイラ、アセンブラ、リンカ等で構成される。プリプロセッサは拡張された言語の仕様を受け持ち、拡張仕様の部分を高水準言語に置き換える動作をする。言語プロセッサの中心であるコンパイラは高水準言語を機械語ないし中間言語に置き換える。リンカはコンパイラの生成したファイルをつなぎ合わせ、一つの実行可能ファイルとしてのオブジェクトファイルを生成する。多くのコンパイラはアセンブリ言語のファイルを出力可能である。

プログラマ　→　ソースファイル　→　オブジェクトファイル
エディタ　Ｃコンパイラ

（８）コーディングスタイル

コーディンクを行うには使用する高水準言語の文法とコーディンクスタイルを身につける必要がある。プログラムの書き方がコーディングスタイルである。これから学んでゆくＣは「フリーフォーマット」でタブや改行に区切り記号としての強い意味はない。しかしコンパイラに理解できさえすれば良いというコーディングスタイルは誤りである。プログラムは人工言語で記述されているが文章であることに違いはない。またその良い文章の条件は読みやすさである。これと同じで良いプログラムの重要な条件は他のプログラマにとっても読みやすい記述がなされている事である。読みやくするためには、①インデントを用いてブロックを明確にする、②データの入れ物である変数名の決め方に留意する、③適宜コメントを入れておく、等に留意したコーディンクスタイルを身につける必要がある。ここでの学習の中心はまずＣの文法の習得で、②や③については以後余り触れないがその重要性を忘れないでほしい。

良いプログラム＝可読性の高いプログラム

（９）設計と単体テスト

モジュールの入出力値と機能を明確に記述しモジュール設計書にまとめる作業がモジュール設計である。モジュール設計書さえあれば他の資料（仕様書）がなくてもコーディンク可能にしておく必要がある。モジュール設計書が完全に作成されていれば単体テスト用のテスト計画書も作成可能である。設計の時点で同時にテスト計画書も作成される。

モジュールに要求されるロジックが複雑な場合、制御構造の概略を設計時点で決めることになる。その場合流れ図等の図式や簡略化した自然言語による疑似コードを用いてロジックを理解しやすいよう表現する。

テストの結果発見された不具合が「バグ」である。このバグを取り除く作業、不具合を直す作業が「デバッグ」（「虫取り」と呼ぶ人もいる）である。デバッグには実際に実行プログラムを動かす前に行う「机上デバッグ」と実行プログラムを動かしながら行う「実機デバッグ」がある。

机上デバッグの主たる目的はアルゴリズムの検証である。最低限、こんな値を入力したらこんな値が出力されるはずであるという入出力表を作成しておくべきである。

プログラムはプログラマの思った通りに動作するものではなく、ソースファイルに記述された通り動作するものである。机上デバッグの時に作った入出力表を中心に、実際にプログラムが設計した通りに動作しているか検証するのが実機デバッグである。

プログラムはソースファイルに記述された通りに動作する。

２．Ｃ言語

（１）Ｃ言語の歴史

Ｃ言語は、１９７２年、アメリカＡＴ＆Ｔベル研究所の Dennis M. Ritchieによって設計された。Ｃ言語は、当時彼が開発していたオペレーティングシステム（ＯＳ）であるＵＮＩＸの開発用言語として開発された。当初Ｃ言語は開発者である Ritchie や Brian W. Kernighanのグループ内でのみ用いられていたが、Ｃ言語自体の持つ良さが多くのプログラマに認められた事とＵＮＩＸの普及を背景に広範囲のプログラムに用いられるようになった。構造化プログラミングを完全にサポートした高水準言語でありながらハードウェアレベルの記述が可能である。かつ高度な拡張性をもつシステム記述言語としてビジネスアプリケーションの開発から制御用プログラムの開発まで広く利用されるようになった。パーソナルコンピュータやワークステーションを中心として利用可能なＣコンパイラが存在しないコンピュータはないと言っても言い過ぎではないような状態になった。

広範囲のプログラミングに用いられるようになったＣ言語であるが、当初Ｃ言語の文献としては「Ｋ＆Ｒ」本と呼ばれる”The C Programming Language”しか存在せず、”The C Programming Language”自体が既にＦＯＲＴＲＡＮ等、何らかのプログラミング言語でプログ

ラミング可能なプログラマに対するＣ言語の紹介を主たる目的として書かれていたこと、Ｃ言語の柔軟さが災いして使用するコンパイラによって少しずつ仕様が異なる「方言」ができてしまった。そこで、１９８９年末ＡＮＳＩ（American International Stamdard Institute：米国国内規格協会）によって規格がまとめられ、承認された。さらに、１９９０年にはＩＳＯ（International Standard Organization：国際標準化機構）規格が制定され、これを受けて日本でも１９９３年、ＪＩＳ　Ｃが制定され、情報処理技術者試験のプログラミング言語としても採用された。

現在では多くのＯＳに対応しているＣ言語の一番ベースとなる環境はＵＮＩＸである。例えばマイクロソフト社の Windows やＩＢＭのＯＳ／２は大文字と小文字を区別しないがＵＮＩＸは大文字と小文字を区別する。Ｃ言語では大文字も小文字も利用可能でかつ大文字と小文字は区別される。

Ｃ言語では’Ａ’と’ａ’は別の「もの」を示す。

１９９０年代後半になってからＧＵＩ（グラフィカルユーザインタフェース）を持つパーソナルコンピュータ上のアプリケーションプログラムの開発を中心に、開発言語の主力は、Bjarne　Stroustrup 博士によって開発されたＣ＋＋言語に移りつつある。Ｃ＋＋言語はオブジェクト指向言語であるが、Ｃ言語を完全に包括している。これはＣ言語の学習が無駄にならないことを意味すると同時に必須であることを意味する。

Ｃ言語はＣ＋＋言語の基盤でもある。

（２）Ｃ言語の特徴

Ｃ言語と同等の機能を有するプログラム言語として Pascal がある。ＩＳＯでは Pascal 表記されるが、ALGOL ６０をもとにして１９７１年 Niklaus　Wirth によって発表されたプログラム言語である。ALGOL は１９５８年から１９６２年にかけて開発されたプログラム言語で、現在はほとんど利用されていないが、Pascal に限らずＰＬ／Ｉ、Ｃ、Ａｄａ等はほとんど全てのプログラム言語にその影響が見られる。ALGOL で最初に実装された概念としてブロック構造がある。ブロック構造とは何らかの機能単位を実現するための一連の宣言や文をひとくくりにする構造である。

ブロック構造を構成する各ブロック間には、列挙と包括の二通りの関係がある。ALGOL 以前の高水準言語でも、「包括関係」を条件分岐や繰り返しを表すステートメントに注意することで読み取ることは可能である。しかし ALGOL や Pascal ではbegin～end、Ｃ言語では {～} のようにいつも同じキーワードか記号でブロックの境界が示される。ブロックの境界の明示、特に３つの基本制御構造のうちの選択構造と反復構造におけるifブロックelseブロック及び繰り返し内容を示すブロックの明示は、可読性の向上に大きく貢献する。

今述べた選択構造と反復構造に加えて連続構造の３つが基本制御構造である。３つの基本制御構造で全てのアルゴリズムの記述が可能である。特に、反復構造を明示するステートメントを持たない低水準言語では、ＧＯＴＯ文に相当するステートメントの使用が避けられない場合があるが、ALGOL 以降の言語ではＧＯＴＯ文の使用は不可避ではない。ＧＯＴＯ文の無節操な使用はブロック構造を不明確にし可読性の低下を招く。

ブロックは何らかの機能単位に対応する存在である。データの流れに着目すると、ブロックは処理のもとになるデータを受け取り、処理結果を出力する。すなわち一つの入り口と一つの出口を持つ形に記述可能な存在である。一つの入り口一つの出口を持つ各ブロックは他のブロックとのデータの授受の関係が捉えやすくブロック単位での試験・デバッグが可能になる。このようにブロック構造を明確にしたプログラムの記述により可読性の向上、機能ブロックの独立性の向上によるデバッグの効率化を目指したプログラミング技法が構造化プログラミングである。Ｃ言語は ALGOL に代表される高級言語と同じく構造化プログラミングに対応した言語である。

特に独立性の高い機能モジュールは、低水準言語ではサブルーチンとして記述される。Ｃ言語では、全てのモジュールは関数として記述される。関数もブロックの一種であるので関数名に続けて｛～｝を置いたブロック中に関数の本体を記述しモジュールの動作を定義する。プログラムにおいて多くの場合、巨大で複雑な問題も小さな単純な問題の集合として捉えることができる。求める複雑な機能も単純な部品としての機能の集合として記述が可能である。Ｃ言語では各部品は部品としての独立した構造を保ったまま関数として記述が可能である。

各モジュールは、入り口に相当する変数にデータをセットし、処理を行い、結果を出口に相当する変数に設定する。あるいは関数の戻り値として結果を返す。データの入れ物である変数の取り扱い可能な桁数、あるいは内部フォーマットを指定するのがデータタイプである。Ｃ言語では変数も関数も名前によってアクセス可能な存在で全ての存在はデータタイプが明確になっている必要がある。プログラムを現実世界のシュミレーションとして捉えた場合、存在の種類に対応するデータタイプの種類が多い方がプログラムの記述時に有利なことは容易に想像可能であろう。豊富なデータタイプを持つこともＣ言語の特徴である。更に複数のデータタイプを組合わせて任意のデータ構造の定義が可能な構造体、共用体の機能も用意されている。

変数はメモリのアドレスに対応する存在である。多くの高水準言語では変数を通じてのメモリ操作だけで事足れりとしている。コンピュータシステムにおけるランダムアクセスメモリの特徴はその均質性である。２０００番地に位置するメモリセルと３０００番地に位置するメモリセルの間に差異は存在しない。しかし、ＯＳの記述等、ハードウェアに近い立場で考えた場合ＣＰＵの特権モードでのみアクセス可能なアドレス空間とそうでないアドレス空間の扱いは異なるのである。そのためＣ言語はその基本データタイプの一つとしてポインタ型というアドレスを扱うためのデータタイプを持つ。通常は低水準言語でしか行えないハードウェアに近い操作・低水準な記述もＣ言語では高水準言語の高い可読性を保ったまま可能である。ハードウェアの制御からビジネスアプリケーションの記述まで対応可能な豊富な機能を持つのもＣ言語

の特徴である。

更に、これだけ豊富な機能を持ちながらその実装において、ライブラリの大幅な採用により言語仕様自体を非常にコンパクトにまとめ上げる事に成功している。言語仕様のコンパクトさとライブラリの採用によりＣ言語は非常に広範囲なコンピュータシステム上で利用可能になっている。言語仕様のコンパクトさは機種間のコンパイラの移植作業を容易にする。コンパイラ本体とライブラリ、リンカの分離によりコンパイラ本体は基本的にはＣＰＵにしか依存しない。同一ＣＰＵであっても入出力機能はＯＳに依存する。ＯＳに依存する機能はＣ言語ではライブラリとして実装される。ライブラリはライブラリ関数の集合で、同じ事をするライブラリ関数には同じ名前を付けることによりＯＳの差異を吸収している。このように機種依存部分をライブラリで持つことによるソースコードレベルの高い互換性を持つのもＣ言語の特徴である。特に，ＡＮＳＩ等で規定された標準関数についてはほぼ完全な互換性が実現されている。であるからＣ言語の学習に際しては、代表的な標準関数の扱いも学ぶ必要がある。

構造化プログラミングとは読みやすく理解しやすいプログラムによりバグの発生率を下げ開発効率の向上を目的としたプログラミング手法である。構造化プログラミングの特徴は基本制御構造とＧＯＴＯレスと入り口と出口が一つのモジュール構造の三つである。このうちＧＯＴＯレスは三つの基本制御構造と追加機構を用いて入り口一つ出口一つのモジュール構造を実現した結果そう見える、あるいはＧＯＴＯを用いなくてもプログラミングできるという意味である。

Ｃ言語は、連続構造と選択構造と反復構造の三つの基本制御構造に加えて後判定型反復構造と多分岐構造に対応したステートメントを持っている。ちなみにＧＯＴＯに対応したステートメントも持っている。構造化プログラミング対応のステートメントだけを用いてプログラムを記述したとしても各制御構造を単独で理解できなかったり、制御構造単位で上から下へ連続して読んでいけないのでは構造化されたプログラムとはいえないのである。

（３）構成要素

①ステートメント

コンパイラに入力可能な任意の表現がステートメントである。ステートメントはソースランゲージの命令と注釈と翻訳操作を指示する指示文からなる。ステートメントを「文」ということもある。注釈は自然言語で注釈を加えるためのものでソースプログラムを読む人のためのものである。「/*」で書き始め「*/」で注釈の終わりを示す。翻訳操作を指示する指示文はＣ言語の場合多くはプリプロセッサに対する指示で#includeや#defineのように#で書き始める。

ステートメント（文）：＝　ソースランゲージの命令
　　｜注釈
　　｜翻訳操作を指示する指示文

ソースランゲージの命令は、制御文、複文、式文、ラベル付き文からなる。式文とは「；」

セミコロンで終了する。ラベル付き文はその先頭に「コロン」：で区切られたラベルを持つ。複文は｛｝（中括弧）で括られ、その中に複数の文を含む。中括弧で囲まれた範囲をブロックと呼ぶこともある。制御文は制御構造を示す if,elseや switch、for、do、while、goto、continue、break、returnのことである。

ソースランゲージの命令　：＝　制御文｜複文｜式文｜ラベル付き文

ステートメントの並びがプログラムを構成する。各文は上から下へ、左から右へ順番に実行される。制御文だけが実行の順番を変更する。

②式文

式のみから構成される文が式文である。式は日常用いられる算術式のように定数や変数を算術演算子（加法演算子「＋」、減法演算子「－」、乗法演算子「＊」、除法演算子［／］）で結んだものであり、小丸括弧「(」「)」で式をグループ分けしたものをさらに演算子で連結することもできる。Ｃ言語では算術演算子に加え論理演算子を用いた論理式（真か偽の２値を扱う）も式であり、記述される場所により用いている演算子に無関係に条件式として評価される。また式の項として関数名の記述も可能である。

日常生活では、式はその計算結果を用いるだけのことが多い。Ｃ言語でもイコール「＝」を用いて「変数名＝式」と記述した場合、式の計算結果が左辺の変数に収納される。あるいは、「変数名＝」のない、「式」とだけ記述されても、その場所によりその計算結果が評価される。多くの場合制御構造の条件を表す式（条件式）として丸括弧に挟まれた形で記述される。

Ｃコンパイラは式を必ず評価しようとする。必ず計算結果を求めようとするが、その評価結果・計算結果は「変数名＝」と記述がなければ特に変数の中に値を残すことはしない。Ｃ言語の式と日常生活の式の異なる点である。厳密にはｘ＝３＋５；という文があった場合Ｃコンパイラは式の＝で区切られた固まりを右から見てゆき３＋５を評価し、評価結果の８を変数ｘにいれて、さらにｘを変数ｘだけの式として評価し８という値を得る。この場合最終的に得られた８という値は何も利用されずに棄てられてしまう。これは一見無駄なように見えるが、Ｃ言語独自の柔軟な記述のもとである。

Ｃ言語では、「式」は評価の対象である。

式だけの記述は、特に関数呼び出しの記述に顕著で、数学と同じように y=fun(3);と記述することも多いが、単に fun(3);と記述されることも多い。funは関数名で 3 は引数を表す定数式である。fun()がどんな処理（計算）を行うのかは別の場所で定義されているのだが、y=fun(3);の場合 3 という値をもとに何らかの計算（処理）を行いその計算結果（「戻り値」という）を変数 y に代入し（入れて）、さらに y も変数名だけの式としてその内容の値をいったん評価するのは先に述べた通りである。fun(3);の場合値 3 をもとに fun()で定義された通りの処

理（動作）を行い、その計算結果はもし存在しても、代入する場所が指示されていないので無視されてしまう。

ｙ＝ｆ（ｘ）；　も　ｆ（ｘ）；　も　どちらも関数ｆ（ｘ）の呼び出しを行う。

③複文

制御文で繰り返し実行される文の連なりや関数の本体の定義を示す文の連なりは、中括弧で囲まれた複文（ブロック）として示される。while(条件式){式１；　式２；　式３;}や if(条件式){式１；　式２；　式３;}のような場合は「条件式が成立している間は、以下の式１、式２、式３を繰り返す。」あるいは「条件が成立した場合、以下の式１、式２、式３を実行する。」という意味であるから複文（ブロック）に名前をつける必要はない。これに対して関数の本体の定義では、関数名(引数名){　式１；　式２；　式３；　return(式);}のような記述になる。これは関数名で指定される関数が呼ばれた場合、引数として与えられた式の評価結果（計算結果）を引数名で示される変数に入れ、式１、式２、式３と実行し、その結果をもとに return 文中の式を評価し、その計算結果を関数の戻り値として返し、関数呼び出しの次の処理を行うことを示している。関数の本体の定義を示すブロックの場合、そのブロックに関数名()という名前がついていると考えられる。

Ｃ言語では名前を持つブロックはあと２種類ある。一つは３人分の英語のテストの成績の集合を｛88,67,93｝のように表現する場合、これは配列に対応する。もう一つは主要３科目の成績は{英語の点数,国語の点数,数学の点数}であるというような場合、これは構造体に対応する。この二つの場合、前者はその値を設定（代入）する配列変数の名前が、後者は指定された構造を持つ構造体の名前がそれぞれのブロックの名前に相当する。

ブロックはデータの集合

　　レコードの構造

　　関数の本体

　　制御文の繰り返し内容等を示す。

④名前

Ｃのプログラムの基本要素である式は、定数、変数、関数、演算子で構成される。Ｃ＋＋言語では定数も定数名として名前で扱うことが可能であるが、Ｃ言語では変数は変数名、関数は関数名で表現され操作の対象となる。さらにレコードの構造を定義する構造体や共用体も構造体名、共用体名として名前で表現され、初めて使用可能になる。多くの場合、これらの名前を決めるのはプログラミングの最初の作業となる。

Ｃ言語のプログラマは名付け親？！

プログラムの動作は、実行する動作を指定する「文」(「ステートメント」) を書かれた順に実行することによって実現する。C言語のプログラムは文と宣言から構成される。(注、C++では宣言は文に含まれる。) 宣言は識別子の解釈と属性の確立を行う、すなわちオブジェクト (変数) や関数を使うための準備である。識別子・識別名・名称・名前、通常この4者は同じ意味に用いられ、オブジェクト (変数) 、関数、構造体・共用体・列挙型のタグやメンバー、あるいは、typedef、ラベル、マクロ、マクロ仮引数を識別するために付けられる名前を意味する。

C言語の識別子は下線、英小文字、英大文字で始まり、これらと数字の組み合わせが続く綴りで長さの制限はないが、有効先頭文字数は制限されている。識別子においても大文字・小文字は区別される。予約語、16 進定数、浮動小数点定数は識別子として扱われない。現在のところ$や@を含む綴りや漢字を識別子として扱えるかどうかは処理系依存である。

下線に続き大文字か下線で始まる綴りはライブラリ中等で用いられる予約識別子として扱われる。下線で始まる綴りはユーザ定義の識別子として用いても、ライブラリ中で用いられていなければコンパイルエラーにはならないが、システム (OS) に相当する操作を記述している関数以外では用いない方が良い。

初心者の気づきにくい誤りの例として先頭が数字の綴りの識別子がある。

(4) 式

式には一次式、後置式、前置単項式、キャスト式、ポインタメンバ式、乗除式、加減式、シフト式、不等式、等非等式、ビット積式、ビット差式、ビット和式、積結合式、和結合式、条件式、代入式、順次式の種類がある。このように多くの種類があるが、演算子とそのオペランドの並びで値の計算方法を指定したり、オブジェクトや関数を示したり、副作用を生じたり、これらの複合した効果を生ぜしめるのが式である。代入演算子で代入操作を行った場合オブジェクト (変数) の値が変更される。これはオブジェクトの値の変更という副作用をもたらしたことになる。副作用は式に限らず、演算子や関数の作用において副次的に実行環境の変化を起こすものが副作用である。ファイルの変更や標準出力関数による画面への出力も副作用の例である。

①リテラル

リテラルは定数のことであり直定数・直接定数・即値と表現されることもある。リテラルは数値定数と文字定数と文字列定数 (文字列リテラル) に分類される。C言語ではシングルクオート「’」で囲めば文字定数、ダブルクオート「”」で囲めば文字列定数となる。数値定数と文字定数は一つの値であり、文字列定数は複数の値の組であり配列に対応する存在である。

3.14 や 12 は数値定数 'K'は文字定数　　"Hello"は文字列定数

多バイトコードに対応して、文字定数は通常文字定数とワイド文字定数、文字列リテラルは通常文字列リテラルとワイド文字列リテラルに分類される。コンパイラが多バイトコードに対

応していればワイド定数の扱いについては接頭文字Ｌを付ける以外通常定数と余り差異はない。

数値定数は整数定数と浮動小数点整数に分類される。整数定数は１０進表現以外に８進表現と１６進表現が可能である。０以外の数字から始まり０，１～９の範囲の数字列で表現されるのが１０進数定数、０で始まり０，１～７の範囲の数字列で表現されるのが８進数定数であり、０Ｘまたは０ｘで始まり０、１～９、ａ～ｆ、Ａ～Ｆの数字列ないし文字列で表現されるのが１６進数定数である。

981 は１０進数定数　071 は８進数定数　　0xf91 は１６進定数

浮動小数点定数は１０進表記で表現された小数部を含む数値定数である。コンピュータの基本データ形式は整数である。仮数部と指数部の二つの整数の値の組で実数を表現するのが浮動小数点である。仮数部で有効数字列を、指数部で小数点の位置を示す。小数点を含む１０進数は浮動小数点定数として扱われる。あるいは１２３を１．２３×１０の２乗という意味で明示的に 1.23E+02 と書くこともできる。Ｅは大文字でも小文字でも良い。

3.14、.15、3.02e+3、.99e-2　は、浮動小数点定数

浮動小数点といっても無限桁の完全な実数を扱えるわけではなく、有限桁で処理される。これは整数でも同様である。扱う桁の多さを浮動小数点では単精度、倍精度及び拡張倍精度という言葉で表す。浮動小数点では単精度を意味するのにｆまたはＦ、拡張倍精度を表すのにｌまたはＬを数字列の末尾につける。これらを浮動小数点接尾語という（特に何もつけないのは倍精度である）。整数型ではｌまたはＬを長接尾語、ｕまたはＵを無符号接尾語として用いる。無符号接尾語がつくと符号ビットも他のビットと同じように操作する。単精度と拡張倍精度あるいは長接尾語のあるなしの効果については使用するコンパイラの影響を受ける。

文字定数は下記のアスキーコード表で対応する整数値に置き換えられる。コンピュータの基本データ形式は整数である。アルファベットや数字といった半角文字は内部ではこのような８ビット長（１６進数二桁）の整数として扱われている。漢字に代表される全角文字は正確には２バイト（１６ビット）長で扱われ、文字定数としてはワイド接頭辞のＬをシングルクオートの前につけて L'漢'のように表現される。通常の文字定数では'A'は整数定数の 0x41 と同じ値を意味する。ちなみに数字の０は'0'で 0x30 と同じ値である。

アスキーコード表

上位→ 下位↓	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0：				0	@	P	`	p								
1：			!	1	A	Q	a	q								
2：			"	2	B	R	b	r								
3：			#	3	C	S	c	s								
4：			$	4	D	T	d	t								
5：			%	5	E	U	e	u								
6：			&	6	F	V	f	v								
7：			'	7	G	W	g	w								
8：			(	8	H	X	h	x								
9：			)	9	I	Y	i	y								
A：			*	:	J	Z	j	z								
B：			+	;	K	[	k	{								
C：			,	<	L	¥	l	|							
D：			-	=	M	]	m	}								
E：			.	>	N	^	n	~								
F：			/	?	O	_	o									

注）日本製のパーソナルコンピュータでは独自の仕様で上位の空いている所に半角のカタカナ等が定義されている事が多い。

文字列定数は各文字が文字定数としてメモリ中に置かれる。文字定数との違いは末尾に文字列の末尾を示すためにヌル文字(0x00)が追加されることである。例えば文字列定数"ABC"がメモリ中の 1000 番地からに格納されたとする。1000 番地には'A'に対応する 0x41 が 1001 番地番地には'B'に対応する 0x42 が、1002 番地には'C'に対応する 0x43 が収納されさらに末尾を示すために 1003 番地には 0x00（ヌル文字）が追加される。

１０００番地　　'A'

１００１番地　　'B'

１００２番地　　'C'

１００３番地　　0x00（ヌル文字）

文字列定数はメモリ中に連続して置かれ、末尾にヌル文字が追加される。

②変数

変数（バリアブルともいう）は、値を記憶する入れ物である。変数の利用が可能なのが高水準言語、利用できないのが低水準言語という区分も可能である。変数は名前により他の変数と区別・認識されるので識別子ともいう。変数の種類、入れ物としてのサイズを決めるのが型である。Ｃ言語では変数は有効な範囲が特定可能で記憶期間により寿命が定められる。「変数名＝」記述は代入操作を意味し、変数の値を変える操作である。

③演算子

［ ］、（ ）、.、－＞

＋＋、－－、＆、＊、＋、－、／、％、

~、！、＜＜、＞＞、

＜、＞、＜＝、＞＝、＝＝、！＝、

＾、｜、＆＆、｜｜、？：、sizeof

＝、＋＝、－＝、＊＝、／＝、＜＜＝、＞＞＝、＆＝、＾＝、｜＝、

，、＃、＃＃、define

これらは全てＣ言語の演算子である。代表的なものは加減乗除を表す（＋－＊／）の四則演算子であろう。マイナス３（－３）のマイナス（－）と７引く４（７－４）の引く（－）は同じ－の記号を用いるが－３のマイナスは負符号演算子でオペランドを一つしか取らない単項演算子の一種で７－４の引くはオペランドを二つ取る二項演算子の一種である。このように同じ記号で異なる演算を示すものもある。四則演算では加減乗除の順に演算子の優先順位が上がってゆくが、Ｃ言語でもこれら全ての演算子に数学に近い優先順位がある。演算子の利用に際しては優先順位に加えて結合規則と評価順序を考慮する必要がある。

ｘ＝ｙ＝ｚ＝７のように同じ優先順位の演算子（この場合は代入演算子）が入れ子になっている場合の結合方向が結合規則である。代入演算子ではｘ＝（ｙ＝（ｚ＝７））という結合をしていることになる。ただしｘとｙ＝（ｚ＝７）のどちらが先に評価されるかは不定である。

演算子のオペランドが評価される順番が評価順序である。関数呼び出し演算子（）、積結合演算子＆＆、和結合演算子｜｜、条件演算子？：や順次演算子，以外はオペランドの評価順序は不定である。Ｃ言語ではオペランドの評価順序の影響がでるような記述はしてはならない。ちなみにオペランドとは演算子が作用する実体、演算の対象となる「もの」のことである。

（５）型

変数等をメモリ空間に割り振るときのサイズ、形式の種類が型である。変数名や関数名等の名前を持つオブジェクトは式で使用される前に、型指定子からはじまる宣言によって型指定をされている必要がある。Ｃ言語の型は基本型と派生型に分類される。基本型は、文字型、整数型、浮動小数点型を示す char、int、double(float)の三つの予約語のみで示される型である。派生型は、配列型、要素型、構造体型、共用体型、関数型、ポインタ型の６つで、オブジェクト型、関数型、不完全型の組み合わせによって誘導される型である。

有効な変数のように、宣言で指定された型に応じたサイズと値を持つ（データ記憶領域を割

り振られている）もので、名前で直接参照可能な存在がオブジェクトである。ここでは定数、関数、演算子、ラベル、文はオブジェクトに含めない。サイズの確定しているオブジェクトを示す非参照型がオブジェクト型である。オブジェクト型に対してサイズの確定していないオブジェクトを示す非参照型が不完全型である。関数を示す非参照型が関数型である。この３つの型は排他的であり、この３つの型でポインタ型の対象である被参照型を構成する。

void を含まない型指定修飾子の並びがオブジェクト型である。void を含む型指定修飾子の並びや内容不明の構造体型や共用体型が不完全型である。サイズの確定しない配列型は不完全型である。関数型は型指定修飾子の並びや直接抽象関数宣言子で関数を示すのが関数型である。（注：ここで用いているオブジェクトはオブジェクト指向で扱われているオブジェクトとは同じではない。）

型指定子や型修飾子が型指定修飾子である。型を修飾する宣言指定子 const と volatile が型修飾子である。void、char、short、int、long、float、double、signed、unsigned、構造体指定子、共用体指定子、列挙指定子、typedef　名　が型指定子の構成要素である。これらの構成要素を組み合わせ下記の型指定子を構成する。

void	値の空集合指定をする型、不完全型である。	
char	文字型	
signed char		
unsigned char		
short	有符号整数型	char 以上 int 以下のサイズ
signed short	有符号整数型	
unsigned short	無符号整数型	
short int	有符号整数型	char 以上 int 以下のサイズ
signed short int	有符号整数型	
unsigned short int	符号整数型	
int	有符号整数型	
signed int	符号整数型	
unsigned int	無符号整数型	
long	有符号倍長整数型	int 以上のサイズ
signed long	符号倍長整数型	
unsigned long	無符号倍長整数型	
long int	有符号倍長整数型	
signed long int	符号倍長整数型	
unsigned long int	無符号倍長整数型	
float	単精度浮動小数点型	
double	倍精度浮動小数点型	

long double
構造体指定子
共用体指定子
列挙指定子
typedef　名

通常 short と long は int と組み合わせて使用される。short、long のみの記述は int の省略と見なされる。volatile　int は揮発性有符号整数型で、volatile　unsigned　int は揮発性無符号整数型である。volatile のみの記述は int の省略と見なされ、volatile　int と同じになる。

①文字型

主に１個の文字定数の扱いを想定して設定された型である。文字型の変数ないし char に sizeof 演算子を作用させたときの評価値は必ず１となる。ANSI　x3.4-1986 や JIS　X0201 で規定されている情報交換用米国標準符号(American Standard Code for Information Interchange：ASCII、アスキー）に対応した形の文字定数の扱いを想定しているため、char だけでは有符号か無符号かは不定である。

大多数のパーソナルコンピュータ対応のＣ言語コンパイラの char 型は８ビット長であるが、このような処理系で char を８ビット長の数値型として利用する場合、有符号なら signed　char、無符号なら unsigned　char　と記述する。

前出のアスキーコード表では、アルファベットの大文字２６文字、小文字２６文字と数字１０文字分を紹介した、これは８ビット長で対応可能な２５６通りの内６２通りの利用方法を規定していることになる。残り１９６通の幾つかに、日本のパーソナルコンピュータではカタカナをふり当てていることもある。アスキーは最低限のサイズとして７ビット長を想定しているがそれでも６６通りの空きが存在する、この空きの内０～０ｘ１Ｆの範囲と０ｘ７Ｆには制御文字が割り振られている。

制御文字に対して前出のアルファベットや数字は印字文字と言う。正確にはアルファベットや数字を図形文字と言い、図形文字に空白文字（スペース）を加えたものが印字文字である。制御文字はコントロール文字ないし機能文字と呼ばれることもある。制御文字はプリンタや標準出力への出力を制御する。改行動作を示す Line　Feed（0x0A）やタブ（水平タブ）を示す Horizontal　Tabulation（0x09）が制御文字の例である。改行は「プリンタのヘッドを次の行へ」の指示であり、タブは次のタブ位置へヘッドを移動させる指示である。以下に制御文字の一覧を示す。

16進	意味	略号表現	２文字表現
００	空白(NULL)	NULL	ＮＵ
０１	ヘッディング開始(Start of Heading)	TC1、ＳＯＨ	ＳＨ
０２	テキスト開始(Start of Text)	TC2、ＳＴＸ	ＳＸ
０３	テキスト終結(End of Text)	TC3、ＥＴＸ	ＥＸ
０４	伝送終了(End of Transmission)	TC4、ＥＯＴ	ＥＴ
０５	問い合わせ(Enquiry)	TC5、ＥＮＱ	ＥＱ
０６	肯定応答(Acknowledge)	TC6、ＡＣＫ	ＡＫ
０７,¥a	ベル(Bell)	BEL	ＢＬ
０８,¥b	後退(BackSpace)	FEO、ＢＳ	ＢＳ
０９,¥t	水平タブ(Horzontal Tabulation)	FE1、ＨＴ	ＨＴ
０Ａ,¥n	改行(Line Feed)	FE2、ＬＦ	ＬＦ
０Ｂ,¥v	垂直タブ(Vertical Tabulation)	FE3、ＶＴ	ＶＴ
０Ｃ,¥f	書式送り(Form Feed)	FE4、ＦＦ	ＦＦ
０Ｄ,¥r	復帰(Carrige Return)	FE5、ＣＲ	ＣＲ
０Ｅ	シフトアウト(Shift Out)	ＳＯ	ＳＯ
０Ｆ	シフトイン(Shift In)	ＳＩ	ＳＩ
１０	転送制御拡張(Data Link Escape)	TC7、ＤＬＥ	ＤＬ
１１	装置制御１(Device Control1)	ＤＣ１	Ｄ１
１２	装置制御２(Device Control2)	ＤＣ２	Ｄ２
１３	装置制御３(Device Control3)	ＤＣ３	Ｄ３
１４	装置制御４(Device Control4)	ＤＣ４	Ｄ４
１５	否定応答(Negative Acknowledge)	TC8、ＮＡＫ	ＮＫ
１６	同期信号(Synchronous Idle)	TC9、ＳＹＮ	ＳＹ
１７	伝送ブロック終結(End of Transmisson Block)	TC10、ＥＴＢ	ＥＢ
１８	取消し(Cancel)	ＣＡＮ	ＣＮ
１９	媒体終端(End of Madium)	ＥＭ	ＥＭ
１Ａ	置換キャラクタ(Substitute Character)	ＳＵＢ	ＳＢ
１Ｂ	拡張(Escape)	ＥＳＣ	ＥＣ
１Ｃ	ファイル分離キャラクタ(File Separator)	IS4、ＦＳ	ＦＳ
１Ｄ	グループ分離キャラクタ(Group Separator)	IS3、ＧＳ	ＧＳ
１Ｅ	レコード分離キャラクタ(Record Separator)	IS2、ＲＳ	ＲＳ
１Ｆ	ユニット分離キャラクタ(Unit Separator)	IS1、ＵＳ	ＵＳ
７Ｆ	抹消(Delete)	ＤＥＬ	ＤＴ

先の一覧の英字拡張表記（￥ａ（ベル）～￥ｒ（復帰））に一重引用符を示す￥’、二重引用符を示す￥”、疑問符を示す￥？、円記号を示す￥￥を加えたものが単純拡張表記である。単純拡張表記に￥０～の８進拡張表記と￥ｘ～の１６進拡張表記を加えたものが拡張表記エスケープシーケンスである。これらのエスケープシーケンスはデータタイプとしては文字型に対応する。￥の直後にａ、ｂ、ｆ、ｎ、ｒ、ｔ、ｖ、ｘ、’、”、￥、０，１，２，３，４，５，６，７，以外がくる拡張表記を用いたプログラムは規格厳密一致プログラムではない。英小文字は将来の規格化のため予約されており、現状では未定義で、数字や８，９、英大文字の場合は処理系依存である。

②整数型

文字型と列挙体型を除いた汎整数型が整数型で、先に述べたｓｈｏｒｔ　ｉｎｔ、ｉｎｔ、ｌｏｎｇ　ｉｎｔ、ｕｎｓｉｇｎｅｄ　ｓｈｏｒｔ　ｉｎｔ、ｕｎｓｉｇｎｅｄ　ｉｎｔ、ｕｎｓｉｇｎｅｄ　ｌｏｎｇ　ｉｎｔが整数型を示す。整数型で取り扱うことが可能な値の範囲、数量的限界は整数限界マクロとして整数限界ヘッダ＜ｌｉｍｉｔｓ．ｈ＞中に等しいか一つ大きい値として定義されている。

ｓｈｏｒｔ　ｉｎｔには、上限値マクロとしてＳＨＲＴ＿ＭＡＸマクロが、下限値マクロとしてＳＨＲＴ＿ＭＩＮマクロが対応する。ｉｎｔには、上限値マクロとしてＩＮＴ＿ＭＡＸマクロが、下限値マクロとしてＩＮＴ＿ＭＩＮマクロが対応する。ｌｏｎｇ　ｉｎｔには、上限値マクロとしてＬＯＮＧ＿ＭＡＸマクロが、下限値マクロとしてＬＯＮＧ＿ＭＩＮマクロが対応する。ｕｎｓｉｇｎｅｄ　ｓｈｏｒｔ　ｉｎｔには、上限値マクロとしてＵＳＨＲＴ＿ＭＡＸマクロが、下限値マクロとしてＵＳＨＲＴ＿ＭＩＮマクロが対応する。ｕｎｓｉｇｎｅｄ　ｉｎｔには、上限値マクロとしてＵＩＮＴ＿ＭＡＸマクロが、下限値マクロとしてＵＩＮＴ＿ＭＩＮマクロが対応する。ｕｎｓｉｇｎｅｄ　ｌｏｎｇ　ｉｎｔには、上限値マクロとしてＵＬＯＮＧ＿ＭＡＸマクロが、下限値マクロとしてＵＬＯＮＧ＿ＭＩＮマクロが対応する。

ｃｈａｒは文字型を示すが、実体は整数型と同じく内部構造を持たないので、その限界も整数限界ヘッダ＜ｌｉｍｉｔｓ．ｈ＞中にｃｈａｒ上限値を示すＣＨＡＲ＿ＭＡＸマクロ、ｃｈａｒ下限値を示すマクロＣＨＡＲ＿ＭＩＮマクロとして定義されている。また整数限界マクロとしてMB_LEN_MAXマクロが定義されている。これは多バイト文字を扱うのに必要な最大バイト数を示すマクロである。

③浮動小数点型

浮動小数点型は実数型ともいい、浮動小数点表現方式の内部表現を持つ。単精度浮動小数点型のｆｌｏａｔ、倍精度浮動小数点型のｄｏｕｂｌｅ、拡張倍精度浮動小数点型のｌｏｎｇ　ｄｏｕｂｌｅの三つに分類される。浮動小数点型は、符号ビット、指数部ビット、小数部ビットの３つつの部分からなる内部構造を持つ。符号ビットは通常１ビットで正負の符号を示し、指数部は指数範囲を示し、小数部は１０進精度桁を示す。これらの特性を示す数量的限界は浮動小数点ヘッダ＜ｆｌｏａｔ．ｈ＞中に浮動小数点特性マクロとして定義されている。

浮動小数点特性マクロは、浮動小数点基数マクロＦＬＴ＿ＲＡＤＩＸマクロと浮動小数点丸

めモードマクロＦＬＴ＿ＲＯＵＮＤＳマクロとｆｌｏａｔ型特性マクロＦＬＴ＿マクロとｄｏｕｂｌｅ型特性マクロＤＢＬ＿とｌｏｎｇ　ｄｏｕｂｌｅ型マクロＬＤＢＬ＿マクロからなる。ＦＬＴ＿ＲＡＤＩＸマクロは指数部表現における基数を示す。ＦＬＴ＿ＲＯＵＮＤＳマクロ浮動小数点加算の丸めモードを示すマクロでその値によって以下のような意味を示す。－１は不確定、０はゼロ方向、１は最も近傍へ、２は正の無限大方向、３は負の無限大方向である。

ＦＬＴ＿マクロ群、ＤＢＬ＿マクロ群、ＬＤＢＬ＿マクロ群は下記のような種類のパラメータを持つ。基数精度はＦＬＴ＿ＭＡＮＴ＿ＤＩＧやＤＢＬ＿ＭＡＮＴ＿ＤＩＧやＬＤＢ＿ＭＡＮＴ＿ＤＩＧで表され浮動小数点有効数字部のＦＬＴ＿ＲＡＤＩＸ桁数を示す。精度はその１０進桁数をＦＬＴ＿ＤＩＧやＤＢＬ＿ＤＩＧやＬＤＢＬ＿ＤＩＧで表す。判別限界は１．０に加えても１．０との差が検出できない値を示し、ＦＬＴ＿ＥＰＳＩＬＯＮやＤＢＬ＿ＥＰＳＩＬＯＮやＬＤＢＬ＿ＥＰＳＩＬＯＮで表す。累乗したとき浮動小数点表現に収まる最小の負の整数や最大の整数はそれぞれ、ＦＬＴ＿ＭＩＮ＿ＥＸＰとＦＬＴ＿ＭＡＸ＿ＥＸＰや、ＤＢＬ＿ＭＩＮ＿ＥＸＰとＤＢＬ＿ＭＡＸ＿ＥＸＰや、ＬＤＢＬ＿ＭＩＮ＿ＥＸＰとＬＤＢＬ＿ＭＡＸ＿ＥＸＰで表される。１０を累乗したとき浮動小数点の範囲内に収まる値の上限を１０指数上限値、下限を１０指数下限値といい、ＦＬＴ＿ＭＡＸ＿１０＿ＥＸＰとＦＬＴ＿ＭＩＮ＿１０＿ＥＸＰやＤＢＬ＿ＭＡＸ＿１０＿ＥＸＰとＤＢＬ＿ＭＩＮ＿１０＿ＥＸＰや、ＬＤＢＬ＿ＭＡＸ＿１０＿ＥＸＰとＬＤＢＬ＿ＭＩＮ＿１０＿ＥＸＰで表現する。表現可能な最小の浮動小数は下限絶対値、最大の浮動小数点数は上限絶対値としてＦＬＴ＿ＭＩＮとＦＬＴ＿ＭＡＸや、ＤＢＬ＿ＭＩＮとＤＢＬ＿ＭＡＸや、ＬＤＢＬ＿ＭＩＮとＬＤＢＬ＿ＭＡＸに定義されている。

④型変換

Ｃ言語では全ての識別子は型を持ち、型の指定は操作可能な値の範囲及び精度の指定に他ならない。演算の実行に際して値の型がそれぞれ異なる場合、型の変換が必要になる。この型の変換が型変換であり、型変換にはキャスト演算子を用いた明示的型変換と暗黙的型変換がある。

キャスト演算子は明示的型変換演算子ともいい、小丸括弧（）で表され、用いる時には型名を小丸括弧で囲む。式の評価値の型が括弧内の型に変換されることを示し、（型名）式の記述が基本形でこのようにキャスト演算子を持つ式をキャスト式という。（型名）キャスト式の記述も許される、この場合（型名１）（型名２）・・・（型名ｋ）式の記述になるがキャスティングは右から実行され、型名ｋへの変換、型名ｋ－１への変換・・・型名２への変換、型名１への変換の順に実行される。

式の値の型と変換後の型の関係で、明示的型変換だけを許すものと暗黙的型変換と明示的型変換の両方を許すものがある。今まで登場した型は基本的に暗黙的型変換も明示的型変換も許すが、オブジェクト型や不完全型や関数型においては明示的型変換のみが可能な事が多い。暗黙的型変換が可能であろうと型変換後の精度等は変換前と変換後の型の関係のみで決まる。

変換前の型をｘ型、変換後の型をｙ型として、汎整数型同士の場合、ｓｉｇｎｅｄ～同士の場合はｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲以上であれば値不変、ｙ型で操

作可能な値の範囲がｘ型で操作可能な値の範囲未満であれば、ｙ型でも表現可能な範囲内の値であれば値不変でそうでなければ処理系依存となる。ｕｎｓｉｇｎｅｄ～同士の場合、ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲以上であれば値不変、ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲未満であれば、ｙ型でも表現可能な範囲内の値であれば値不変でそうでなければｘ型の値を（ｙ型で表現できる上限の値＋１）で割ったものの商（整数値）となる。

ｘ型がｓｉｇｎｅｄ～でｙ型がｕｎｓｉｇｎｅｄ～の場合、ｘ型の値が０以上の時、ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲以上であれば値不変、ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲未満であれば、ｙ型でも表現可能な範囲内の値であれば値不変でそうでなければｘ型の値を（ｙ型で表現できる上限の値＋１）で割ったものの商（整数値）となる。ｘ型の値が－１以下の時、ｘ型の値とｙ型の値が等しい場合ｘ型の値＋（ｙ型で表現できる上限の値＋１）が変換値となる。ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲より大きければ、ｘ型の値をｓｉｇｎｅｄ　ｙ型でキャストした値＋（ｙ型で表現できる上限の値＋１）が変換値となり、ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲未満であれば、（ｙ型で表現できる上限の値＋１）－（－ｘ型の値を（ｙ型で表現できる上限の値＋１）で割ったものの商（整数値））が変換値となる。

ｘ型がｕｎｓｉｇｎｅｄ～でｙ型がｓｉｇｎｅｄ～の場合、ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲より大きければ値不変、ｙ型で操作可能な値の範囲がｘ型で操作可能な値の範囲以下であれば、ｘ型の値がｙ型でも表現可能な範囲内の値であれば値不変でそうでなければ処理系依存となる。

汎整数型と浮動小数点型の間の型変換では、浮動小数点型から汎整数型への変換の場合、小数点以下は切り捨てられ、整数部の値が整数型で表現可能な範囲ならば整数部が変換結果となり、そうでない場合の動作はＡＮＳＩ案としては未定義である。汎整数型がｕｎｓｉｇｎｅｄ～であっても整数型同士の場合行われたような剰余操作は行われない。汎整数型から浮動小数点型への変換の場合、浮動小数点型の側の精度が十分であれば変換値は不変であるが、そうでない場合整数型の値に対して、精度分上側の値になるか、下側の値になるかは処理系依存である。

浮動小数点型間の型変換では、変換前の型をｘ型変換後の型をｙ型として、ｙ型の表現可能範囲の方が広い場合は値は不変である。そうでない場合精度が影響を与え、精度が十分であれば値不変で、精度が不足している場合、一番近い値の上側の値または下側の値になるかは処理系依存である。

（６）制御文

制御文がない場合、プログラムの実行順序は基本実行順序に従う。Ｃ言語の基本実行順序は、日本語や英語の読みと同じで上から下、左から右である。Ｃ言語には＃で行を書き始める前処理制御行以外は行の概念はなく＃で始まる行以外はどのように書いても基本実行順序に従って実行される。また関数呼出しがあれば呼び出した関数を実行し、呼び出した関数の終了後、呼

び出した位置に戻る。

プログラムの実行順序を制御する構造が制御構造でＣ言語では、制御文による選択制御構造や反復制御構造のほか、ｓｅｔｊｍｐ・ｌｏｎｇｊｍｐ関数による非局所分岐や、ｓｉｇｎａｌ・ｒａｉｓｅ関数による割り込み処理などがある。

制御文は実行順序を変える文で、選択文・反復文・跳躍文の三つに分類できる。制御文は基本的な英単語を用いた予約語で表される。接続詞のｉｆ（もし）やｗｈｉｌｅ（間）、副詞のｅｌｓｅ（でなければ）や前置詞のｆｏｒ（間）、動詞のｄｏ（する）やｂｒｅａｋ（中断する）やｃｏｎｔｉｎｕｅ（続ける）、名詞のｓｗｉｔｃｈ（スイッチ）が制御文を構成する予約語の例である。Ｃ言語の制御文は３つに分類されるが基本的にはｉｆ文とｇｏｔｏ文の二つがあれば全ての制御文が実現可能である。実現可能であるがｉｆ文とｇｏｔｏ文の二つだけで制御構造を記述した読みにくいプログラムより適切な制御文を用いた読みやすいプログラムの方が良いプログラムであるのは、前にも触れた通りである。

①選択文

Ｃ言語の選択文にはｉｆ文（判断文）とｓｗｉｔｃｈ文（多方向分岐文）の二種類があり、どちらも制御式の値によって幾つかの文の集合から実行する文を選択するための制御文である。制御式に定数式を用いることは余りしない。Ｃ言語では数量型の式は全て制御式に利用可能である。選択文を形づくる制御構造が選択制御構造でその入れ子のレベルは最低１５が保証されているはずである。

ｉｆ文は制御式の評価を行って、その結果がもし真（非０）ならば真文を実行し、偽（０）ならば偽文を実行する、判断制御を行う文である。記述としてはｉｆ（制御式）{真文}ｅｌｓｅ{偽文}であるが、ｅｌｓｅ{偽文}はなくても構わない。ｅｌｓｅ{偽文}のないｉｆ～形式と、ｅｌｓｅ{偽文}の存在するｉｆ～ｅｌｓｅ～形式が存在する事になる。ちなみに真文と偽文が同時に（連続して）実行されることはない。ｅｌｓｅは同一ブロックにある直前のｉｆと対応する。以下にｉｆ文の簡単な例を挙げる。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int x;
    x=100;
    if (x>=55) puts ("x>=55");／*if～形式*／
    if (x>=55) puts ("x>=55");／*if～      */
    else     puts ("x<55");  ／*else～形式*／
    return EXIT_SUCCESS;
}
```

注）ｐｕｔｓ関数はｓｔｄｉｏ．ｈ中に定義されている関数で、標準出力ストリーム（ｓｔｄｏｕｔ）に文字列の末尾に改行文字を追加して出力する。

注）ＥＸＩＴ＿ＳＵＣＣＥＳＳマクロはｓｔｄｌｉｂ．ｈ中に定義されているプログラムの成功終了状態を示すマクロで、上記のプログラムの場合ｍａｉｎ（）関数を呼び出したシステム（多くの場合ＯＳ）に成功終了状態を伝える。上記のようなプログラムの場合ｒｅｔｕｒｎ～の記述はなくても動作結果は同じである。

ｓｗｉｔｃｈ文はｃａｓｅラベルやｄｅｆａｕｌｔラベルとｂｒｅａｋ文の４つの予約語で構文を構成する。ｓｗｉｔｃｈ（分岐式）～ｃａｓｅ　事例式：～ｄｅｆａｕｌｔ：～の形で記述し分岐式の評価値と一致する評価値を持つ事例式を持つｃａｓｅラベルに続く文群に制御を移す。もし一致する評価値を持つ事例式がない場合、ｄｅｆａｕｌｔラベルが存在すればｄｅｆａｕｌｔラベルに続く文群（既定文群）に制御を移す。ｄｅｆａｕｌｔラベルは必ずしも存在しなくても良い。一致する評価値を持つ事例式がなくｄｅｆａｕｌｔラベルも存在しない場合ｓｗｉｔｃｈ文の｛｝で示されるブロック全体から抜け出す。各文群の最後にはｂｒｅａｋ文を記述することが多い、ｂｒｅａｋ文の記述が無い場合実行された文群に続くｃａｓｅラベルの事例式が評価されてしまう。

ｃａｓｅラベルとｄｅｆａｕｌｔラベルはｓｗｉｔｃｈ文の本体中でのみ用いられるが、任意の識別子の末尾に：コロンをつければラベルとなる。ｓｗｉｔｃｈ文では事例式との組み合わせで次に実行する文の置かれている場所を示したが、ｇｏｔｏ文ではｇｏｔｏ　ラベル；の形で次に実行する文の場所を示すのに用いられる。識別子にコロンがつけばラベルとなるのでｓｗｉｔｃｈ文の場合ｄｅｆａｕｌｔラベルの綴りを間違えた場合等、ユーザ定義のラベルとして認識され文法エラーは発生しないので注意が必要である。なお１つのｓｗｉｔｃｈ文中には少なくとも２５７個のｃａｓｅラベルが使用可能である。以下にｓｗｉｔｃｈ文の簡単な例を挙げる。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int x;
    x=1;
    switch (i) {
        case 0:
            puts ("x=0");
            break;
        case 1:
            puts ("x=1");
```

```
            break;
        case  2:
            puts (" x＝2");
            break;
        default:
            puts (" x<>0, 1, 2");
            /*caseラベルが続いていないので*/
            /*break文は無くてもかまわない*/
    }
    return  EXIT_SUCCESS;
}
```

②反復文

反復文は、ＪＩＳでは繰り返し文とも表現され、前判定反復文のｗｈｉｌｅ文、後判定反復文のｄｏ文及び所定回反復文のｆｏｒ文の３つがある。いずれも制御式の値が０になるまで、各予約語（ｆｏｒ、ｗｈｉｌｅ、ｄｏ）に続くループ本体を反復実行する制御文である。このように反復文で形づくる制御構造が反復制御構造である。その入れ子のレベルは最低１５が保証されている。制御式に含まれ反復制御構造を制御するために用いる変数を制御変数と呼び、特にｆｏｒ文の基本形においてはその存在と働きを意識する必要がある。

ｗｈｉｌｅ文は、まず継続条件式の評価を行いその結果が真（非０）ならばループ本体中の文の実行を行う。文の実行に先立ち継続条件式を評価するので前判定の反復を行っている。ｗｈｉｌｅループの最小繰り返し回数は０回である。ｗｈｉｌｅ文やｆｏｒ文は、ファイルの終わりまで読み込むとか、あるいはキーから特定の入力を待つといった、ある条件が満たされるまで特定の処理を繰り返す場合に用いる。ループの途中からの脱出にはｂｒｅａｋ文、一部を省略し次の繰り返しにかかるにはｃｏｎｔｉｎｕｅ文を組み合わせる。継続条件式に定数０（偽）を用いると、ｗｈｉｌｅ（０）｛ｐｕｔｓ（" Ｉ　ｈａｖｅ　ｎｏ　ｒｏｌｌ！"）；｝のような記述になりループ本体中のｐｕｔｓ文は一度も実行されない。継続条件式に非０（真）の定数を用いると、ｗｈｉｌｅ（１）｛ｐｕｔｓ（" ｅｎｄｌｅｓｓ　ｌｏｏｐ！"）；｝のような記述になりループ本体中のｐｕｔｓ文が何回も実行される無限ループとなる。無限ループは次の節で説明するｂｒｅａｋ文を組み合わせて使用するのが基本である。以下にｗｈｉｌｅ文の簡単な例を示す。

```
#include  <stdio.h>
#include  <stdlib.h>
int  main ()
{
```

```
    int  x＝0；    ／＊宣言と同時に値０をセットしている＊／
    while（x＜3）{   ／＊式xの評価値が３未満の間繰り返す＊／
        printf（" x＝%d￥n", x）；
        ／＊式xの評価値を１０進表現で表示する＊／
        ＋＋x；    ／＊変数xの内容を１つ増加させる＊／
    }
    return  EXIT_SUCCESS；
}
```

実行結果

```
x＝0
x＝1
x＝2
```

注）変数xの内容が０から１つづつ増加してゆき、３になった時点でx＜３の継続条件式の評価値が偽（０）となりループ本体の実行を終了する。

do文はdo～whileの形で後判定反復を実現する。while文がまず継続条件式の評価を行うのに対して、do文はまずループ中の文を実行してから継続条件式の評価を行い、その真偽によって次のループを繰り返すかを決定する。do～whileの最小繰り返し回数は１回である。継続条件式に真（非０）の定数を用いた無限ループの基本動作はwhile文と同じであるが、継続条件式に偽（０）の定数を用いた場合の動作は以下に示すようにwhile文の場合と異なる。

```
#include  <stdio.h>
#include  <stdlib.h>
int  main ()
{
    do {
        puts（" I  have  no  roll！"）；
    } while（0）；
    return  EXIT_SUCCESS；
}
```

実行結果

```
I  have  no  roll！
```

注）ループブロックに続くwhile（継続条件式）の後に；（セミコロン）が必要なことに注意。

ｆｏｒ文は所定回の反復をするための文で、初期設定を追加したｗｈｉｌｅ文と同等であるが、再設定式としてｗｈｉｌｅのループブロック中の内容を持たせる事もできる。ｗｈｉｌｅ文では（）中に記述するのは継続条件式だけであったが、ｆｏｒ文ではセミコロンで区切って初期設定式、継続条件式、再設定式の３つを記述することができる。この３つのうち継続条件式は省略すると非０定数（真）に置き換えられるので、省略の場合、そのｆｏｒ文は無限ループを構成する。以下にｆｏｒ文の簡単な例を示す。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int x;／＊ｆｏｒ文で初期設定するのでここで設定しなくても良い＊／
    for (x＝0;x＜3;＋＋x){
        printf ("x＝%d￥n", x);
        ／＊式xの評価値を１０進表現で表示する＊／
    }
    return EXIT_SUCCESS;
}
```

実行結果

```
x＝0
x＝1
x＝2
```

注）ｘ＝０の初期設定式は最初に１回だけ実行されるのに対し、ｘ＜３の継続条件式や再設定式はループを繰り返す毎に実行される。上記の例では変数ｘがループの繰り返し回数を決める働きをしている。このような変数が制御変数である。

③跳躍文

跳躍文は、他の場所への「無条件跳躍」によってプログラムの実行の流れを変える制御文で分岐文・飛躍文・ジャンプ文とも表現する。Ｃ言語の跳躍文にはｇｏｔｏ文、ｃｏｎｔｉｎｕｅ文、ｂｒｅａｋ文、ｒｅｔｕｒｎ文の４種類が存在する。このうちｇｏｔｏ文のみが次に実行する場所を示すラベル識別子を持ち、プログラム中の任意に場所に制御を移すことができる。一見この機能は強力で便利そうであるが、構造化プログラミングの技法に習熟するまでは、安易なｇｏｔｏ文の利用は慎むべきである。またＣ言語をはじめとする構造化言語ではｇｏｔｏ文を用いなくても通常のプログラムの記述に差し障りないようになっている。安直にｇｏｔｏ文を用いてロジックの流れがゴチャゴチャになったプログラムをスパゲッティプログラムと言うこともある。（注、任意の場所に制御を移すことができると書いたが、ｓｗｉｔｃｈ文中のｃａｓｅラベルやｄｅｆａｕｌｔラベルには制御を移すことはできないことになっている。）

ｇｏｔｏ文以外の跳躍文は、構造化されたプログラムの記述に必要である。ｃｏｎｔｉｎｕ

ｅ文やｂｒｅａｋ文はｗｈｉｌｅ文やｆｏｒ文等の反復文の中で（多くは）ｉｆ文を組みあわせて使用される。ｒｅｔｕｒｎ文は既に何回か登場しているが、ｒｅｔｕｒｎ；又はｒｅｔｕｒｎ　式；の形で用いられ、関数の終了、すなわち関数を呼び出した次のステートメントに制御を移す働きをする。ｒｅｔｕｒｎ　式；の場合、式がそのｒｅｔｕｒｎ文を含む関数の戻り値を示す。

ｂｒｅａｋ文やｃｏｎｔｉｎｕｅ文は反復構造の反復の仕方に影響を与える。ｂｒｅａｋ文は名前のとおり、反復繰り返しを壊す働きがある。すなわちｂｒｅａｋ；を直接含むｗｈｉｌｅ文やｆｏｒ文やｄｏ～ｗｈｉｌｅ文の繰り返しの終わりを示す閉じ中括弧の次のステートメントに制御を移す。数え上げ型のｆｏｒ文中でｂｒｅａｋ文が動作した場合、ｆｏｒの条件で指定した繰り返し回数よりも少ない繰り返し回数で反復が終了する。以下にｂｒｅａｋ文のごく簡単な例を示す。

```
#include <stdio.h>
#include <stdlib.h>
void main ()
{
    int x;
    for (x=0;x<3;++x){
        printf ("ここは実行される。¥n");
        break;
        printf ("ここは実行されない。¥n");
    }
}
```

実行結果

ここは実行される。

注）ｆｏｒ文の繰り返し回数としては３回のはずだが１回しか実行されていない。

```
#include <stdio.h>
#include <stdlib.h>
void main ()
{
    int x=0;
    while (1){
        x++;
        printf ("break前¥n");
        if (x>2) break;
        printf ("break後¥n");
```

```
    }
  }
```

実行結果

```
  ｂｒｅａｋ前　・・・ｘの値は１
  ｂｒｅａｋ後
  ｂｒｅａｋ前　・・・ｘの値は２
  ｂｒｅａｋ後
  ｂｒｅａｋ前　・・・ｘの値は３
```

ループの繰り返しを途中で強制終了させるｂｒｅａｋ文に対して、ｃｏｎｔｉｎｕｅ文はループの繰り返し回数に影響は与えないが、ｃｏｎｔｉｎｕｅ文以降繰り返しの終了を示す閉じ中括弧までの内容をスキップする。

```
  ＃ｉｎｃｌｕｄｅ　＜ｓｔｄｉｏ．ｈ＞
  ＃ｉｎｃｌｕｄｅ　＜ｓｔｄｌｉｂ．ｈ＞
  ｖｏｉｄ　ｍａｉｎ（）
  {
     ｉｎｔ　ｘ；
     ｆｏｒ（ｘ＝０；ｘ＜３；＋＋ｘ）{
        ｐｒｉｎｔｆ（”ここは実行される。￥ｎ”）；
        ｃｏｎｔｉｎｕｅ；
        ｐｒｉｎｔｆ（”ここは実行されない。￥ｎ”）；
     }
  }
```

実行結果

```
  ここは実行される。
  ここは実行される。
  ここは実行される。
```

注）ｆｏｒ文の繰り返し回数の３回分ｃｏｎｔｉｎｕｅ文の前まで実行されているがｃｏｎｔｉｎｕｅ文の後ろはスキップされている。

（７）関数

ステートメント以外で動作の固まりの記述に用いられるのが関数である。関数はステートメントや他の関数を用いて記述され、関数呼び出しによって実行される。実行に際して呼び出す側から呼び出される関数に対して値を引き渡すのに用いられるのが引数である。引数は必要がなければ省略可能である。呼び出された関数が呼び出した側に渡す計算結果等の値が戻り値である。特に戻り値を持たない関数はｖｏｉｄ型として定義される。引数を省略する場合もｖｏｉｄと記述するのが厳密なＡＮＳＩ案に準拠した記述スタイルである。

①関数定義

関数はその利用に際して本体が記述されている必要がある。戻り値のデータタイプ　関数名（引数の並び）｛　関数の動作を記述する複文　｝の形で記述する。戻り値のデータタイプがvoidの場合は、式なしのｒｅｔｕｒｎ文が含まれるかｒｅｔｕｒｎ文が含まれず関数の動作の記述の最後を示す閉じ中括弧がｒｅｔｕｒｎ文と同じ動作をするかのどちらかである。戻り値のデータタイプがｖｏｉｄ以外の場合は式を持つｒｅｔｕｒｎ文が含まれ式の評価結果は戻り値のデータタイプにキャスティングしてから返される。

```
#include <stdio.h>
#include <stdlib.h>
void f1(void)
{
    printf("f1() ¥n");
}
void f2(int x)
{
    printf("f2() ¥n");
    printf("x=%d¥n");
    return;
}
void main()
{
    f1();／＊voidだから引数はない＊／
    f2(10);
    f2(21);
}
```

実行結果

```
f1()
f2()
x=10
f2()
x=21
```

```
#include <stdio.h>
#include <stdlib.h>
int f1(void)
{
```

```
    return 3.14;
}
double f2(void)
{
    return 3.14;
}

void main()
{
    double x;
    x=f1();/* double ← int */
    printf("x=%5.2f¥n", x);
    x=f2();/* double ← double */
    printf("x=%5.2f¥n", x);
}
```

実行結果

```
x=3.00
x=3.14
```

②関数呼び出し

定義された関数の機能を利用するには、利用したい関数を呼び出す必要がある。関数を呼び出したい場合、上記のようにその名前を記述すればよい。戻り値を持たないｖｏｉｄ型の関数の場合、関数名（引数値の並び）；の形で記述する。戻り値を持つ関数の場合、評価対象となる式の中に記述してもその関数が呼び出され、戻り値がその関数の名前に対応する評価値となる。戻り値を持つ関数を関数名（引数値の並び）；の形で記述した場合その戻り値は利用されないことになる。

```
#include <stdio.h>
#include <stdlib.h>
void f1(void)
{
    printf("f1() ¥n");
}
double f2(void)
{
    return 3.14;
```

```
}
void main ()
{
    double x;
    f1 ();
    x=f1 ();
    x=f2 ();
    f2 ();／＊関数ｆ２ () は呼び出されているが ＊／
    printf (" x=%5. 2f¥n", x);
    printf (" x=%5. 2f¥n", f2 ());
}
```

実行結果

```
f1 ()
f1 ()
x=3. 14
x=3. 14
```

C言語では、その関数がその関数自身を呼び出す再帰的呼出（リカーシブ・コール）が許されている。再帰的呼出を用いると再帰的に定義されている数学的関数等の記述が非常に見通しの良いものになる。例えばｎ！で表現される階乗は、１！＝１、２！＝２・１＝２・１！＝２、３！＝３・２・１＝３・２！＝６、４！＝４・３・２・１＝４・３！＝２４ 、、、ｎ！＝ｎ・（ｎ－１）！、、と定義される。階乗を計算する関数ｆｎを記述してみると以下のようになる。

```
#include <stdio. h>
#include <stdlib. h>
int fn (int n)
{
    if (n==1) return 1;／＊ 1！＝1 ＊／
    else return n＊fn (n－1);
       ／＊ n！＝n＊（n－1）！ ＊／
}
void main ()
{
    printf (" 1！＝%d¥n", fn (1));
    printf (" 2！＝%d¥n", fn (2));
    printf (" 4！＝%d¥n", fn (4));
}
```

実行結果

```
１！＝１
２！＝２
４！＝２４
```

このようにリカーシブ・コールは、再帰的な記述に用いると非常に見通しが良くなるが、C言語では関数の呼出が発生する毎にスタックを消費する。この例では関数呼出の回数はデータの値によって大きく異なる。故にリカーシブ・コールを用いる場合、特に組込機器等では関数呼出の最大回数を検討し、ワーストケースでもスタックを消費しつくしてしまわない事を確認しておく必要がある。

```
#include <stdio.h>
#include <stdlib.h>
void f1(void)
{
    printf("f1() ¥n");
}
void f10(void)
{
    f1();
    printf("f10() ¥n");
}
void main()
{
    f10();
}
```

実行結果

```
f1()
f10()
```

③関数原型宣言

C言語は前方参照型の言語で変数名や関数名等、ステートメント以外の綴り・名前が登場してきた場合前方側にその名前のデータタイプの定義を探しに行く動作をする。上記の例では、main関数中でf10()に出会い、f10()はステートメントではないので前方にf10()の戻り値の定義を求めている。これはf10()中のf1()の呼出の記述でも同じである。またこのような記述が旧来のC言語で許された記述であり、ANSI案のCコンパイラもこのような記述は受け付ける。しかしANSI案ではこのような記述は推薦されてはいない。f1()とf10()の関係を考えてみるとf1()は呼ばれる側でf10()は呼ぶ側

の関係にある。関数の呼出は機能モジュールの呼出である。一般的に呼ばれる側の機能モジュールの方がより低位の機能・より細かい記述を受け持っている。表現を変えれば呼ぶ側の最上位に位置するｍａｉｎ（）関数は、書物の目次に相当する位置づけになる。通常目次は書物の先頭に位置する。目次が末尾にある書物は決して読みやすいものではない。ＵＮＩＸには、旧来のＣ言語の記述に対応して目次に相当するファイルの末尾に存在するｍａｉｎ（）関数を表示しやすくするｔａｉｌなるｔｙｐｅ（ｃａｔ）コマンドのバリエーションが存在する。かといってｍａｉｎ（）を先頭近くに位置させて読みやすくするために、ｆ１（）やｆ１０（）の定義をｍａｉｎ（）の後方においたのでは、前方参照型の動作をするＣコンパイラがうまく動作できなくなる。そこでＡＮＳＩ案で導入されたのがプロトタイプ宣言・関数原型宣言である。Ｃコンパイラが前方参照で求めているのは名前のデータタイプである。関数ならばその戻り値のデータタイプと引数のデータタイプと個数である。見出しに対して詳細な記述である関数定義をｍａｉｎ（）の後方に記述し、その関数の鋳型に相当する原型（戻り値、引数のデータタイプ、個数）を前方に記述する事によってＣコンパイラの負担を増加させずに可読性を向上させるものである。前記の例を関数原型宣言を用いて記述すると以下のようになる。

```
#include <stdio.h>
#include <stdlib.h>
void f1(void);／＊ 宣言文であるからセミコロンが必要 ＊／
void f10(void);
void main()
{
    f10();
}
void f1(void) ／＊ f1() 本体の定義 セミコロンは不必要 ＊／
{
    printf("f1() ¥n");
}
void f10(void)／＊ f10() 本体の定義 ＊／
{
    f1();
    printf("f10() ¥n");
}
```

実行結果

```
f1()
f10()
```

（８）配列

配列・アレイ（ａｒｒａｙ）は、同じタイプの変数が連続してメモリ上に位置するよう作られた変数（オブジェクト）の列である。宣言時、まずデータタイプを記述するのは単純変数の宣言と同じであるが、同じタイプの変数を何個連続して欲しいのかを記述するパラメータが必要である。欲しい個数は識別子に続けて角括弧・[] の中に整数定数で記述する。識別子ｘで代表されるｉｎｔ型の変数が５個欲しければｉｎｔ　ｘ［５］；と宣言を記述すれば良い。宣言時、角括弧の中の値はオブジェクトの個数を示すが、配列変数を利用する時、角括弧の中の記述は添え字と呼ばれ、代表識別子名と一緒になって、変数オブジェクトを特定する働きをする。その時添え字に式の記述が可能となっている。またＣ言語では代表識別子はポインタ型の変数としてその配列の先頭アドレスを持つ。すなわちｉｎｔ　ｘ［５］；と宣言した時点で、ｘ［０］、ｘ［１］、ｘ［２］、ｘ［３］、ｘ［４］の５個のｉｎｔ型の変数が利用可能になると同時に、代表名に相当するｘはｉｎｔ型のポインタを持つ変数として確保され、＆ｘ［０］・配列の先頭の要素のアドレスがセットされている。この時、ｘはポインタ型の変数であるが、ｘの持つ値が配列の列の場所を示す手がかりなので、ｘの値を参照する事は可能であるが変更は不可能である。

ｉｎｔ型の変数ｉを想定して、ｘ［ｉ］のように添え字の位置に式を置くことも可能であり、式を置くことによって変数の列である、配列の操作が可能になる。配列は変換テーブルに用いる以外はｆｏｒ文等の反復構文と組みあわせての利用が多い。ｉの値が２の時、変数ｘ［ｉ］は変数ｘ［２］を意味する。このように添え字に式を用いる事によってダイナミックな変数の指定が可能になる。その反面、ダイナミックな指定は実行時まで式の値が確定せず添え字の式の値の有効範囲のチェックが困難になる。Ｃ言語では配列の添え字の限界チェックは完全にプログラマにゆだねられており、Ｃコンパイラは、上記の宣言後ｉ＝９；、、、ｘ［ｉ］、、等の記述があっても文法上のエラーとはしない。

キーボードやファイルから配列に値をセットする場合は、ｆｏｒ文等を用いて一個一個値を設定する必要がある、しかし初期値を設定する場合一個一個代入文でセットしていたのでは大変である。しかも配列はメモリ上に連続した位置をしめる。これはデータブロックそのものである。そこでｉｎｔ　ｘ［５］＝｛１０，２１，３２，４３，５４｝；のような列挙定数による初期化が認められている。このような整数定数のリストを数え上げ・列挙という。特にｃｈａｒ型の配列はｃｈａｒ　ｓ［５］＝”ａｂｃｄ”；のように文字列を指定した記述が可能であり、この記述はｃｈａｒ　ｓ［５］＝｛’ａ’,’ｂ’,’ｃ’,’ｄ’,’￥０’｝；と同じになる。もっとも文字列定数はｃｈａｒ型のポインタであるからｃｈａｒ　＊ｓ＝”ａｂｃｄ”；の方が文法上の整合性は高い。

数学の行列のように２次元的に並んだデータを扱うには、２組の添え字を持つ２次元配列を用いる。例えば｛１００，１０１，１０２｝と｛１１０，１１１，１１２｝のような２行３列の配列を扱う場合ＩＮＴ　ｔ［２］［３］；のような宣言を行い２行３列のサイズを持った配列変数を用意する。３次元以上の多次元配列も可能で添え字の組の個数を増やしてゆけば良い。

そのとき右端の添え字の値が１つ大きい、ないし小さいセルはメモリ上で隣り合っていることになる。

（９）構造体

任意の名前によって指定されるメンバによって構成されるデータ構造の体制が構造体であり、各メンバはそれぞれ異なったデータタイプを持ち得る。プログラム中では構造体オブジェクト宣言として記述される。学籍番号（ｉｎｔ型)、身長（ｄｏｕｂｌｅ型)、体重（ｄｏｕｂｌｅ型）で表現される一組のレコードを表すオブジェクト（変数）の宣言を考えてみる。学籍番号、身長、体重は、このレコードを構成するメンバーである。この３つのメンバーで構成された変数ｍａｎの宣言は下記のようになる。

```
ｓｔｒｕｃｔ｛
  ｉｎｔ        学籍番号；
  ｄｏｕｂｌｅ  身長；
  ｄｏｕｂｌｅ  体重；
｝ｍａｎ；  ・・・①
```

注）実際のＣ言語では、メンバ名を表現する識別子名に漢字は利用できない。コーディンク時、学籍番号はｎｏや、身長はｌｅｎや、体重はｗｇｈ等通常の識別子名に置き換える必要がある。

この時、変数ｍａｎの持つ値は３つのメンバのそれぞれに保持されている。学籍番号にアクセスするときの記述はｍａｎ．学籍番号、身長にアクセスするときの記述はｍａｎ．身長となる。構造体オブジェクト名＋ピリオド＋メンバ名で通常の変数名と同じ扱いが可能になる。

ｍａｎと同じ構造（メンバ）を持つｗｏｍａｎオブジェクトが必要になったとする。上記①の行を｝ｍａｎ，ｗｏｍａｎ；としても良いが、そのような場合は構造体タグを用いて以下のような記述も可能である。

```
ｓｔｒｕｃｔ　ｂｏｄｙ｛
  ｉｎｔ        学籍番号；
  ｄｏｕｂｌｅ  身長；
  ｄｏｕｂｌｅ  体重；
｝；
ｓｔｒｕｃｔ　ｂｏｄｙ　ｍａｎ、ｗｏｍａｎ；
```

ここで用いられているｂｏｄｙは、構造体タグで構造を示すために用いられる名前でオブジェクトではない。ｓｔｒｕｃｔ　構造体タグで通常のｃｈａｒやｉｎｔと同じようにデータタイプの指定が可能だと思って良い。

質点は、質量とその位置を表す座標と速度及び加わる力によって記述される。位置はｘ座標、ｙ座標、ｚ座標の組で表され、速度はｘ方向の速度、ｙ方向の速度、ｚ方向の速度で表され、力はｘ方向の力、ｙ方向の力、ｚ方向の速度によって表される。これを構造体で記述すると以下のようになる。

```
struct p {
  double x座標;
  double y座標;
  double z座標;
};
struct v {
  double x方向;
  double y方向;
  double z方向;
};
struct f {
  double x方向;
  double y方向;
  double z方向;
};
struct 質点 {
  double 質量;
  struct p 位置;
  struct v 速度;
  struct f 力;
};
struct 質点 o;
```

このように構造体のメンバに構造体を含めることも可能で、各メンバの記述は、質点ｏの質量はｏ．質量、ｘ方向の速度はｏ．速度．ｘ方向、ｙ方向の力はｏ．力．ｙ方向、ｚ方向の座標はｏ．位置．ｚ座標のようにピリオドを連ねて用いることで可能となる。ただしメンバに構造体を含む場合その構造体はメンバの指定に用いられるより以前（前方）に規定されていることが必要である。

```
struct body {
  int    学籍番号;
  double 身長;
  double 体重;
};
struct body student[50];
struct body *p;
```

上記のように配列やポインタを定義する事も可能である。配列の場合ｓｔｕｄｅｎｔ［７］．学籍番号でメンバ変数にアクセスできる。ポインタの場合ピリオドではなく矢印を用いる。矢

印といってもハイフンと大なりの２つのキャラクタを組みあわせて－＞のように用いる。ｐ＝ｓｔｕｄｅｎｔ；を実行した場合、変数ｐはｓｔｕｄｅｎｔ［０］のアドレスを持つのだがｓｔｕｄｅｎｔ［０］．身長にポインタｐからアクセスするにはｐ－＞身長といった記述が必要になる。

宝探しのチェックポイントに相当するものを構造体で表現することを考えてみる。各チェックポイントは宝の有無・次のチェックポイントへの位置を持っている。

```
ｓｔｒｕｃｔ　チェックポイント｛
  ｉｎｔ　宝；／＊　１ならここに宝がある　＊／
  ｓｔｒｕｃｔ　チェックポイント　＊次；
｝ｐｏｉｎｔ；
```

上記のようにメンバ変数にポインタ型を用いる事も可能だし、その構造体と同じ構造を持つポインタ変数をメンバに含めることも可能である。この場合ｐｏｉｎｔ．次－＞宝が１なら次のチェックポイントに宝があることを示す。構造体の中に同じ構造体のポインタを含ませる事は可能だが、その構造体型のポインタでない変数を含ませるとメンバーの指定が入れ子の連続になり不具合を生じる。

（１０）共用体

宣言の記述やメンバへのアクセスは、構造体と同じである。共用体宣言時に用いるキーワードがｓｔｒｕｃｔからｕｎｉｏｎに代わるだけである。ｓｔｒｕｃｔでは各メンバはメモリ上の異なった位置を占め、重なりは発生しないが、ｕｎｉｏｎの各メンバはメモリ上で同じ位置を占め、重なり合っている。前出のチェックポイントの場合宝があれば次のチェックポイントの位置を示す地図等は置いていないはずであるから、イメージ上は共用体で表現した方が実際のモデルに近づく。

インテル８０８６ＣＰＵのレジスタは、１６ビットレジスタＡＸと８ビットレジスタＡＬ＋ＡＨが重なり合って存在している。これを共用体と構造体を用いて表現すると。ＡＬとＡＨはそれぞれ８ビットレジスタとして同時に使用可能であるから構造体で表現する。ｃｈａｒ型を８ビット長、ｉｎｔ型を１６ビット長と想定して記述すると以下のような記述になる。

```
ｕｎｉｏｎ　８６レジスタ｛
  ｓｔｒｕｃｔ｛
    ｃｈａｒ　ｈ；
    ｃｈａｒ　ｌ；
  ｝ｂｙｔｅ；
  ｉｎｔ　ｘ；
｝ａ；
```

ａ．ｘは１６ビット長の変数でａ．ｘと重なりあう形で８ビット長のａ．ｂｙｔｅ．ｈとａ．ｂｙｔｅ．ｌが存在する。