第1章　この講義の目的

1.1　この講義の目的:プログラミングとは

プログラム (programs) を作る目的は何でしょうか。プログラムをすること自体が目的ではありません。何か、具体的な問題があり、それをコンピュータを使って解決するが目的でしょう。例えば、定型的な作業を大量に行ったり、大量のデータを処理して、平均や分散などの統計的性質調べるような場合です。あるいは、何か調べたい現象を数理的なモデルにして、シミュレーション (simulation) を行うのがプログラム作成の目的かもしれません。

プログラムを作成する作業をプログラミング (programming) と呼びます。具体的なプログラミングは、利用するプログラミング言語 (programming language) に依存します。しかし、問題を解決する手順を考え、それを次第に具体的にしていくという大きな手順には変わりはありません。一番大切なことは、問題を数理的な手順として具体化することなのです。

そこで、本講義では、具体的な問題を通じて、C++ を使ったプログラミングの技術を習得することを目的とします。特に、演習では、具体的な問題を扱いたいと思います。ただし、難しいアルゴリズム (algorithms) を必要とする問題は扱いません。アルゴリズムについては、別の講義に任せましょう。

1.2　プログラミングを習得するには

プログラミングに当たって一番大切なことは、解くべき問題を理解することです。そして、問題の解法の大きな構造を理解し、解法の手順をプログラミング言語として具体化する必要があります。

では、そのようなプログラミングを身に付けるにはどうすれば良いでしょうか。上述の手順は、具体的な問題解決の経験を通じてのみ習得することができます。つまり、プログラミングを習得するには、教科書を読んだり講義を聞くだけではなく、演習など具体的プログラミングの経験を重ねなければなりません。「習うより慣れろ」が基本なのです。

問題ごとにその解法が異なるように、プログラミングも問題または対象に応じて異なります。これらを全て教科書や講義で扱うことは原理的に不可能です。数学の問題解法を学ぶ際に例題を見るように、プログラミングでも例題を知ることは非常に重要です。教科書の例や他人のプログラムから、その技術を学び、自らの技術の向上を計ることを心掛けてください。ただし、他人のプログラムを写すのはいけません。プログラムにも著作権があることに注意が必要です。

プログラミング言語もまた、言語の一種です。ですから、プログラミング言語の学習は、外国語の学習に似ている部分があります。プログラミング言語にも、もちろん文法があります。また、言

語固有の単語もあります。英語の初歩を勉強した際には、文法は単語を簡単な文章を使いながら学びました。プログラミング言語の学習でも、簡単なプログラムを通じて、文法と言語固有の単語 (予約語など) を習得する必要があります。

しかし、文法と単語だけでは、長い文章を理解したり書いたりすることはできません。日本語には日本語の論理があり、英語には英語の論理があります。どのような順序で話を進めるかという論理に慣れなければなりません。プログラミング言語でも同じです。プログラミング言語にとって適切な問題解決の論理を身に付ける必要があります。

日常使われている言語 (自然言語と言います) の場合、すこしくらい文法が間違っていたり、単語の発音がおかしくても、話の大きな流れ (文脈と言います) から内容を理解してもらえます。しかし、プログラミングの場合、そういう甘い考えは通用しません。文法が間違っていれば、コンパイラという翻訳機能が文句を言います。論理が間違っていれば、実行できなかったり、正しい結果が得られなかったりします。

つまり、プログラミングには、問題に応じた論理を構築するという大局的な思考から、文法や予約語を覚え、エラーが発生した際に一つ一つを訂正していく、細かい技術までが必要です。暗記学習は通用しません。

1.3　オブジェクト指向プログラミング

プログラミング言語として古くから使われている言語に FORTRAN や C があります。従来はこれらの言語に合わせて問題をモデル化し、プログラムしてきました。これらのプログラミングでは、データ構造とその操作の関連が不明確でした。また、データへのアクセス制御などが不十分で、対象の内部データと外部から操作可能なデータの区別が付けにくくなっています。そこで、実際の問題に合わせてプログラムできるプログラミングの方法が求められて来ました。

オブジェクト指向プログラミング (Object Oriented Programming) という枠組は、そのような期待に答えるために作られたものです。対象 (object) をデータの集合として記述し、その対象ごとに特有な操作を定義するものです。そのため実世界での対象の操作とプログラム上の操作の対応を付けやすいものになっています。

例えば、行列を考えましょう。行列は、通常は要素となる複素数の集まりです。また行列の四則演算は、実数の四則演算とは異なります。そこで、行列を要素の集まりとして記述し、プログラムの上では、その行列の和や積などを+や*の記号で表すことが出来れば、数学的な操作とプログラムの対応が明確になります。オブジェクト指向プログラミングは、そのような対象の操作とプログラム上の表記の対応を簡素化します。

生物の集団のシミュレーションを行う場合を考えましょう。生物の個体には、種類や年齢あるいは遺伝情報などの個別の情報があります。また、系の中で個々の個体は生まれたり死んだりします。このような系のシミュレーションを行う場合、個体に対応するデータの集合を動的に生成したり消滅させたりできればシミュレーションプログラムの作成が容易になるでしょう。生成時の初期化や消滅時のメモリ返却が自動化できれば更に良いでしょう。このようなデータ構造の生成消滅及び自動化も、オブジェクト指向言語によって容易に実行されます。

オブジェクト指向プログラミング言語としては、古くは SmallTalk があります。現在では Java や C++などが使われています。

1.4　本講義では

本講義では、C++を使ったプログラミングを講義します。C++はCを基に作られたプログラミング言語です。CはUNIXシステムそのものの記述言語であり、UNIXシステムで実行可能なことのほとんど全てを記述することが可能です。

Javaというプログラミング言語は、オペレーティングシステム(OS)に依存しない言語であり、インターネット上のアプリケーション作成に利用され、身に付けておきたい言語の一つです。C++を習得すれば、Javaの習得は非常に簡単になるでしょう。

前期の講義では、Cと共通の機能を中心に講義していきます。繰り返しや条件分岐などの制御構造が正しく扱えることと関数を正しく作成できるようになることが目標です。

第2章　コンパイルの方法

2.1　コンパイラとインタプリタ

プログラミング言語には、コンパイラ型とインタプリタ型の言語があります。

インタプリタ型言語:

インタプリタ (Interpreter) 型言語では、プログラムの実行時に、プログラムの各行が逐次 (一行ずつ) 実行されます。プログラムに誤りがあれば、その行の直前まで実行され、誤りの箇所で停止します。その為、小さなプログラムでは、誤り発見が容易です。その半面、大きなプログラムでは、文法的誤りがある箇所まで処理が進んでしまうという問題があります。大規模な処理を行うプログラムの場合、誤りの発見が遅れるとともに、そこまで行なってしまった処理の後始末が大変です。

また、インタプリタはプログラムを 1 行ずつ解釈し実行するので、実行速度が一般的に遅くなります。実行速度をあげる為の最適化、つまり計算機にとって高速に処理できるように処理の順序などを変更することをインタプリタが行うことはありません。

インタプリタ型言語としては、様々なスクリプト言語や Basic 、また LISP や Prolog などの論理型言語等があります。

コンパイラ型言語:

コンパイラ (Compiler) 型言語では、プログラムはコンパイラによって機械語に翻訳されます。実行時には、機械語に翻訳されたものが使われます。実行に先だって機械語への翻訳が行われるので、文法的誤りのあるプログラムは、翻訳時に誤りの検出が行われ、翻訳することができません。また、機械語への翻訳時に、実行速度を上げるように、最適化処理を行うことができます。

コンパイラ型言語としては、`C`、`C++`、`Fortran`、`Pascal` などの科学技術計算用言語や `Java` などがあります。

2.2　C++プログラミングの作業の基本的流れ

プログラムが一つのファイルで構成される場合の作業の流れを示します。ここでは、g++ というフリーで取得可能なコンパイラを使うことにします。

1. プログラムをエディタで編集します。`C++` のプログラムの拡張子は`.cc` を使います[1]。使うテ

[1] コンパイラによっては、拡張子が大文字`.C` であったり、`.cpp` であったりします。

キストエディタはなんでも構いません。Emacs のように、拡張子に応じてモードを制御できるものを使うと、括弧の対応や自動インデントなど、プログラム作成を支援する機能を使うことができます。

2. プログラムができたら、コンパイルします。

```
g++ file.cc
```

この処理は、利用している開発環境に依存します。

3. コンパイル時にエラーのあった場合は、そのエラーメッセージを良く読みます。エラー箇所を特定して、1 へ戻って編集します。

4. 正しくコンパイルされると、`a.out` というファイルが生成されます。

5. `a.out` を実行します。UNIX システムの場合、コマンドラインから単に `a.out` と入力することで実行できます。

6. 実行時のエラーがあれば、1 へ戻ります。

プログラムは、複数のファイルに分割して記述することも可能です。複数のファイルをそれぞれコンパイルすることを分割コンパイルと言います。分割コンパイルしたファイルを `ld`(リンクエディター) を使って結合して、実行可能形式とします。このような方法については、必要な箇所で説明します。

また、こうした作業を効率良く行うための様々な道具類を UNIX は備えています。これについては、別の箇所で説明します。

2.3　簡単な例

最も簡単な例として、文字列を印刷するプログラムを作りましょう。UNIX などの OS は、標準で使われる出力 (`STDOUT`、標準出力といいます) をもっています。通常は画面が標準出力です。その標準出力に `hello C++`と印字するプログラムを示します。

プログラム中の`//`は注釈 (コメント、comment) を表し、プログラムを見やすくするだけで、プログラムの一部ではありません。`//`から文末までが注釈であると解釈されてコンパイルされます。

演習 2.1　例 2.3.1 のプログラムを適当なエディタ (emacs など) を使って作成しなさい。コンパイルを実行し (`g++ hello.cc`)、ファイル `a.out` が生成されるのを確認した後、実行しなさい。

演習 2.2　例 2.3.1 のプログラムで、`cout` の後ろの二重引用符`"`で囲まれた文字列が印刷されているでしょう。この印刷される文字列を変更しなさい。

Program 2.3.1 hello.cc

```
//** hello.cc **************************
//
// 簡単な C++ プログラム (Simple C++ Program)

// 必要なヘッダファイルを読み込む (Include necessary header files)
#include <iostream.h>

// プログラムの主部分 (Main part)
int main(int argc, char** argv)
{
  cout << "hello C++\n"; //標準出力へ文字列を印刷する
                         //Print the string to STDOUT (Standard Output)
  return 0; //終了状態として 0 を返す
            //Return status code 0
}
```

第3章　良いプログラミングのために

3.1　良いプログラムとは

プログラムは、正しく動作するのが最低条件です。更に、正しく動作することがプログラムから理解できる必要があります。そのように考えると、良い文章を書くことに対応した、良いプログラムを書く技術が必要になるでしょう。

ここで、「良い文章」と言うのは、技術報告、学術論文、操作説明書のような理工系の文書を書く際に問題となる「良い」文章のことです。その最も大事な要素は、文章の理解が容易であり、かつ正しく理解されることです。良いプログラムの書き方も同様です。

たとえ、自分しか使わないプログラムであっても、時間が経過した後で自分で読み直して、分かるように書いてある必要があります。プログラミング言語は自然言語ではありません。ですから、自分の書いたプログラムでも、時間が経過すると何をしようとしていたかが判読できなくなる場合があります。

仕事でプログラムを作成する場合、複数の人々が様々な部品を持ち寄って作る共同作業になります。そのような共同作業の場合、協力者がプログラムを理解出来るように書いてあるかが問題となります。

コンピュータを使って仕事をするのは、手で実行できないほど量が多い処理を行う場合です。そのような処理では、正しく処理されたことを全て確認するのは非常に困難です。従って、プログラムされた処理内容が正当であることをプログラムから読み取れるようにしておかなければなりません。

よほど小さなプログラムでない限り、プログラムは、開発時に非常に多くの時間をかけてデバッグ (debug)、つまり誤りを無くす作業が行わます。その後の保守にも非常に多くの労力を必要とします。また、多くのプログラムは、既存のプログラムを基に開発されています。読みやすいプログラムであれば、デバッグ、保守、そして再利用が容易になります。

策を弄したプログラムは、作成した本人でさへ読解不能となる危険性があります。また、過度の最適化も同様に読みにくくなる要因です。少々長くなっても読みやすいプログラムを書くほうが良い場合が多くあります。特に、最近のコンピュータの処理能力の向上のスピードは非常に早く、策を弄してわずかばかりの高速化を図るのは、全く意味がありません。そんなことをするよりも、新しいコンピュータを買ったほうが良いでしょう。

では、読みやすいプログラムを書くにはどのようにすれば良いでしょうか。読みやすさを構成する要素として、

適切な注釈、適切な名前付け、一貫したスタイル、適切な構造化及び階層化

などがあげられます。読みやすいプログラムを書く技術をただちに習得することはできません。し

かし、プログラミングの習得時に意識しておくべきでしょう。

3.2　適切な注釈

読みやすいプログラムのための最も基本的な手法は、適切な注釈 (コメント、comments) をプログラムに書き込むことです。注釈は、変数の説明、操作の説明など、プログラムの部分に関するものと、プログラム全体の説明に関するものがあります。

注釈は例 2.3.1 で使ったように、//で始まり改行で終るものが C++ では標準です。また、従来の C で使われて来た/*で始まり*/で終る注釈も使うことができます。本講義では、// の形式だけを使うことにします。また、本講義で使用しているコンパイラでは、日本語の注釈文を使うことができます。

コンピュータ内では、文字も全て数値として処理されます。各文字をどの数値に対応させるかを決めるのがコードです。通常の英文字や数値は ASCII と呼ばれるコードで、8 ビット (bit) の整数と対応付けが行われています。

日本語は英数字と数値を合わせたよりもはるかに多数の文字を使って表されています。コンピュータでは 16 ビットの整数との対応付けをするのが一般的です。日本語コードは JIS、EUC、SJIS の 3 種類が良く使われます。C++ では、これらのいずれでも利用可能ですが、UNIX システム上では EUC を使うのが標準です。UNIX システムに接続されたプリンタは EUC しか使えないことにも注意してください。パーソナルコンピュータでプログラムを作成した場合には、日本語コードが SJIS であることに注意してください。

また OS によって改行を表す記号が異なります。通常二つのコード LF(ラインフィード、ASCII コード 0A) と CR(キャリッジリターン、ASCII コード 0D) が使われます。UNIX では LF が改行を表します。Windows では LF と CR の二文字で改行を表しています。

プログラムの先頭には、プログラム全体に対する注釈を付けます。直接あるいはプリントされたプログラムを見た時に、何時、何のために作成した、何のプログラムであるかが分かるようにするのが目的です。それは、以下の要素を含みます。

- プログラムの名前
- 作成者と作成日。変更があった場合には、変更日。
- 講義のレポートならば、講義名、課題名、学籍番号、提出日など。
- プログラム全体の説明
- 使い方
- 参考にしたプログラムや文献
- 制限事項
- その他

プログラムの途中では、変数の説明、操作の説明を記述します。これらは、余り長すぎるとプログラム全体を見にくくしてしまう場合があります。読みやすいものになるように適度な注釈を書くようにします。

注釈は、単にプログラムの説明をするだけではない使い方があります。プログラム開発の途中では、一部を別のもので置き換える場合や、定数の値を一時変更する場合があります。このような場合、プログラムを直接書き換えるのではなく、古い部分を注釈として残すことで、後で変更箇所を復元することが容易にできるようになります。

また、注釈はプログラムを書いた後に書く説明のためだけのものではありません。長い文章を書く際に、最初に箇条書きで要点を書いて、それに肉付けしていくことがあります。プログラミングに際しても、最初に手順を注釈として書き、その後に各手順に対応したプログラムを実際に書くという方法で開発を行うことがあります。

3.3 名前付け

C++ では、変数や関数の名前は、大小英字、数字、幾つかの記号で定義します。その長さには文法上の制限はありません。従って、その内容が分かるような英語またはローマ字表記の日本語で名前を付けるべきです。しかし、あまり長い名前はプログラムを読みにくくすることがあることに注意してください。

名前を付ける際に、変数と定数、局所的変数と大局的変数、クラス名とオブジェクト名など (これらの意味は後述する)、一貫したスタイルを使うことで、プログラムが読みやすくなります。記号_も変数名に使うことができます。しかし、記号_で始まる名前はシステムが使っている場合があります。注意しましょう。

3.4 簡潔さ

プログラムは、文章と同様に、適切な長さの単位を組み合わせて作るべきです。通常の文章でも、あまり長い文あるいは長い段落は、文章を読みにくくする大きな要因です。また、一つ一つの構成単位は、内容が単一でかつ簡潔である必要があります。通常の文章では、一つの文や段落に複数の内容が含まれていると趣旨が不明確になり読みにくくなります。

プログラムの場合にも同様で、以下の点に注意します。

- 一つの関数は、できるだけ一つの機能に限定する。たくさんの機能をもつ関数は、機能ごとに更に小さな関数に分割する。
- 一つの関数が印刷時に多くのページに渡らないようにする。長くなるようならば、適度に分割する。
- 過剰な入れ子構造 (繰り返しの中の繰り返し、条件分岐の中の条件分岐など) は避ける。
- 実行文なども、適度に分割する。一つの文の中で多くのことを一度に実行しようとしない。
- プログラムを機能ごとにモジュールとして分割し、モジュールごとに別のファイルに記述する。

このような操作は、実行速度に影響を及ぼさない場合がほとんどです。読みやすいプログラムであることを優先すべきでしょう。

3.5　プログラムはどうやって作る

プログラミングの目的は、プログラムを読みやすくすることではなく、プログラムを作って何か作業をすることです。そこで、プログラミング作業を通じて読みやすいプログラムができる方法を作っておく必要があります。

プログラム作成の前に、解くべき対象を整理しておく必要があるのはいうまでもありません。大きな問題を考える場合には、小さな問題に分割し、それらの小さな問題を解く手順・関数を組み上げて行きます。

小さな問題の単位ごとのプログラム作成にあたって、問題解法の手順を考えます。その手順を、プログラムを実際に書く前に、プログラムファイルに注釈として書き込みます。解法手順を、順次具体的なプログラムで置き換えて行きます。この際に、注釈はそのまま残します。

以下に手順の概要をまとめます。

1. まず、作成しようとするものがプログラムなのか、関数なのかを考えます。
2. 入力と出力を整理します。つまり、どのような変数 (意味、型、数) を与えて、どのような出力を得る関数かを定めます。
3. 作業の手順を整理します。これを、プログラムの中にコメントとして記述すると良いでしょう。
4. それぞれの手順をプログラムとして実装します。ただし、それぞれの手順が、更に関数で記述される場合もあります。

これらの手順は面倒に思えるでしょう。しかし、プログラミング作業に慣れるとは、上記の手順が無意識に行えるようになることです。面倒くさがらずに、実行しましょう。また、プログラミング作業に慣れれば、日常言語で手順を書くのと同様にプログラムが書けるようになり、注釈を書く部分がある程度省略できるようになります。

第4章　基本的な宣言及び式

4.1　基本的プログラム構成

いよいよプログラミングをはじめましょう。この章では、簡単な式の計算を行います。手でもできるような計算です。しかし、プログラミングの基本となる大切な部分です。

Program 4.1.1 main.cc

```
//*****  main.cc ***************
//
//簡単な四則演算
//created : 1999/4/5
//******************************
#include <iostream.h>

int main(int argc, char** argv)
{
//この関数 (main) 全体で使う変数の定義
  int a=15;
  int b=7;

//乗算
  int product; //変数の定義
  product = a*b;
  cout <<"整数の乗算: "<<a<<" * "<<b<<" = "<<product<<"\n";

//整数の除算
  int intdiv; //変数の定義
  int res;
  intdiv = a/b;//整数の商
  res = a%b;   //余り
  cout <<"整数の除算: "<<a<<" = "<<b<<" * "<<intdiv<<" + "<<res<<"\n";

  return 0;
}
```

基本的なプログラム構成は、次のような要素からなります。

- 表題コメント部
- include 部
- データ宣言部

- main 関数
  - 変数宣言部
  - 実行部
  - retern
- 他の関数

表題コメント部

プログラム 4.1.1 の最初の部分にあるコメントが、このプログラムの表題および内容などを表す部分です。前章で説明したように、プログラムの全体構成や作成時期などが分かるように記述します。

もちろん、この部分が無くてもプログラムは動きます。しかし、この部分が無いと、後でこのプログラムを使用する際や、再利用する際に、その目的などが分かりません。その結果、全てを作り直すことになる場合があります。

include 部

#include は外部から定数や関数の定義などをヘッダファイル (header file) から読み込むことを指示します。後述するように、C/C++のプログラムでは、関数の宣言だけを行う部分と内容を実際に書く部分は別々に記述することができます。ヘッダファイルでは、関数に渡す値や戻って来る値の型だけを宣言したり、変数の型や定数を宣言します。

プログラムを書く際には、システムがあらかじめ持っている定数や関数を使う場合があります。あるいは、他のファイルに定数や関数が定義されている場合もあります。そのような場合に、定数や関数の定義されているファイルをここに含める (include) ことをコンパイラに指示しています。

プログラム 4.1.1 に現れる

```
#include <iostream.h>
```

は、標準入出力に関するクラスの定義を読み込み、cout を利用して出力することを可能にしています。C/C++は、標準的に使われる入力と出力の場所 (デバイスと呼びます) が指定されています。通常は、標準出力は画面、標準入力はキーボードに割り当てられています。

ヘッダファイルを囲んでいる<> は、コンパイラが標準で知っているシステムのヘッダファイルのディレクトリにあることを示しています。通常の UNIX システムでは/usr/include の下に置かれています。コンパイラのオプションでヘッダファイルを探すパスを追加することができます。

使用する組み込み関数や組み込みクラスによって、include するヘッダファイルを選択します。どのヘッダファイルを読み込むかは、それぞれの組み込み関数や組み込みクラスのマニュアルページを参照して決めます。例えば、プログラム 4.1.2 では、平方根を求める関数 sqrt を利用するために、システムのヘッダファイル math.h を読み込んでいます。

自分のプログラムが、複数ファイルに分かれている場合には、他のファイルで定義されている関数やクラスのヘッダファイルを読み込みます。詳しくは、後述します。

データ宣言部

C や C++ では全ての変数と関数は、型を宣言しなくてはなりません。ここでは、プログラムやプログラムファイル全体に共通する定数や変数の定義を行います。変数の定義は、安易に行ってはいけません。変数の有効範囲 (スコープ、scope) については、後述します。

関数の定義

C や C++ は、関数型プログラミング言語とも呼ばれます。プログラム全体は、関数の集合として定義されます。そのなかで、main は特別な関数で、実行はつねにここから開始されます。

関数には、数学の関数と同様に、戻ってくる値 (戻り値) と引数があります。特に main の場合は、

```
int main(int argc, char** argv)
```

という形式をとります。つまり、戻り値は整数 (int) で、引数は、コマンドラインに現れるオプションの数 (int argc) とその内容を表す文字列配列 (char** argv) です。コマンドラインオプションの扱いは後で詳しく説明します。

上述のように、C 及び C++ の構成要素は関数です。関数は、値を持ち、関数を呼び出した部分にその値を戻さなければなりません。命令 return が値を戻す命令です。main の場合、値はそのプログラムを起動したシステムに戻されます。通常は、正常終了を表す 0 を戻します。UNIX システムでは、main からの戻り値は exit を使って戻すこともあります。

4.2 変数

プログラムの中で可変な値を保持しているのが変数です。全ての変数は、型の宣言をする必要があります。型とは、その変数名にどのような内容のものを保持できるかを表します。C++ では、基本になる型は、整数型、浮動小数点型および文字型で、それぞれメモリ上での大きさが異なります。

変数名は、英文字、数字、及び_で構成されます。英文字は、大文字と小文字が区別されます。数字で始まる変数名は許されません。また、_ で始まる変数名は、システムが使っているので、安易に使ってはいけません。

どのような規則で変数名を使うかは、スタイルの問題です。一貫したスタイルを使うことが読みやすいプログラムを書くうえで重要です。例えば大文字で始まるものは定数や後述するクラス名と決めてプログラムを作成します。また、変数名および関数名に長さの制限は文法上ありません。しかし、あまり長いとプログラムを読みにくくしてしまいます。

なお、命令名は、予約語であり、変数名として使うことはできません。

変数宣言は

```
型名 変数名;
```

で行います。

コンパイル時に値が確定し、実行途中で変更できないものを定数と言います。定数の宣言は、キーワード const を使って

Program 4.1.2 sqrt.cc

```
//**** sqrt.cc *******************
//
// 2 次方程式の解を求める。
// ax^2+bx+c=0
//
// 2002/12/8
// コンパイル方法
// g++ sqrt.cc -lm
//*******************************
#include <iostream.h>
#include <math.h>//平方根を求める関数 sqrt を使うために必要

int main(int argc,char** argv){
  // ax^2+bx+c=0
  double a=1.;
  double b=3.;
  double c=1.;

  double d=b*b-4.*a*c;//判別式
  if(d>=0.){//実解がある場合 (条件判断は後述する)
    double xp=(-b+sqrt(d))/2./a;
    double xm=(-b-sqrt(d))/2./a;
    cout <<xp<<" "<<xm<<"\n";
  } else {
    cout <<"No real solutions\n";
  }

  return(0);
}
```

```
const 型名 変数名;
```

で行います。

　また、古い C の場合は、各関数内で使われる全ての変数をその関数内の実行文に先だって宣言しなくてはなりませんでした。現在の C および C++ では、その必要はありません。変数は、必要な箇所で宣言すれば良いのです。

4.3　整数型と浮動小数点型

　C++ の変数の基本になるのが、整数型です。UNIX の場合、32bit で表現されます (最近の UNIX 専用機では 64bit になっている)。最上位 bit は符号に使われるので、$2^{31}-1$ から -2^{31} の値を保持することができます。

　値の代入は=を使って行います。

```
a=b;
```

は、変数 b の値を変数 a へ代入することを表します。数学の表現と違い、=は両辺が等しいことを示していないことに注意してください。左辺と右辺の型が異なる場合には、左辺の型への変換が行

われます。特に、浮動小数点型を整数型へ代入すると、小数以下の桁が無くなる (桁落ちといいます) ことに注意が必要です。プログラム 4.1.1 において、割算の結果 intdiv が小数以下を切り捨てた値になっていることを確認してください。

小数点を含む変数は、浮動小数点型と呼ばれる型に保存されます。浮動小数点型は、有効数字が格納される仮数部、符号部、及び指数部から構成されています。例えば次のものです。

$$\pm f.fff \times 10^{\pm e} \tag{4.1}$$

浮動小数点は、単精度の float と倍精度の double があります。プログラム 4.1.2 では、倍精度の double を使っています。本講義では専ら double を使うことにします。通常の 32 ビット機の場合、double では、仮数部 52 ビット、符号部 1 ビット、指数部 11 ビットで構成されています。

4.4　文字型

一文字を表すデータ型が文字型、char です。文字型定数は、シングルクォーテーション' で囲んで表します。ダブルクォーテーション"は、文字列定数を表し、厳密に区別されるので注意が必要です。

文字型には、キーボードから直接入力可能な文字の他、エスケープ記号\で始まる特殊文字があります (\n など)。実は、文字型は短い (8bit) の整数型と同じです。文字型変数の整数としての値は、文字コードが保持されています。

文字と文字列の扱いは第 5.2 節で扱います。

4.5　簡単な式

C++ は、もちろん、四則演算を行うことができます。計算を行う場合には、値の型に注意が必要です。また、数値計算と数式上の計算との違いに注意する必要があります。変数の型や精度についてです。

C++ では、基本となる変数の型は、整数型と小数を保持できる浮動小数点型です。いずれも、通常の四則演算 (+、-、*及び/) を使うことができます。整数同士の演算の場合には、結果は整数で、浮動小数同士の演算の場合には、結果は浮動小数となります (プログラム 4.1.1 での割算に注意)。整数同士の場合には、

```
a%b
```

で変数 a の値を変数 b の値で割った余りを計算することができます。

4.6　cout 出力クラス

C++ では、結果の出力の際、標準出力、つまりディスプレイへの出力に cout というクラスを使います。cout と呼ばれる標準出力オブジェクトに、プログラムからリダイレクト (<<) で送り出す形式をとります。

```
cout <<"hello C++"<<"\n";
```

文字`\n`は改行を表します。

第5章　配列、修飾子および数値の使用

5.1　配列

私達がコンピュータを使う理由の中には、単純で大量の作業を行わせるものがあります。つまり、コンピュータを使った処理の中には、同じ処理を繰り返し行うものが多く含まれています。例えば、表に記入された数値を全て加算したり、平均値を求めたりするものです。そのように、同じ処理を繰り返し行う際には、プログラムの中では配列と呼ばれるデータの集まり (データ構造) を用います。

Program 5.1.1 noarray.cc

```
//*** noarray.cc **********
//
// 配列を使わない計算の例 (第 4 回講義資料)
// 作成日:1999/4/26
//*************************
#include <iostream.h>

int main(int argc,char** argv)
{
// データの定義
  int a1=1;
  int a2=12;
  int a3=24;
  int a4=-4;

// 和の計算
  int answer;
  answer = a1 + a2 + a3 + a4;

//結果の出力
  cout <<"和="<<answer<<"\n";
}
```

プログラム 5.1.1 は、同じ型に属する変数を番号付けして計算に用いるものです。この例では、変数の数が少ないため、変数名に番号を直接付けることで対応できました。しかし、この番号付けはプログラムを書いた本人には意味がありますが、コンピュータには単に別の変数であること以上の意味はありません。更に、変数の数が非常に多かったり、特定の番号のものに選択的に操作を行ったり、あるいは変数の総数が事前に分からない場合には、このような対応そのものができません。

多くのプログラミング言語には、同じ型の要素を連続したメモリ領域に確保する「配列」(array) という型を持っています。配列は、連続したメモリ領域に一つの名前を付け、順番を表す数字で

個々の値を区別することができます。

　プログラム 5.1.2 では、配列を使って、初期データの設定と和の計算を行っています。このように、配列を使うと、同じ操作の繰り返しなどを簡単に記述することができます。同じ操作を繰り返す for などの制御に関しては、第 6.2 節で扱います。

Program 5.1.2 array.cc

```
//*** array.cc **********
//
// 配列を使う計算の例（第 4 回講義資料）
// 作成日:1999/4/26
//*************************
#include <iostream.h>

int main(int argc,char** argv)
{
  const int NumData=4;  //データ数（定数）
  int a[NumData];       //データを格納する配列

  //データ入力
  cout <<"Input 4 data (integer) :";
  cin >>a[0]>>a[1]>>a[2]>>a[3];

  //入力されたデータの表示
  cout <<"List of Data\n";
  for(register int i=0;i<NumData;i++)
    cout <<"data "<<i<<" : "<<a[i]<<"\n";

  // 和の計算
  int answer=0;
  for(register int i=0;i<NumData;i++)answer += a[i];

  //結果の出力
  cout <<"和="<<answer<<"\n";
}
```

　C++では、配列は次のように定義します。

型 配列名 [サイズ];

型は、配列に保存される要素の型です。例えば、プログラム 5.1.2 では

```
int a[4];
```

と宣言しています。これは、整数を要素とする大きさ 4 の配列、つまり 4 つの整数値を保存できる配列を定義します。なお、このような形で配列を宣言をする場合には、配列の大きさは整数型定数でなければなりません。配列の大きさが変数になる場合、つまり配列の大きさがプログラム作成時に不明である場合については後述します。

　配列の要素は、

```
a[0], a[1], a[2], a[3]
```

として読み書きすることができます。要素の番号を表す部分をインデクス (index) と呼びます。C/C++ では、インデクスが 0 から始まることに注意が必要です。

5.2　文字列

C/C++ では、文字列に対応する特別な型は存在しません。文字列は、文字型の配列として定義されます。文字列を表す配列の終端には、終端を表す特殊な文字'\0' が置かれていることに注意が必要です。従って、目に見える文字列の長さより一つ長い文字列として構成されています。

Program 5.2.1 string.cc

```
//*** string.cc **********
//
// 文字列の操作 (第 4 回講義資料)
// 作成日:1999/4/26
//*************************
#include <iostream.h>
#include <string.h>

int main(int argc,char** argv)
{
  char prompt[]={'I','n','p','u','t',' ',':',' ','\0'};  // プロンプトの定義
  char line[256];   //入力用文字列
  char output[256]; //出力用文字列

  // データ入力
  cout <<prompt;
  cin.getline(line,sizeof(line));

  //出力文字列の生成
  strcpy(output,prompt);
  strcat(output,line);

  cout <<output<<"\n";
}
```

C/C++ では、配列の代入などは定義されていません。従って、文字列の代入や結合などは、専用の関数を必要とします。それらの文字列操作の関数は、<string.h> で定義されています。

これらの文字列操作の関数を調べるには、コマンドラインで

```
man string
```

とします。プログラム 5.2.1 では、strcpy を使って文字列を複製し、さらに strcat を使って文字列の連結を行っています。

演習 5.1　オンラインマニュアルを見て、strcpy 及び strcat の使い方を調べなさい。

プログラム 5.2.1 では、二つの文字列 line と output の為の配列の大きさが 256 となっています。これは、この二つの配列に格納される文字列の長さの最大値が\0 を含めて 256 であることを

表しています。実際に格納される文字列の長さが 256 であるということではないことに注意してください。

5.3　データの入力

標準出力のクラス cout に対応して、標準入力のクラス cin を使ってデータの入力を行うことができます。

```
cin >> 変数名 1 >> 変数名 2 >> ...;
```

値は空白で区切って入力できます。改行も空白として解釈されます。例 5.1.2 では、4 つの整数を空白または改行で区切って入力することで、和を求めることができます。

cin は、不適切な入力への対応などを完全に行うことはできませんが、簡単な入力を十分に行うことができます。

上述のように、cin では空白は区切り文字として解釈するのが原則です。そこで、文字列の入力の場合には注意が必要です。空白を含んで改行までを一つの文字列をして入力するには、プログラム 5.2.1 のように、cin.getline を用います。sizeof はある変数や型の大きさを表します。詳しくは後述します。

5.4　変数の初期化

変数は、宣言時に初期化、つまり初期の値を与えることができます。例えば

```
int a=0;
```

や

```
int a(0);
```

によってです。何も初期化を行わない場合にどのような値を初期値とするかは、C++ の文法では定められていません。コンパイラによって異なる初期化が行われます。注意が必要です。

配列の場合は、次のように初期値を与えます。

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

配列の大きさと右辺の要素の数が異なる場合、右辺の数が多ければコンパイラからの警告が行われます。少なければ、初期値の与えられていない要素の部分は、コンパイラに依存した初期化が行われます。プログラム 5.2.1 のようにサイズを指定しなければ、右辺に合わせて配列の大きさが決定されます。

5.5　多次元配列

C/C++ では、2 次元以上の配列を定義することができます。例えば、

```
int a[3][2];
```

です。これは、大きさ 2 の整数型配列を 3 つ連続して確保することを意味しています。要素の参照は a[i][j] という形で行います。

数値計算を得意とする FORTRAN と比べると、C/C++ では、多次元配列はあまり上手に扱えません。例えば、関数の引数として渡すには面倒な取扱となります。多次元配列を 1 次元配列としたり、後述する構造体やクラスとして扱うことが多いようです。

5.6 修飾子

5.6.1 様々な整数型

C/C++ では、int などの基本的な型宣言に対して、幾つかの修飾子を付けることができます。

通常の int 型は 32 ビットのうち、最上位ビットを符号として扱っています。そこで、32 ビット全てを使って整数を表現する符号の無い整数型を宣言する unsigned が修飾子の一つです。

```
unsigned int a;
```

また、通常の整数型は 32 ビットですが、short、long を使って、記憶容量を制御することも可能です。UNIX では

```
short int a;
```

と宣言した場合、16 ビットの記憶が割り当てられます。long は 32 ビットなので特別な影響はありません[1]。更に短い記憶割り当てが必要な場合には、8 ビットである char 型を使って整数を記憶することができます。

5.6.2 定数と参照型

修飾子 const を付けることによって、定数であることを宣言することができます。定数は、コンパイル時に値が確定されるので、代入などで値を変更することができません。プログラム 5.1.2 では、配列の大きさが const を付けて宣言されています。

関数の引数などに修飾子 const を付けることで、関数内で誤って変数の値を変更するのを防ぐこともできます。関数への変数渡しの方法については後述します。

既に宣言されている変数に別名を付けることもできます。

```
int c;
int &dummy=c;
```

によって、dummy が整数型 c の別名であることが宣言されます。つまり、二つの変数 c と dummy は異なる名前を持っていますが、メモリ上の実体は一つになります。これを参照型と呼びます。ここでの例のような参照型の使い方はあまり一般的ではありません。関数への変数渡しの際に、参照型は非常に重要になります。このことは、後述します。

[1]UNIX 専用機の場合、long に 64 ビット整数が対応する場合があります。コンパイラのマニュアルを参照してください。

5.6.3　register

コンピュータのメモリは、通常の CPU の外部の部分と、CPU が持つ小さな領域 (レジスタと呼ぶ) に分かれています。当然、レジスタへのアクセスが高速です。そこで、非常に頻繁に使う変数を、レジスタへ割り付けることをコンパイラに知らせるのが、修飾子 register です。繰り返しを示す for の index 変数をレジスタに割り当てるのが標準的な使い方です。

5.7　16 進数と 8 進数

整数定数の表現として、16 進や 8 進を使うことが可能です。8 進は 0 で始まり、16 進は 0x で始まります。

5.8　簡略化演算子

プログラムの中で、整数型変数を 1 だけ増やしたり減らしたりする操作を行うことが多くあります。例えば、プログラム 5.1.2 で

```
for(register int i=0;i<NumData;i++)
```

は、繰り返し数を表している変数 i を 0 から 1 ずつ増やすことを示しています。i++は

```
i=i+1;
```

を表す簡略化演算子です。同様に i--は変数 i を 1 減らします。
簡略化演算子の記法には二種類あります。上述の場合と同様に

```
int i=1;
int j=i++;
```

と書く場合を考えます。この場合、先に j に i の値が代入された後で、i の値が 1 加算されます。従って、j=1 です。しかし、

```
int i=1;
int j=++i;
```

と書くと、先に i への加算が先に行われ、次ぎに j への代入が実行されます。つまり、j=2 となります。
この差を頭で記憶しておいて、プログラムを書くのはあまり得策ではありません。簡略化演算子を式の一部として書かないのが混乱を招かない方法です。
また、1 だけの増減ではない操作もプログラムの中で頻繁に出て来る操作です。プログラム 5.1.2 で行った和の計算のような場合です。

```
answer+=a[i];
```

は

```
answer=answer + a[i];
```

の簡略化演算子です。同様な簡略化演算子に

```
-=
*=
/=
%=
```

があります。

第6章　制御文

6.1　条件文

これまで見て来たプログラムでは、命令がプログラムの上部から下部に向かって書かれた順序に従って実行されていました。この章では、条件によって実行する部分を変更したり、条件に応じて繰り返しを行う方法について説明します。

Program 6.1.1 control.cc

```
//*** control.cc **********
//
// 制御文 (第 5 回講義資料)
// while と for
// 非負の整数の和、平均、分散
// 作成日:1999/5/8
// コンパイルには-lm オプションが必要
//     g++ control.cc
//*************************
#include <iostream.h>
// 平方根を求める関数 sqrt を使うためのヘッダファイル
#include <math.h>
int main(int argc,char** argv)
{
  const int NumMax=2048;// データの最大値
  int data[NumMax];

  int num=0;
  int data_dum;
  cout << "データ入力 (整数) : 負の入力で終了\n";
  cin >> data_dum;
  while(data_dum>=0 && num < NumMax){//正の値である限り、データとして保存する
    data[num] = data_dum;
    num ++;
    cin >> data_dum;
  }

  if(num == 0){
    cout << "データが入力されませんでした。\n";
    return 0;
  }else{
    cout << "データの平均と分散を計算します。\n";
  }
```

Program 6.1.2 control.cc の続き

```
  int sum=0,square_sum=0;
  for(register int i=0;i<num;i++){
    sum += data[i];                  //データの和
    square_sum += data[i]*data[i];   //データの平方和
  }

  double mean=sum/double(num);
  double var = sqrt(square_sum/double(num) -  mean*mean);

  char id='n';
  while(id!='q'){
    cout <<"**********************************\n";
    cout <<"結果出力\n";
    cout <<"以下の命令のいずれかを入れて下さい。\n";
    cout <<"n : データ数\n";
    cout <<"m : 平均\n";
    cout <<"v : 分散\n";
    cout <<"l : データ一覧\n";
    cout <<"q : 終了\n";

    cin >>id;
    cout <<"**********************************\n";
    switch(id){
    case 'n':
      cout << "データ総数 : "<<num<<"\n"; break;
    case 'm':
      cout << "平均 : "<<mean<<"\n"; break;
    case 'v':
      cout << "分散 : "<<var<<"\n"; break;
    case 'l':
      for(register int i=0;i<num;i++)cout << data[i]<<"\n";
      break;
    case 'q':
      break;
    default:
      cout <<"正しい番号を入れて下さい。\n\n";
    }
  }
  return 0;
}
```

条件によって実行部分を変える必要がある場合に使う、最も簡単な方法が `if` 文による条件分岐です。`if` 文は、

```
if (条件) {
    文 1;
    文 2;
    ...
}
```

という形で使います。条件を満たす時、文が実行されます。実行部分が複数の命令から構成される場合は、命令部分を括弧でグループ化します。条件が満たされない場合には、単純に条件節の部分が実行されません。

条件の部分には、表 6.1 に示す演算子を使った比較、及びそれらの比較の論理演算 (表 6.2) を書きます。

表 6.1: 比較演算子

演算子	意味
==	等しい (=との違いに注意)
!=	等しくない
<	左辺が右辺より小さい
<=	左辺が右辺以下
>	左辺が右辺より大きい
>=	左辺が右辺以上

表 6.2: 論理演算子

演算子	意味
||	論理 OR
&&	論理 AND
!	論理 NOT

条件を満たす場合と、満たさない場合にそれぞれ異なる操作を行いたい場合には、

```
if（条件）{
    文 1;
} else {
    文 2;
}
```

と記述します。

条件節の中に更に新たな条件節を記述することもできます。条件節の中に条件節があったり、繰り返しの中に繰り返しがあることを入れ子 (nesting) 構造と呼びます。

プログラム 6.1.1 は、キーボードから整数を入力し、それらの平均値と分散を計算するプログラムです。データは全て非負であるとし、負の値を入力することで入力終了とします。

プログラム 6.1.1 は、平均と分散を計算するプログラムですから、データ数が 0 の場合には処理を継続しないように処理が行われています。そのようにデータ数に応じて処理を変更するために、if 文が利用されています。

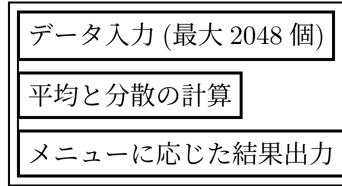


図 6.1: プログラム 6.1.1 の制御の大きな流れを表す PAD 図

```
if(num == 0){
  cout << "データが入力されませんでした。\n";
  return 0;
}else{
  cout << "データの平均と分散を計算します。\n";
}
```

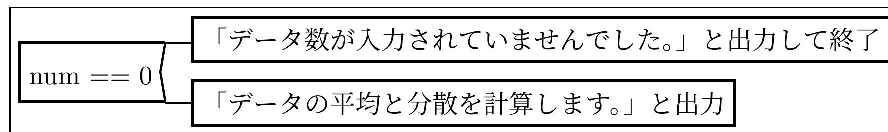


図 6.2: プログラム 6.1.1 での、データ数に応じた条件分岐

6.2　繰り返し

前節で見た例のように、同じ操作を決められた回数繰り返すとき for 文を使います。図 6.3 に、プログラム 6.1.1 で平均と分散を計算するために for を利用している部分の PAD 図を示します。

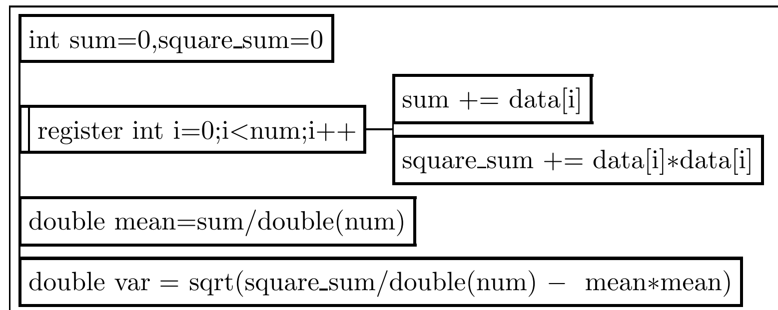


図 6.3: プログラム 6.1.1 での、平均と分散の計算部分

for 文は、以下のように記述します。for 文を使う場合、通常は、繰り返しの回数を指定する変数 (カウンタ) を用います。for 文では、カウンタの初期値、最大値、増分を指定します。

```
for(初期値; 最大値または終了条件; 増分){
     文 1;
```

```
    文 2;
    ...
}
```

例のように、カウンタは for ループの中でのみ有効な場合が多いので、

```
for(register int i=0;i<num;i++)
```

のように、ループ内のみの変数として定義すると、混乱を避けることができます[1]。なお、register というキーワードは、CPU 内に変数メモリを取ることを指定します。こうすることで、カウンタ変数へのアクセスを高速に行うことができます。ループカウンタ変数には、通常、i や j などの短い変数名を使います。

C/C++ では、カウンタの値をループ内で変更することが可能です。従って、for 文を使っていても、無限ループになることがあります。注意が必要です。例えば

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

とすると、無限ループとなります。

あらかじめ繰り返し回数を指定できない場合に使うのが while 文です。例えば、プログラム 6.1.1 のように、入力データの数が事前に分からない場合などに使います (図 6.4)。

```
while(条件){
    文 1;
    文 2;
    ....
}
```

条件が満たされている限り、ループが繰り返されます。for に比べると、終了までの回数が指定されず、無限ループになる危険度が高いので、注意が必要です。

6.3 多重条件分岐

if を使った条件分岐では、一つの条件の真偽によって最大二つの場合への分岐が行われます。二つの場合以上への分岐を記述する方法の一つは、条件分岐の中で条件分岐を行うことです。しかし、多数の条件への分岐をするには不便です。

多数の場合への分岐を可能にするのが switch 文です。プログラム 6.1.1 では、結果表示に使われています。まず、場合分けを行い、各場合に対して整数の値を割り当てます。switch 文は、各場合を表す int 型の変数と、それぞれの場合の処理を表す case 文から構成されます。各場合の動作の終了には、通常は、break 文を置き、その場合の処理が終了した後、switch のグループから抜け出し、他の case 文を飛ばすことを指示します。break 文を置かないと、次に現れた実行文を

[1] 一部のコンパイラでは、ループ変数の有効範囲がループのプログラムブロックから洩れているものがあります。

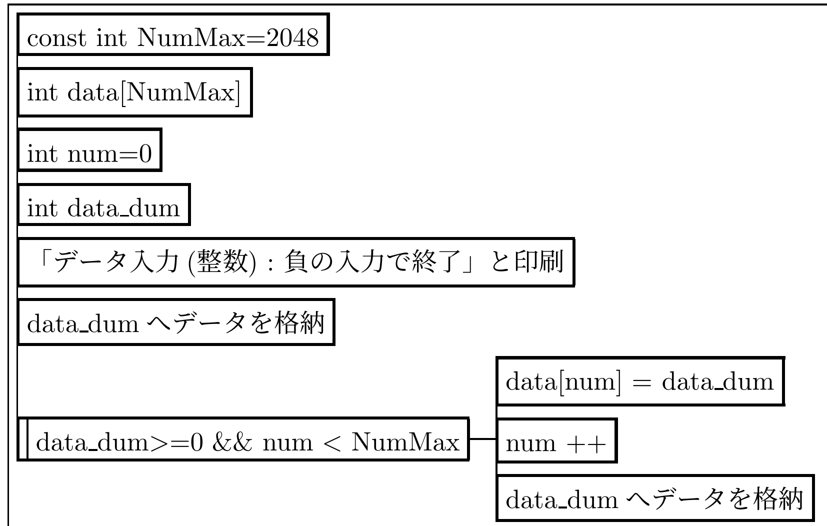


図 6.4: プログラム 6.1.1 での、データ入力部分

実行します。どの場合にも当てはまらない場合に対応する `default` という特別な場合を用意することを忘れないように注意が必要です。

```
switch(s){//変数 s の値で場合分け
  case 1:
    実行文;//s=1 の場合の処理
    breaek;
  case 2:
  case 3:
    実行文;//s=2 または s=3 の場合の処理
    break;
  ...
  default:
    実行文;//どの場合にも当てはまらない場合の処理
    break;
}
```

6.4　ループからの脱出

ループの途中から脱出する方法には大きくわけて二つの方法があります。一つは、脱出条件を表す変数を定義して、`while` や `for` の繰り返し条件に加える方法です。プログラム 6.1.1 では、この方法を採用しています。

もう一つの方法が、`break` を使う方法です。`break` は、`swhich` 文において、条件分岐から抜け

る場合、及び while ループから抜ける場合に使うことができます。

似た命令に continue があります。これは、ループの最初に戻ることを指定します。

第7章　プログラミング手順

7.1　プログラミング手順

プログラミングは、実際にプログラムを書く部分だけではなく、プログラム作成以前の作業、プログラム作成と動作確認、更に完成後の作業を含んでいます。この章では、それらの作業と、それらを支援する仕組みを扱います。

1. 仕様書

   プログラムを作成する場合、特に大規模なものの場合、プログラムを実際に書き始める前に、プログラミングの目標を定める必要があります。それが仕様書 (Specifications) と呼ばれるものです。

   仕様書では、プログラムが何をするものなのかを記述します。その目的、必要な入出力、性能、動作に当たっての制限などの記述が含まれます。

   作成するプログラムが、他人の要求に答えて作成するものである場合、仕様書作成は非常に重要な段階です。依頼者は、プログラミングに詳しくない場合が多いため、依頼者の要求がコンピュータの動作として何に対応しているかを詳しく調査する必要があります。

2. コード設計

   仕様に従って、プログラム全体を設計します。主要なアルゴリズム、クラス定義、モジュール設計、モジュール間のインターフェイス、データ構造などを記述します。

   プログラミング以前の、仕様書とコード設計を十分に行っておけば、プログラム作成のコストを小さくすることができます。

   コード設計時に、仕様書の詳細を詰める作業も行います。

3. プログラミング

   実際にプログラムを作成します。大きな枠組から作る Top-down 方式と、小さな関数などの要素から作る Bottom-up 方式があります。二つの方法は、排他的ではなく、実際には両側からの作業を行います。

   仕様やコード設計の内容や記述が不十分な場合には、仕様策定やコード設計に戻る必要があります。

4. テストとデバッグ

   出来たプログラムの動作をテストします。結果の分かる場合について、十分にテストします。仕様やコード設計を見直す必要がある場合もあります。

5. 保守

   バグの発生、要求仕様の変更などに応じて、次の開発過程にはいります。

7.2　デバッグ

デバッグ (debug) とは、プログラムに含まれる不具合 (虫、bug) を取り除く作業の総称です。

1. コンパイル時のデバッグ

   プログラムがコンパイル出来なければ、デバッグとは、文法的誤りの訂正を意味します。コンパイル時には、エラーがプログラムの行番号と共に指示されるので、それを参考に訂正します。

   一つの文法エラーが、それに続く部分に影響を与える場合が多くあります。従って、ソースファイルの上の方から直して行きます。

   大規模なプログラムを作成する場合には、プログラムを機能などで小さな要素 (モジュール) に分割し、それぞれをコンパイルします。そのようにすることで、各要素ごとにデバッグを完了して、全体を構築することができます。

2. 実行時のデバッグ

   実行時のエラーも、エラー内容が表示されます。しかし、あまり分かり易いものではありません。そこで、多くの場合、デバッグを支援するツール (デバッガ) を使います。

   デバッガが使えない場合には、エラーの恐れのある箇所の動作を表示するために、プログラムにデバッグ用出力を入れて、動作の確認をします。

3. 結果がおかしい場合のデバッグ

   エラーは出ないが動作結果がおかしい場合、最もデバッグが困難です。デバッグ用出力を入れて動作を確認するとともに、アルゴリズムなどの再確認が必要になります。

7.3　Makefile と Imakefile

プログラムのコンパイルには、様々なコンパイルオプション、ヘッダファイルやライブラリファイルへのパスの指定などが必要です。プログラムファイルが一つの場合でも、毎回正しくこれらのコンパイルオプションを指定するのは困難です。

複数ファイルから構成されるプログラムの場合は、コンパイル作業は更に困難となります。また、複数ファイルから構成されるプログラムの場合に、毎回全てのファイルを再コンパイルするのは時間と資源の無駄です。一般に、ファイルの依存関係などがあるため、各ソースファイルに対する再コンパイルの必要性を正しく認識して、小さなコストでプログラム全体を再構築するのも困難な作業です。

そこで、通常は、適当なコンパイル用のスクリプトを使って作業コストを縮小します。最もよく使われているのが UNIX が標準で持っている make という機能です。make は、指定されたファイルの依存関係に従って、必要に応じて再コンパイルを行う機能を有しています。

Program 7.3.1 Makefile

```
# Makefile for control.cc
#
#    作成：只木進一
#    作成日：1999/5/17
#
#   make でコンパイル
#-----------------------------------------------
#一般的書式
#ターゲット:依存する項目
# 動作
#-----------------------------------------------
RM=/bin/rm -f

#実行形式ターゲット一覧
all:control

#コンパイル
control:control.cc
        $(CXX) $(CXXFLAGS) -o control control.cc -lm

#後始末
clean:
        $(RM) control *.o *~
```

make の設定ファイルは Makefile と呼ばれ、次のような一般的記述をします (例 Program7.3.1)。

```
ターゲット:依存するもの
        動作
```

例えば

```
control:control.cc
        $(CXX) $(CXXFLAGS) -o control control.cc -lm
```

は control という実行形式をターゲットにしています。それを作成するには control.cc が必要であり、作成方法は c++ によるコンパイルです。“-o control” はコンパイル後に作成される実行形式を a.out ではなく、control というファイル名で保存することを指示しています。オプション”-lm” は、数学ライブラリをリンクすることを指示してい

“$” で始まる文字列は、マクロ (macro)、つまり先行して設定されている変数を表します。CXX には c++コンパイラが、CXXFLAGS にはコンパイルオプションがマクロとして指定されています。control.cc に変更があった場合には、その再コンパイルが必要であることが指示されます。

makefile を記述すると

```
make
```

だけで、コンパイルを実行することができるようになります。

make は非常に便利な機能であり、UNIX システムで広く利用されています。しかし、条件文を記述することができません。このため、OS ごとに異なる状況に対応することができないのが欠点です。

imake は X システムに付随したシステムで、OS ごとに異なる状況に応じて `Makefile` を生成することができるツールで、多くの X ウィンドウアプリケーションで利用されています。

まず、`Makefile` の基になる `Imakefile` を記述ます。`Imakefile` のあるディレクトリで

```
xmkmf -a
```

を実行することで、`Makefile` を生成することができます。

Program 7.3.2 Imakefile

```
/* Imakefile for sort.cc

作成者：只木進一
作成日：1999/5/17


xmkmf -f で新しい Makefile を作成し
make でコンパイル実行
*/
SRCS=sort.cc    /* ソースファイル一覧 */

OBJS=sort.o     /* オブジェクトファイル一覧 */

CXXEXTRA_DEFINES=   /* コンパイラに渡す define */
CXXEXTRA_INCLUDES=  /* コンパイラに渡す include path */

/* 作成するターゲットの一覧*/
AllTarget(sort)
/* ソースファイルからオブジェクトファイルへのコンパイル方法の指定 */
NormalCplusplusObjectRule()
/* 実行形式ターゲットのコンパイル */
NormalCplusplusProgramTarget(sort,$(OBJS),NullParameter,NullParameter,-lm)
/* ファイルの依存関係の確認 */
DependTarget()
```

ソースファイル一覧 `SRCS` を記述すると、`xmkmf` の実行によって、各ソースファイルが依存するファイルがチェックされます。ファイルの依存関係を示す記述が `Makefile` に生成されます。あるいは、`.depend` ファイルが生成され、`Makefile` に読み込むことが指示されます。

コンパイラに渡すオプションは、マクロを定義する部分 `CXXEXTRA_DEFINES` とヘッダファイルを探すパスを指定する `CXXEXTRA_INCLUDES` に記述します。

`make` コマンドで作成されるターゲット名 `all` に相当するものをを `AllTarget` に記述します。これらのターゲットは、`make all` で生成されます。

各ターゲットのコンパイル方法を `NormalCplusplusProgramTarget` に記述します。`NormalCplusplusProgramTarget` は第一引数にターゲット、第二引数にターゲットを生成するオブジェクトファイル一覧、最後の引数に必要なライブラリを記述します。

参考書

- Andrew Oram and Steve Talbott 「Make 改定版」(オライリージャパン)
- Steve McConnell 「コードコンプリート」(アスキー)
- Paul DuBois, *Software Portability with imake* (O'Reilly & Associates)

第8章　変数のスコープと関数

8.1　変数のスコープと記憶クラス

C++ の変数には、変数が有効なプログラムの範囲を表すスコープと、変数が有効となるプログラム実行中の時間を表す記憶クラスという二つの属性があります。

8.1.1　変数のスコープ

変数のスコープ (scope) とは、変数の定義が有効なプログラム中の範囲のことです。一般には、プログラムのブロックを表す括弧{と}に囲まれた範囲内で宣言された変数は、そのブロック内だけで有効になります。このように、有効な範囲を限定された変数を局所変数 (ローカル変数、local variables) と呼びます。

関数は、プログラムブロックの一つです。関数の引数を含めて、関数の内部で宣言された変数は、その関数の中だけで有効です。`for` や `while` ループもプログラムブロックです。ループの中で宣言された変数はそのループの中だけで有効となります。

例としてプログラム 8.1.1 を考えましょう。関数 `main` の中で宣言されている変数 `a` と `b` は、関数 `main` の中だけで有効な変数です。関数 `swap` の引数 `aa` と `bb` は、関数 `swap` を呼び出している `main` の中の変数 `a` と `b` とは関係無い、独立な変数であることに注意が必要です。後述するように関数を呼び出す際には、変数の値の写しが呼び出された関数に渡され、呼び出した側の変数と呼び出された側の変数は全く別のものとして扱われます。

関数ブロックの外側で宣言される変数を、大域変数 (グローバル変数、global variables) と呼びます。大域変数は、全てのブロック内で使用することができます。関数の引数として渡せない変数を共有したり、OS などによって規制されている、プログラム実行中で変更されない定数などに利用します。

大域変数と同じ名前を局所変数として宣言すると、局所変数の宣言が優先され、大域変数は隠蔽されます。しかし、このような名前の重複はプログラムを読みにくくする原因となるので、避けて下さい。

変数宣言を面倒臭がって大域変数として宣言してはいけません。変数の内容を思いがけない形で変更してしまう危険性があるからです。局所的に使う変数は、そのブロック内の局所変数として宣言すべきです。複数の関数で変数を共有する必要がある場合には、引数として渡す工夫をします。

Program 8.1.1 swap.cc

```
//*** swap.cc **********
//
// 関数とスコープ (第 7 回講義資料)
// 入力の並べ変え
// 作成日:1999/5/27
//*************************
#include <iostream.h>
// 二つの整数を入れ換える関数のプロトタイプ宣言
// inline 宣言をしているので、引用箇所に埋め込まれる
// 引数が参照型であることに注意
inline int swap(int &,int &);

int main(int argc,char** argv)
{
  //データ入力
  int a,b;
  cout <<"二つの整数の入力 ";
  cin >>a>>b;

  if( a < b ) { // a<b の場合 a と b を入れ換える
    swap(a,b);
  }

  cout << a << " " << b << "\n";
  return 0;
}

// a と b を入れ換える関数の定義
inline int swap(int &aa,int &bb){
  int c=aa;
  aa=bb;
  bb=c;
  return 1;
}
```

8.1.2 変数の記憶クラス

変数の有効な時間を示す属性が記憶クラスです。変数の記憶クラス (strage class) には、恒久的クラス (static) と一時的クラス (automatic) があります。

恒久的クラスの変数は、プログラムの実行開始時あるいは、その変数が始めて宣言された際に、生成され初期化され、プログラムの終了時まで存在します。途中で値が変更されれば、その値が保持されます。

大域変数は恒久クラスとして扱われ、プログラムの実行開始から終了まで存在します。局所変数は、静的であることを示すキーワード `static` が付いている場合に静的局所変数と呼ばれ、恒久クラスとして扱われます。静的局所変数は、有効範囲はその変数が宣言されたブロック内に限定されています。しかし、プログラムの動作がそのブロックを離れている間も直前の値が保持され、プログラムの動作がそのブロックに戻ると保持されていた値を使うことができます。

大域変数にキーワード static を付けると、若干異なる意味を持つので注意が必要です。恒久的大域変数であることに変化はありません。しかし、その有効範囲は、その変数が宣言されたプログラムファイルの中に限定されます。

通常の局所変数は、全て一時的記憶クラスです。一時的記憶クラスの変数は、プログラムの動作がそのブロックに入った際に、スタックと呼ばれる記憶領域が割り当てられ、その中で初期化され有効となります。そのブロックが終了すると、スタックに割り当てられた領域は開放され、一時的記憶クラスの変数は消去されます。

演習 8.1 プログラム 8.1.1 において、二つの整数が大きい順であれば、入れ換えず、小さい順であれば入れ換えるように、関数 swap を呼び出す条件部分を、関数 swap の中で行うように書き換えなさい。

8.2 関数

8.2.1 関数とは

Program 8.2.1 control.cc その 1

```
//*** control.cc **********
//
// 関数とスコープ (第 7 回講義資料)
// 非負の整数の和、平均、分散
// 作成日:1999/5/27
// コンパイルには-lm オプションが必要
//     g++ control.cc -lm
//*************************
#include <iostream.h>
// 平方根を求める関数 sqrt を使うためのヘッダファイル
#include <math.h>

// 関数のプロトタイプ宣言
int getdata(int data[],const int);
double calc_mean(const int data[],const int);
double calc_rms(const int data[],const int);
void showresult(const int data[],const int,const double,const double);
```

C++ は関数型のオブジェクト指向プログラミング言語です。C++ では、操作は全て関数 (functions) というまとまりで記述していきます。例えば、プログラム 8.1.1 で使った関数 swap は、入れ換えの行える変数の型が整数型であることを除けば、様々な状況で使うことができる汎用的な関数です。このように、汎用的な操作を関数として作ることができれば、他のプログラムでも再利用することができます。

C++ の関数は、数学の関数と同様に引数 (arguments) を持ち、戻り値 (return value) を返します。一般に、次のような形式で宣言されます。

型名 関数名 (引数並び)

```
{
  関数の本体の記述
}
```

関数の値は、return を使って、その関数を呼び出した箇所に返されます。

Program 8.2.2 control.cc その 2

```
// プログラムの主部
int main(int argc,char** argv)
{
  const int NumMax=2048;//データの最大値
  int data[NumMax];     //データ格納用配列

  int num=getdata(data,NumMax); //データ入力

  if(num == 0){
    cout << "データが入力されませんでした。\n";
    return 0;
  }else{
    cout << "データの平均と分散を計算します。\n";
  }

  double mean=calc_mean(data,num); // 平均の計算
  double var =calc_rms(data,num);  // 分散の計算

  showresult(data,num,mean,var);
  return 0;
}

int getdata(int data[],const int NumMax)//データ入力関数
{
  int data_dum,num=0;
  cout << "データ入力（整数）：負の入力で終了\n";
  cin >> data_dum;
  while(data_dum>=0 && num < NumMax){//正の値である限り、データとして保存する
    data[num] = data_dum;
    num ++;
    cin >> data_dum;
  }
  return num;//データ数を返す
}

double calc_mean(const int data[],const int num)//平均を計算する関数
{
  int sum=0;;
  for(register int i=0;i<num;i++){
    sum += data[i];                  //データの和
  }

  return sum/double(num);// 平均を返す
}
```

Program 8.2.3 control.cc その 3

```
double calc_rms(const int data[],const int num)//分散を計算する関数
{
  int square_sum=0;
  for(register int i=0;i<num;i++){
    square_sum += data[i]*data[i];   //データの平方和
  }
  double mean=calc_mean(data,num);
  return sqrt(square_sum/double(num) -  mean*mean);//分散を返す
}

void showresult(const int data[],const int num,const double mean,const double var)
// 結果の出力の関数
{
  char id='n';
  while(id!='q'){
    cout <<"**********************************\n";
    cout <<"結果出力\n";
    cout <<"以下の命令のいずれかを入れて下さい。\n";
    cout <<"n : データ数\n";
    cout <<"m : 平均\n";
    cout <<"v : 分散\n";
    cout <<"l : データ一覧\n";
    cout <<"q : 終了\n";

    cin >>id;
    cout <<"**********************************\n";
    switch(id){
    case 'n':
      cout << "データ総数 : "<<num<<"\n"; break;
    case 'm':
      cout << "平均 : "<<mean<<"\n"; break;
    case 'v':
      cout << "分散 : "<<var<<"\n"; break;
    case 'l':
      for(register int i=0;i<num;i++)cout << data[i]<<"\n";
      break;
    case 'q':
      break;
    default:
      cout <<"正しい番号を入れて下さい。\n\n";
    }
  }
}
```

プログラム 8.2.1 は平均と分散を計算するプログラムです。ここで使われている関数 `getdat` は、データを配列 `data` に読み込み、入力されたデータの数を値として返します。

ここでの例のように、関数へ渡す引数は、整数や浮動小数点型のような基本となる型だけでなく、配列を始めとする様々な型のものを渡すことができます。

8.2.2　関数プロトタイプ

宣言されていない変数がプログラム中に現れると、コンパイラはその旨をエラーメッセージとして出力しコンパイルを中止します。一方、宣言されていない関数が現れても、コンパイラは警告のエラーメッセージ (warning) を出し、その関数の型を int と仮定してコンパイルを継続してしまいます。つまり、呼び出し側との関数の型の整合性や、引数並びの整合性などをチェックしてくれません。これらの不整合性は、ライブラリとの結合時や実行時に深刻な障害を起こします。

関数の実体が定義されていなくても、関数の型と引数並びだけを宣言しておけば、コンパイラはその整合性を調べてくれます。不整合があれば、エラーとなってコンパイルは中止されます。このように、関数の型と引数並びだけを関数の実際の引用や定義に先だって宣言することを関数プロトタイピング (prototyping) と呼びます。プログラム 8.2.1 の関数 main の前での関数の宣言がプロトタイプ宣言です。

8.3　関数の引数

8.3.1　値渡しと参照型

```
int function(int a)
{
  a++;
  return a;
}

int main(int argc,char** argv)
{
int n=1;
int b=function(n);
cout <<n<<" "<<b<<"\n";
}
```

上の例で関数の引数の動きを考えて見ましょう。関数 function は一つの整数型変数を持つ関数です。関数 function に渡った変数 a は、値を 1 増やされます。しかし、main にはその値の変更を反映することができません。もちろん、変更を受けた値は return 文で main 中の変数 b へ代入されます。

関数 function の引数 a は、この関数の中だけで有効な局所変数です。C++ では、このような形での引数定義の場合、関数 function が呼ばれるたびに、そのためのスタック領域が作成され、呼び出し側の関数の引数 (main の中での n) の値が a にコピーされます。呼び出し側 n と関数内の変数 a は別の領域に保存されていることに注意してください。このような、C++ での引数の扱いを「値によるコール」と呼びます。呼び出し側 n と関数内の変数 a は別のものですから、関数が終了すると a は消去され、n には何も影響を与えません。

プログラム 8.1.1 では、関数に引数を渡して中を変更できていました。関数 swap の引数に記号

&があることに注意してください。引数名に&がある変数は、参照型 (references) と呼ばれます。参照型は、それ自身で独自の記憶領域を持たず、参照される変数と領域を共有しています。関数の引数として現れると、「値によるコール」の場合と異なり、コピーは作成されません。従って、関数の内部での値の変更は、呼び出し側に反映されます。

上述の例では

```
int function(int& a)
{
  a++;
  return a;
}
```

とすることで、内部での変更を反映することができます。

8.3.2 const 引数

関数の宣言や関数への変数渡しに際して、特に内部での変更を禁じておきたい場合があります。その場合に、変数に const を指定することで、コンパイラに変更が行われないことを知らせることができます。

通常の値渡しの場合は、特に必要がありませんが、参照型や後述する配列を引数として渡す場合には、関数の中で変更されてしまう恐れがあります。それを禁じるのが目的です。

8.3.3 配列

配列を関数の引数として使うこともできます。この場合、配列の先頭アドレスが渡されます。従って、通常は配列の内容を変更すると、呼び出し側に反映します。配列のサイズが必要な場合には、別変数として渡します。

8.4 関数の多重定義

C++ では、関数を、その名前と引数並びの型で区別します。つまり、同じ名前の関数でも引数並びの異なる関数は、別の関数として区別することができます。例えば

```
int swap(int& a,int& b);
int swap(double& a,double& b);
```

のように、整数型の値を入れ換える関数と浮動小数点型の値を入れ換える関数を同じ名前 swap で定義することが可能です。

8.5 デフォルトパラメタ

C++ では、関数の引数の一部 (引数並びの終端部) を省略して引用することができます。例えば、

```
int function(int a,int b=1);
```

と宣言された関数の場合、二番目の変数を省略して引用すると、`b=1` が代入されて引用されます。
　便利な機能ですが、引用時にパラメタの指定忘れなのか、意図してデフォルト値を使っているのかが判別できない恐れがあります。乱用は謹みましょう。

8.6　`inline` 関数

　関数の呼び出し時には、関数用のスタック領域が確保され、引数の値がコピーされます。関数の終了時には、スタックの開放が行われます。つまり、関数の内容が実行される前後に別の作業 (オーバーヘッド、overhead、と呼びます) が必要です。関数が小さい場合、関数本体よりも前後の作業のコストが大きくなってしまうことが起こります。しかし、関数を使わずに書くと、プログラムが読みにくいなど、プログラム開発のコスト等が大きくなってしまう恐れがあります。
　関数の前に `inline` というキーワードを書くと、コンパイラは関数を呼び出し場所にその関数を展開してからコンパイルしようとします。つまり、プログラムでは別の関数として記述しているものが、その場所に書かれたようにコンパイルが行われます。これをインライン展開と呼びます。
　インライン展開が行われれば、スタックの確保などのオーバーヘッドが無くなります。インライン展開が可能なのは、短い関数に限られています。

8.7　再帰的関数

　関数が、その内部でその関数自身を呼ぶ場合、その関数を再帰的関数 (recursive function) と呼びます。例えば、自然数 n の階乗は $n-1$ の階乗で定義されます。つまり、次のように再帰的に書くことができます。

$$n! = n \times (n-1)!, \qquad 0! = 1$$

関数 $n!$ は同じ関数で引数の異なる $(n-1)!$ で定義されています。このように、自然数を引数とする関数の中には、再帰的な定義を持つものが多数あります。
　数学的な関数でなくても、再帰的な関数や操作で定義されるものも多数あります。例えば、UNIX のファイルシステムで、ホームディレクトリ以下の全てのファイル名を列挙する場合、各ディレクトリにおいて、その直下のファイル名を列挙するとともに、下位のディレクトリに同様の操作を行うことでファイル名の列挙をおこなうことができます。
　このように再帰的操作や再帰的関数は非常に有用です。C++ でも再帰的関数を定義することができます。再帰的関数の定義に当たっては、必ず停止するように書くことに注意が必要です。階乗の場合には、停止条件 $0! = 1$ が与えられています[1]。
　例として、階乗を計算する C++ のプログラム 8.7.1 を考えましょう。関数 `factorial` は再帰的に定義されている関数です。整数引数 `n` が 0 ならば、$0! = 1$ ですから、1 を返します (`if(n==0)return1`)。0 より大きい場合には、引数の値を 1 だけ減らしながら、再帰的に関数を呼び出します。

```
return n*factorial(n-1)
```

[1] 自然数以外の階乗については適当な数学の定義を参照してください。

Program 8.7.1 factorial.cc

```
//*** factorial.cc **********
//
// 関数とスコープ (第 7 回講義資料)
// 再帰的関数定義
// 整数の階乗
// 作成日:1999/5/27
//*************************
#include <stdlib.h>
#include <iostream.h>
// 階乗の宣言
int factorial(int);

int main(int argc,char** argv)
{
  //データ入力
  int a;
  cout <<"整数の入力 ";
  cin >>a;

  cout << a << "!=" <<factorial(a)<< "\n";
  return 0;
}

// a の階乗
int factorial(int n){
  if(n<0){
    cerr<<"factorial of negative number "<<n<<" is not defined!\n";
    exit(0);
  }

  if(n==0)return 1;
  return n * factorial(n-1);
}
```

演習 8.2 再帰的関数は階乗のように、一つだけ自分自身を呼び出す形に限定されているわけではありません。二項係数 (組み合わせの数) を計算する場合を考えましょう。二項係数とは、n 個から r 個を選び出す組合せの数 ${}_nC_r$ と同じものです。二項係数は

$$\begin{pmatrix} n \\ r \end{pmatrix} = \frac{n!}{(n-r)!r!}$$

のように階乗を使って定義することができますが、

$$\begin{pmatrix} n \\ r \end{pmatrix} = \begin{pmatrix} n-1 \\ r-1 \end{pmatrix} + \begin{pmatrix} n-1 \\ r \end{pmatrix}$$

のように再帰的に定義することもできます。再帰の終点として

$$\begin{pmatrix} n \\ 1 \end{pmatrix} = n, \qquad \begin{pmatrix} n \\ n \end{pmatrix} = 1$$

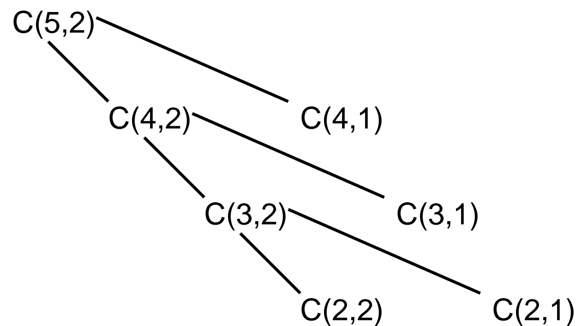


図 8.1: 二項係数計算の再帰木の構造

を定義しておきます。すると再帰的計算は図 8.1 のようになります。

二項係数を計算する関数

```
int combination(int n,int r)
```

を再帰的に定義しなさい。また、その関数を使うプログラムを作成しなさい。

演習 8.3　二項係数を計算する際に、階乗を計算する関数を利用する場合の問題点を考察しなさい。

第9章　CPP

9.1　CPPとは

CPPは、C及びC++のコンパイララッパ (compiler wrapper)、つまりコンパイルの前に実行されるプリプロセッサ (preprocessor) です。通常使っているC++のコンパイルコマンドでは、コンパイルの前に、プリプロセッサと呼ばれるエディタが実行されます。UNIXではcppと呼ばれるコマンドです。

cppは、C及びC++のプログラムソースの中で、行の先頭が#で始まる行を解釈します。しかし、cppは、C及びC++の文法について、何も考慮してくれないので、注意が必要です。逆に、CやC++以外についても使うことができます。例えば、Xのリソースを読み込むxrdbやimakeもcppを使っています。

9.2　#define

変数や小さなマクロを定義することができます。

```
#define 変数名 値
```

あるいは

```
#define 変数名
```

で行います。

このようなマクロの定義は、かっては、定数や小さな関数の定義の代わりに使われました。しかし、現在のC++環境では、定数の定義にはconstを、小さい関数の定義にはinlineを使うべきとされています。

9.3　条件コンパイル

cppの使い方で、もっとも有用なものの一つが、条件コンパイルです。つまり、OSなどの条件に応じてソースを選択してコンパイルさせるものです。

CやC++は、移植性が良い (可搬性が高い、portable) と言われますが、OSやコンパイラに依存した部分が存在しています。こうした、OS等への依存性は、コンパイラそのもの、あるいはシステムのヘッダファイルの中で記述されています。どのような変数が#defineの形式で定義されているかは、

```
g++ -E -dM ソースファイル
```

などで知ることができます。

システムへの依存性だけでなく、デバッグ用コードの追加などでも、#define は有用です。

ソースコード中では

```
#ifdef 変数 0
  ソース 1
#else
  ソース 2
#endif
```

という形式で、ソースコードを選択することができます。変数 0 が定義されてされていれば、ソース 1 がコンパイルされ、定義されていない場合にはソース 2 がコンパイルされます。プログラム 9.5.1 では、DEBUG という変数が定義されている場合は、途中経過を表示するコードがコンパイルされ、定義されていない場合には、その部分を除いたコードがコンパイルされます。

cpp の定数の定義は、#define 文をソースコードに書くだけでなく、コンパイルオプション

```
-D 変数名
```

でも行うことができます。プログラム 9.5.1 をコンパイルする場合には、

```
g++ -DDEBUG comb.cc
```

とすることで、デバッグ用コンパイルを行うことができます。

9.4 #include

変数名、定数、関数プロトタイプなどを別ファイル (ヘッダファイル) に書き、それを読み込む際に使うのが#include 文です。

<> で指定されたファイルの場合、システムの標準ヘッダファイルのパスからファイルが探索されます。"" を使って指定された場合には、絶対パス、あるいは相対パスとして探索されます。相対パスの原点はコンパイラオプション-I で指定することもできます。

9.5 CPP を使う例

cpp を使う例として、二項係数を計算するプログラムのデバックを考えましょう。

二項係数には、前述のような漸化式

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

があります。ただし、

$$\binom{n}{1} = n, \qquad \binom{n}{n} = 1$$

が成り立っています。そこで、関数 comb を再帰的に定義します。$r = 1$ の場合と $r = n$ の場合の終了条件に注意します。

Program 9.5.1 comb.cc

```
//*** comb.cc **********
//
// 関数とスコープ (第 9 回講義資料)
// 再帰的関数定義
// 2 項係数
// 作成日:1999/6/4
//*************************
#include <iostream.h>
// 二項係数
int comb(int,int);

int main(int argc,char** argv)
{
  //データ入力
  int n,r;
  cout <<"二つの整数の入力 ";
  cin >>n>>r;
  if(n<r){
    cerr<<"C(n,r): n should be larger than r\n";
    exit(0);
  }
  if(n<=0)
    {
    cerr<<"C(n,r): n should be larger than 0\n";
    exit(0);
  }

  cout <<"C("<<n<<","<<r<< ")=" <<comb(n,r)<< "\n";
  return 0;
}

// 二項係数
int comb(int n ,int r)
{
#ifdef DEBUG  //動作を確認する際のコード
  cerr <<"Computing C("<<n<<","<<r<< ")";
  if(r==1){cerr <<"\n";return n;}
  if(r==n){cerr <<"\n";return 1;}
  cerr <<" ";
#else         //通常のコード
  if(r==1){return n;}
  if(r==n){return 1;}
#endif
  return comb(n-1,r-1)+comb(n-1,r);
}
```

プログラムのデバッグ時に、漸化式に従って、どの (n, r) の組合せが計算されているかを示すようにします。コンパイル時に

```
g++ -DDEBUG comb.cc
```

とすることでデバッグ用のコードがコンパイルされます。

もうひとつ、再帰的関数の例を見ましょう。Fibonacci 数列は漸化式

$$f_n = f_{n-2} + f_{n-1}$$

で定義されます。ただし、$f_1 = 0$ 及び $f_2 = 1$ です。Fibonacci 数列を計算する再帰的関数 `fibonacci` を定義します。

デバッグ時に、どの n について計算が行われているかを追うためのコードを用意します。

```
//*** fibonacci.cc **********
//
// Fibonacci 数列
// 作成日:1999/6/7
//*************************
#include <iostream.h>
// Fibonacci 数列のプロトタイプ宣言
int fibonacci(int);

int main(int argc,char** argv)
{
  //データ入力
  int a;
  cout <<"整数の入力 ";
  cin >>a;

  cout <<"f("<< a << ")=" <<fibonacci(a)<< "\n";
  return 0;
}

// Fibonacci 数列
int fibonacci(int n){
  if(n<1){
    cerr<<"fibonacci of negative number "<<n<<" is not defined!\n";
    exit(0);
  }

#ifdef DEBUG
  cerr <<"fibonacci("<<n<<")\n";
#endif
  if(n==1)return 0;
  if(n==2)return 1;
  return fibonacci(n-2) + fibonacci(n-1);
}
```

第10章　簡単なポインタ

10.1　ポインタとは

C++ の変数は、メモリのどこかに記憶されています。通常の変数 (整数型や浮動小数型) は、変数名を指定するとその内容が返されます。内容ではなく、その記憶領域の場所を指し示すのがポインタ (pointer) です。

通常の変数を宣言すると、値を保持するのに十分な領域が確保されます。しかし、ポインタ宣言をすると、ポインタを保持するのに必要なだけの領域しか確保されず、ホインタが示す値を保持する領域は確保されないことに注意が必要です。

```
int thing=5;
int *thing_ptr;

thing_ptr=&thing;
cout <<*thing_ptr<<"\n";
```

上の例では、thing は整数型変数として宣言され、整数が保持できる領域が確保されます。thing_ptr は整数型へのポインタとして宣言されています。この段階では、thing_ptr は何も指し示していません。

整数型変数 thing のポインタは&thing で得ることができます。代入

```
thing_ptr=&thing;
```

によって、ポインタ thing_ptr は、整数変数 thing を指し示すようになります。整数型へのポインタ thing_ptr が指し示している領域に保持されている値は*thing_ptr で得ることができます。

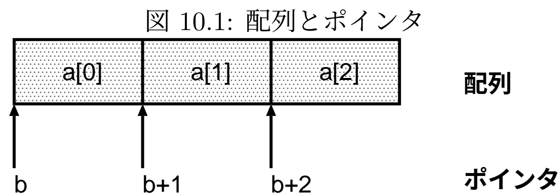
このように、通常の変数名の前に&をつけることで、その変数へのポインタを得ることができます。またポインタ変数の前に*をつけることで、そのポインタ変数が指している変数の値を得ることができます。

10.2　配列とポインタ

配列名は、配列の先頭の要素へのポインタです。従って、ポインタを使うと配列の要素へのアクセスを効率良く行うことができます。

```
int a[256];
int *b=a;
for(register int i=0;i<256;i++)cout <<*(b+i)<<"\n";
```

図 10.1: 配列とポインタ



上の例では、b は配列 a の先頭の要素へのポインタです。b+i は、配列の i 番目の要素へのポインタ、*(b+i) は i 番目の要素の内容を表します。

ポインタへの演算は、ポインタが指している変数の大きさに合わせて行われます。

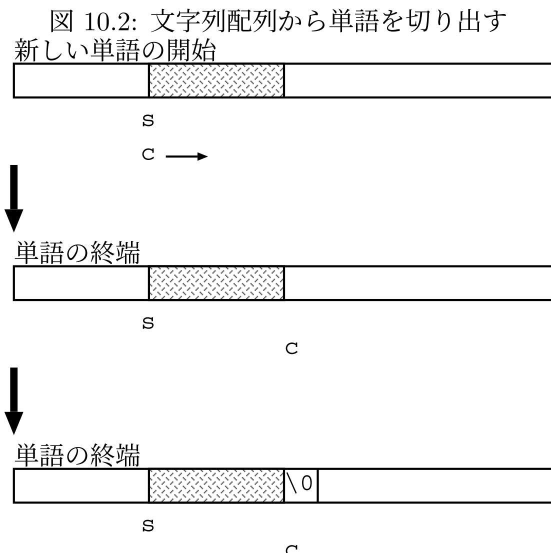
Program 10.2.1 array.cc

```
//*** array.cc **********
//
// ポインタと文字列 (第 9 回講義資料)
// 配列とポインタ
// 作成日:1999/6/8
//
//*************************
#include <iostream.h>
int show(int*);

int main(int argc,char** argv)
{
  int a[3]={9,3,1};
  int *b=&a[0];//配列の最初の要素へのポインタ

  cout <<*b<<"\n"; b++;//ポインタに対する演算
  cout <<*b<<"\n"; b++;
  cout <<*b<<"\n";

  show(a);//配列を関数に渡す
  exit(0);
}

int show(int* b)
{
  cout <<*b<<"\n"; b++;//ポインタに対する演算
  cout <<*b<<"\n"; b++;
  cout <<*b<<"\n";
  return 1;
}
```

10.3　文字列とポインタ

文字列も、文字型配列であるため、ポインタを使うことで効率良く要素にアクセスすることができます。

図 10.2: 文字列配列から単語を切り出す



文字列配列から単語を切り出す場合を考えましょう (図 10.2)。文字列配列 `line` には、スペースで区切られた単語が入っているとします。単語の先頭を指し示すポインタを `s` として、そこから右へ空白か文字列の終端`\0` を見付けるまでポインタ `c` を移動します。

空白を見付けた場合には、`c` の位置に`\0` を置きます。この状態で `s` をプリントすると、`s` から `c` の位置までの文字列が印刷されます。

次の単語を探すために、現在の `c` の位置の次の位置に `s` と `c` を移動して、再び空白か`\0` を探します。

見付けた区切りが`\0` の場合には、単純に `s` をプリントして、作業を終了します。

以上を繰り返すことで、入力された一行から単語を切り出すプログラムをプログラム 10.3.1 に示します。また、対応する PAD 図を図 10.3 に示します。

10.4　記憶領域の動的割り付け

通常の変数の場合には、記憶されるものに応じたサイズの記憶領域が確保されます。しかし、ポインタを宣言した場合には、ポインタの内容、つまり指し示す先の番地を保持するための領域だけが確保されます。

これまでの例では、通常の変数が先に宣言され、それを指し示すポインタを定義しました。しかし、ポインタを宣言し、それに対応する記憶領域を動的に作成することもできます。「動的」とは、あらかじめプログラム作成時に大きさを指定するのではなく、プログラムの実行中に必要な大きさ

Program 10.3.1 words.cc

```
//*** words.cc **********
//
// ポインタと文字列 (第 9 回講義資料)
// 単語の切り出し
// 作成日:1999/6/8
//*************************
#include <iostream.h>

int main(int argc,char** argv)
{
char line[256];
cout <<"一行文字列入力\n";
cin.getline(line,sizeof(line));

char* c=line;                          //入力の最初を指すポインタ
while(1){
  char* s=c;                           //単語の最初を指すポインタ
  while((*c!=' ') && (*c!='\0'))c++;   //単語の終りまでポインタを進める
  if(*c=='\0'){                        //最後の単語ならば印刷して終了
    cout <<s<<"\n";
    break;
  }
  *c='\0';                             //単語を区切る
  cout <<s<<"\n";                      //区切りまでの印刷
  c++;
  while(*c==' ')c++;                   //次の単語の最初へ移動
  }
  exit(0);
}
```

の記憶領域を確保することを言います。

例えば、 5 個の整数型の要素を持つ配列へのポインタは次のように作成することができます。

```
int* b = new int[5];
```

new は、動的記憶領域割り付けを行う命令です。

```
new 変数型 [サイズ];
```

という形式で使います。静的な配列の宣言では配列のサイズは定数でなければなりませんでしたが、動的割り付けでは、サイズは変数で構いません。

動的に割り付けた領域は削除することができます。配列の場合は

```
delete [] 配列名;
```

です。サイズを指定しない [] は、削除しようとしているポインタが指していたのが配列であることをコンパイラへ示します。

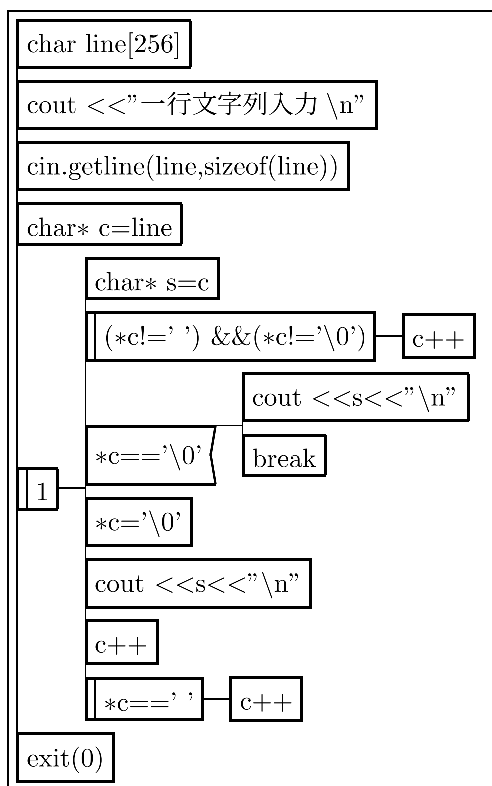


図 10.3: 文字列配列から単語を切り出す (PAD)

10.5 コマンドライン

ポインタへの動的記憶領域割り付けの例として、コマンドラインオプションとして指定されたファイルの各行を配列に保存するプログラムを作りましょう (図 10.4)。

プログラムの実行時のコマンドラインオプションは `main` 関数の二つの引数として渡されます。これまでの `main` 関数の例にも、整数型の引数 `argc` と文字型のポインタのポインタ `argv` がありました。`argc` には、コマンドラインから与えられたオプションの数が入っています。ただし、コマンド名そのものも数えるので、常に 1 より大きな値が入ります。何もオプションを渡さない場合が、1 です。

`argv` には、オプションそのものが入ります。文字型のポインタのポインタであるのは、文字列 (文字型のポインタ) の配列であるためです。`argv[0]` にはコマンド名そのものが、`argv[1]` には最初のオプションが、以下オプションが順に入ります。このプログラムでは、最初のオプションに指定されたファイルを開けることにします。つまり、ファイル名は `argv[1]` に入っています。

UNIX にはオプションを処理する関数 `getopt` が用意されていますが、ここでは扱いません。詳しくは

```
man 3 getopt
```

で調べてください。

10.6　ifstream

Program 10.6.1 getline.cc

```
//*** getline.cc **********
//
// ポインタと文字列 (第 9 回講義資料)
// ファイルの内容を配列に保存する。
// 作成日:1999/6/8
//*************************
#include <stdlib.h>
#include <fstream.h>
#include <string.h>

int main(int argc,char** argv)
{
  if(argc!=2)exit(0);// オプション数の確認

  ifstream fin(argv[1]); //最初のオプションをファイルとして、開ける

  int n=0;//行数カウンタ
  const int size=1024;
  char buf[size];
  while(fin.getline(buf,size))n++;   //行数をカウント

  fin.close(); fin.open(argv[1]);    //ファイルの終端に戻る

  char** lines=new char*[n];         //各行を保存するための配列

  for(register int i=0;i<n;i++){
    fin.getline(buf,size);
    lines[i]=strdup(buf);
  }

  for(register int i=0;i<n;i++)//配列の内容を出力
    cout <<lines[i]<<"\n";

  for(register int i=0;i<n;i++)// 各行の領域を開放
    delete [] lines[i];

  delete [] lines;             //行全体の領域を開放

  exit(0);
}
```

ファイルからの入力は cin の代わりに、ifstream を使います。これについての詳細は後述します。

ifsteram fin(ファイル名);

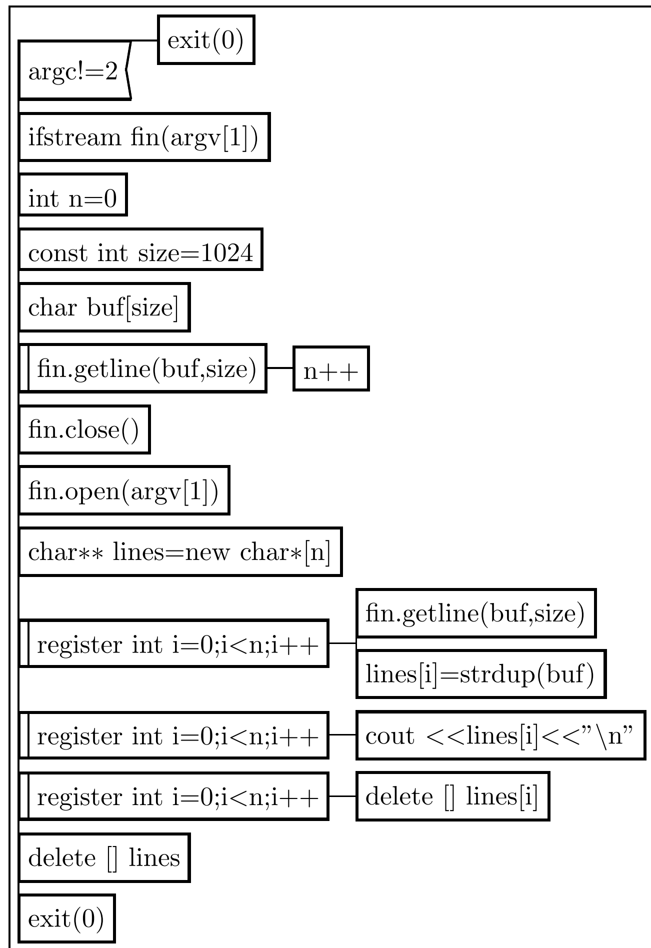


図 10.4: ファイルの各行を配列に保存する

でファイルを開きます。すると、`fin` を `cin` の代わりに使うことができるようになります。

入力するファイルの行数は、あらかじめ分かりません。そこで、行数を始めに数えます。ファイルの行数を `getline` を使って数えて、文字列の配列 `lines` を確保します。配列 `lines` は文字列という文字型配列の配列である点に注意してください。

ファイルは、終端に向かって一方向にしか読めません。そこで、行数を数え終ったのち、ファイルを一旦閉じて、再び開けます。その後、データを一行ずつ読み込み、各行を、配列 `line` の各要素に文字列をコピーする関数 `strdup` を使って保存します。

`strdup` を使わずに単純に代入すると、配列 `line` のポインタが代入されます。次の行を読むと `line` の内容が変更されますから、ポインタが指していた先も変更されてしまうことに注意します。

第11章　構造体(その1)

11.1　構造体とはなにか

複数のデータを関連付けることを考えましょう。例えば、コンピュータの中で、色は、赤、緑、青の三つの色の配合で表されています。例えば、X ウィンドウシステムでは、 色の各要素は、0 から 255 までの整数値で表されます。そこで、三つの要素を一つにまとめて、一つの色の名前を付けることができれば、プログラムが簡潔に記述できます。

複数のデータをまとめる方法の一つが配列です。例えば、オレンジ色は赤 255、緑 165、青 0 で表されます。これを

```
int orange[]={255,165,0};
```

と一つの配列に保存すれば良いでしょう。しかし、この配列の要素 0、1、及び 2 がそれぞれ赤、緑、青に対応していることは、プログラムを見ただけでは分かりにくくなっています。また、色の名前の文字列を関連付けて保存することは、この配列ではできません。配列は、同じ型の要素の組しか許さないからです。

オレンジ色の赤の成分は 255、緑の成分は 165、青の成分は 0 、名前は `Orange` とプログラムの中で明示することができればもっと見やすいプログラムを作ることができます。

異なる型も含めて複数のデータを一つのグループとして関連付けて保存するのが構造体 (structure) です。構造体を構成する要素変数をメンバ変数 (member variables) と呼びます。配列とは異なり、メンバ変数は同じ型である必要はありません。また、ポインタを含んでいても構いません。

構造体は、次のように定義します。

```
struct 構造体名{
  メンバ変数の定義
  ....
};
```

構造体には名前を付けます。構造体の名前へ変数の型に対応するものです。具体的な値を保持するためには、

```
struct 構造体名 変数名;
```

と、構造体名と同じデータ構造を持つ変数を宣言します。

具体的な値の宣言に際して、いちいち `struct` を付けるのは面倒です。C++ には、プログラムの中で新しい変数型を宣言する `typedef` という命令があります。`typedef` を使って構造体を新しい変数型として宣言することができます。

typedef 元の型 新しい型

Program 11.1.1 color.cc

```
//*** color.cc **********
//
// 構造体 (第 10 回講義資料)
// 簡単な構造体 (色を表す構造体)
// 作成日:2001/6/16
//
//*************************
#include <iostream.h>

struct cl {//構造体の定義
  int r;
  int g;
  int b;
  char *name;
};

typedef struct cl Color;//新しい型の定義

//色の定義
const Color NavyBlue={0,0,128,"Navy Blue"};
const Color IndianRed={205,92,92,"Indian Red"};
const Color Orange={255,165,0,"Orange"};

int main(int argc,char** argv){

  cout <<NavyBlue.name<< " : "
    <<NavyBlue.r<<" "<<NavyBlue.g<<" "<<NavyBlue.b<<"\n";
  cout <<IndianRed.name<< " : "
    <<IndianRed.r<<" "<<IndianRed.g<<" "<<IndianRed.b<<"\n";
  cout <<Orange.name << " : "
    <<Orange.r<<" "<<Orange.g<<" "<<Orange.b<<"\n";

  Color SkyBlue={135, 206, 235, "Sky Blue"};

  //色構造体へのポインタ
  Color *skyblue=&SkyBlue;
  cout <<skyblue->name << " : "
    <<skyblue->r<<" "<<skyblue->g<<" "<<skyblue->b<<"\n";

  Color const *orange=&Orange;
  cout <<orange->name << " : "
    <<orange->r<<" "<<orange->g<<" "<<orange->b<<"\n";
  exit(0);
}
```

色を表す構造体を使った簡単なプログラムを Program11.1.1 に示します。ここでは、色を表す構造体及びそのポインタを定義し、メンバ変数をプリントしています。

色を表す構造体の場合は、次のようなものを考えればよいでしょう。

```
struct cl {//構造体の定義
  int r;
  int g;
  int b;
  char *name;
};
```

色を表す三つの要素には、r、g、b と赤緑青を連想させる名前がついています。もっと直接的に red、green、blue と名前を付けても良いでしょう。変数 name には色の名前を保存することができます。cl は構造体の名前です。

上の構造体を宣言する場合には、

```
struct cl orange;
```

などと宣言します。いちいち struct を付けるのは面倒なので、構造体を新しい変数型として宣言します。

```
typedef struct cl Color;
```

構造体は変数型 Color として宣言されました。このようにすると

```
const Color Orange={255,165,0,"Orange"};
```

のような形で定数の初期値を与えることもできます。

構造体の要素へのアクセスは

構造体名.メンバ変数名

という形で表します。例えば Orange.r はオレンジ色の赤の要素の値に対応します。

構造体は、通常の変数型と同様に扱われますから、そのポインタも定義できます。構造体へのポインタの場合のメンバへのアクセスは . の代わりに->を使います。プログラムで、色 Sky Blue に対しては単純にポインタが構成されるのに対して、色 Orange に対して

```
  Color const *orange=&Orange;
```

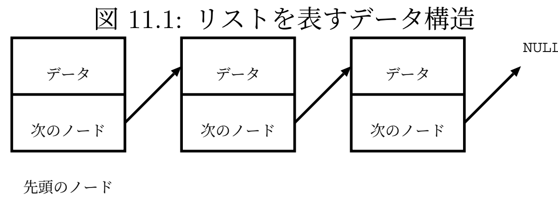
と、const が付いているのは、変数 Orange に定数宣言が行われているためです。

11.2 リスト

要素が一列に繋がったものをリスト (list) と呼びます。配列との相違点は、あらかじめ要素数の上限が定められていない点です。

リストをプログラムとして実装する方法を考えましょう。配列のように、あらかじめ要素数に応じた領域を確保しておくことができませんから、必要に応じてデータを置く場所を確保することになります。つまり、要素は連続した領域に置かれるわけではありません。このように、リストの場合、連続した領域にデータが配置されませんから、データの配置情報も保持する必要があります。

図 11.1: リストを表すデータ構造



データと、次のデータへのポインタをメンバ変数とする構造体 node を繋いだものとしてリストを構成しましょう 11.1。各 node には、保存されるデータそのものと、次の node へのポインタが保存されます。

新しいデータをリストに追加するには、終端に新しい node を追加することで行うことにします。終端の node は次の node として NULL を指しておきましょう。

このリストを利用するプログラムは、リストの先頭の node へのポインタだけを持つことで、リストへアクセスすることができます。

構造体の定義としては

```
struct node {//データを保存するノードの構造体
  int data;           //保存するデータ
  struct node * next;  //次のノードへのポインタ
};

typedef struct node Node; //構造体を新しい型として定義する。
```

とすれば良いでしょう。今回は、整数を保存するリストを考えるので、data は int 型にしておきます。

要素の追加は関数 append で行います。要素の追加のためには、リストの終端に移動しなければなりません。そのため、リストの先頭から、終端まで、再帰的に移動します。終端であることの判断はそのノードのポインタが NULL になっていることで行います。終端まで移動したら、新しいノードを new を使って作成し、次のノードへのポインタを NULL としておきます。

```
    n=new Node;
    n->data=a;
    n->next=NULL;
```

現在のノードへのポインタを戻り値としていることに注意してください。append を呼んだメインプログラムには、再帰的呼び出しを逆に辿って、リストの先頭要素へのポインタが戻ります。

リストの中身を出力する関数 show も再帰的に定義されています。現在のノードに保存された値を出力し、次のノードの出力へ移動します。終端まで再帰的に繰り返します。

終了時に、動的に確保した記憶領域を再帰的に削除するのが、関数 deletelist です。動的に確保した記憶領域は、明示的に解放されるか、使用したプログラム全体が終了するまで、解放されません。

Program 11.2.1 list.cc

```
//*** list.cc **********
//
// 構造体 (第 10 回講義資料)
// 簡単な構造体 (リスト)
// 作成日:2000/6/12
//
//*************************
#include <iostream.h>

struct node {//データを保存するノードの構造体
  int data;             //保存するデータ
  struct node * next;  //次のノードへのポインタ
};

typedef struct node Node; //構造体を新しい型として定義する。

Node * append(Node *,int);
void show(Node *);
void deletelist(Node *);

int main(int argc, char** argv)
{
  Node *list=NULL; //リストの先頭
  int a;
  cout <<"Input integers (negative to quit)\n";

  cin >>a;
  while(a>=0 ){
    list=append(list,a);
    cin >>a;
  }

  show(list);
  exit(0);
}
```

11.3 リスト (その 2)

リストへの新しい要素の追加は、常にリストの終端に行います。従って、毎回リスト終端を探すよりも、常にリストの終端が分かっていれば、効率良く作業を行うことができます。しかし、それはリスト操作の内部の問題であって、main から終端のノードが見えるようにするのは、好ましくありません。

そこで、リストの先頭と終端のノードへのポインタだけを持つリストを表す構造体 List とリスト内のノードを表す構造体 Node の二つに分けて階層化し、main からはリスト構造体だけが見えるようにしましょう。

まずリストを表す構造体を宣言します。構造体の宣言と typedef を一度に行いましょう。

```
typedef struct {//リストを保存する構造体
```

Program 11.2.2 list.cc(その 2)

```
Node * append(Node *n,int a){//要素を追加する
                              //最後のノードを再帰的に探す
  if(n==NULL){//新しいノードの生成
    n=new Node;
    n->data=a;
    n->next=NULL;
  } else
    n->next = append(n->next,a);
  return n;//ノードのポインタを返す
}

void show(Node *n){//リストの先頭から出力する
  if(n!=NULL){
    cout << n->data << "\n";
    show(n->next);
  }
}

void deletelist(Node *n)//確保した記憶領域を開放
{
  if(n!=NULL){
    deletelist(n->next);
    delete n;
  }
}
```

```
  Node *top;
  Node *tail;
} List;
```

新しい型 List は先頭と終端のノードへのポインタだけを保持しています。リストの初期値は、先頭と終端のノードのポインタが NULL を指している状態です。そのような初期化を initlist で行います。この関数はリスト構造体の内容を書き換えますから、引数は List への参照型となっていることに注意して下さい。

リストへの新しい要素の追加は、createnode によって新しいノードを作成し、それをリストの終端に追加します。終端の付け替えが必要なことに注意してください (図 11.2)。

```
Node *n=createnode(a);
l.tail->next=n;
l.tail=n;
```

構造体 List がリストの終端を知っているので、前節の場合のように再帰的に終端を探す必要がありません。

リストに格納された要素の表示は、前節では再帰的に行いましたが、今回は while を使った繰り返しで行っています。

終了時に、main からリストの消去を行う関数 deletelist を呼びます。実際には、その下で再帰的にノードを消去する関数 deletenode が動きます。

Program 11.3.1 list2.cc

```
//*** list.cc **********
//
// 構造体 (第 10 回講義資料)
// 簡単な構造体 (リスト、その 2)
// 作成日:2000/6/14
//
//*************************
#include <iostream.h>

struct node {//データを保存するノードの構造体
  int data;            //保存するデータ
  struct node * next;  //次のノードへのポインタ
};

typedef struct node Node; //構造体を新しい型として定義する。

typedef struct {//リストを保存する構造体
  Node *top;
  Node *tail;
} List;

//関数のプロトタイプ宣言
int initlist(List &);
int append(List &,int);
Node * createnode(int);
void show(List);
void deletelist(List &);
void deletenode(Node *);
```

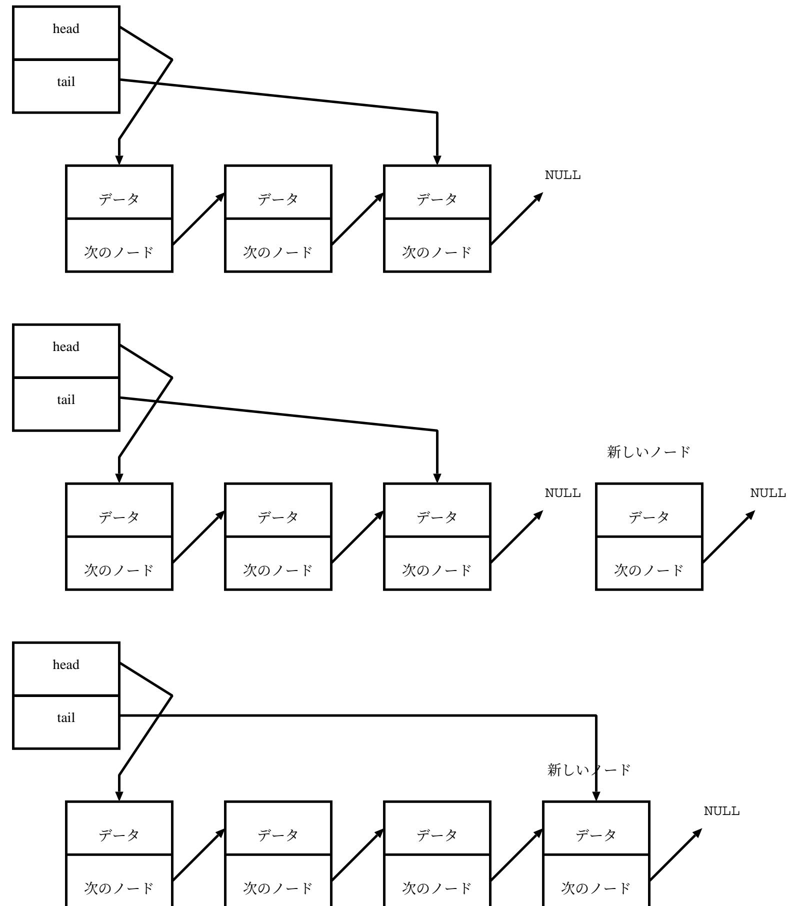
本節の場合、main からはリストを表す構造体だけが使われ、ノードの構造体は見えていません。このようにすることで、リスト構造が実際にどのように操作されているかを気にせずにプログラムができるようになります。

今回の例では、リスト構造体が終端をデータとして知ってるために、新しい要素の追加が効率良く行えます。リスト構造体を使うプログラマは知る必要がありません。

リストの要素数を数えることを考えましょう。要素数を求められた際に実際に要素数を数えることもできますが、要素を追加する際に数え、リスト構造体の中に保持しておくこともできます。もちろん、後者のほうが効率が良くなります。どちらの実装であっても、リスト構造体を使うプログラマはプログラムを変更する必要はありません。

このように、構造体を階層的に使うことで、情報を隠蔽することができるようになります。

図 11.2: リストへの要素追加



Program 11.3.2 list2.cc(その 2)

```
//*****************************************************
int main(int argc, char** argv)
{
  List list;
  initlist(list);
  int a;
  cout <<"Input integers (negative to quit)\n";

  cin >>a;
  while(a>=0 ){
    append(list,a);
    cin >>a;
  }

  show(list);
  deletelist(list);
  exit(0);
}

int initlist(List &list)//リストを初期化する
{
  list.top=NULL; //リストの先頭
  list.tail=NULL;// リストの終端
  return 1;
}

int append(List &l,int a)//要素を追加する
{
  if(l.top==NULL){// リスト先頭が NULL の場合
    Node *n=createnode(a);
    l.top=n;
    l.tail=n;
  } else {//リストの終端に新しい要素を追加する
    Node *n=createnode(a);
    l.tail->next=n;
    l.tail=n;
  }
  return 1;
}
```

Program 11.3.3 list2.cc(その 3)

```
Node * createnode(int a)//新しいノードを作る
{
  Node *n=new Node;
  n->data=a;
  n->next=NULL;
  return n;
}

void show(List l)//リストの先頭から出力する
{
  Node *n=l.top;
  while(n!=NULL){
    cout << n->data << "\n";
    n=n->next;
  }
}

void deletelist(List &l)//確保した記憶領域を開放
{
  deletenode(l.top);
}

void deletenode(Node *n)//確保した記憶領域を開放
{
  if(n!=NULL){
    deletenode(n->next);
    delete n;
  }
}
```

第12章　構造体(その2)

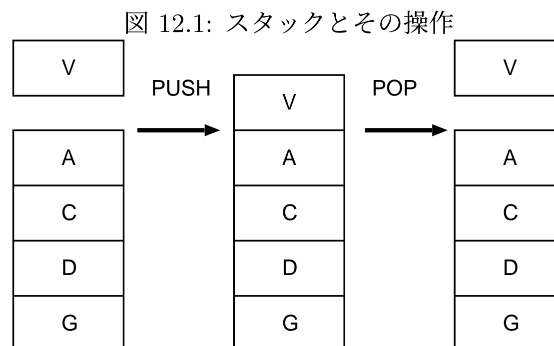
12.1　スタック

リストは、データが一列に並んでいました。スタック (stack) も同様にデータが一列に並んでいますが、アクセスに制限があります。スタックは「先入れ後出し」、つまり先にスタックに入力したものは、後から取り出さなくてはいけないという制限のついたデータ構造です。

リストはデータが横に一列に並んでいましたが、スタックは皿を積み重ねるようにデータが縦に並んでいると考えて下さい。データの追加は一番上に皿を積み重ねること、データの読みだしは一番上のデータ (皿) を取り出すことに対応します。

新しい皿 (データ) を積み重ねることを push、一番上の皿 (データ) を取り出して、そのデータを読み出すことを pop と言います。データを読み出すには、必ず pop しなければなりません。

図 12.1: スタックとその操作



本章では、スタックを使って、コマンドラインのオプションとして指定された文字列を反転するプログラムを例に、構造体の使い方を学びましょう。

また、本章から、プログラムファイルを複数に分けて構成します。構造体や関数のプロトタイプ宣言だけを含むヘッダファイル (拡張子`.h`)、各構造体の操作の実態を記述するファイル (拡張子`.cc`)、そしてそれらを使う `main` を含むファイルです。また、複数のファイルのコンパイル用の `Imakefile` も書くことにしましょう。

12.2　スタックを構造体で表す

スタックはリストと同様にデータが一列に並んでいます。そこで、前章の例 (`list2.cc`) の場合と同じように `Node` という構造体を繋いでいけば良いでしょう。ただし、`List` 構造体とは異なり、

データ並びの先頭だけを知っていれば良いことになります。

まず、ノードとスタックを構成する構造体の定義と関係する関数のプロトタイプ宣言だけを含むヘッダファイルを作成します。スタックを実装するプログラムと、スタックを利用するプログラムで、このヘッダファイルをインクルードします。

Program 12.2.1 node.h

```
//*** node.h **********
//
// 構造体 (第 11 回講義資料)
// 構造体を使ったノードのヘッダファイル
// 作成日:2000/6/19
//
//*************************
struct node {           //一つのデータを保持するノード
  char a;
  struct node* next;
};

typedef struct node Node;

Node *createnode(char c);
```

Program 12.2.2 node.cc

```
//*** node.cc **********
//
// 構造体 (第 11 回講義資料)
// 構造体を使ったノードの実装部
// 作成日:2000/6/19
//
//*************************
#include <stdlib.h>
#include <iostream.h>
#include "node.h"
//*************************************************
Node *createnode(char c){
  Node *n=new Node;
  n->a=c;
  n->next=NULL;
  return n;
}
```

データを保持するノードの定義、及び操作関数 createnode は前章のプログラムと同様です。

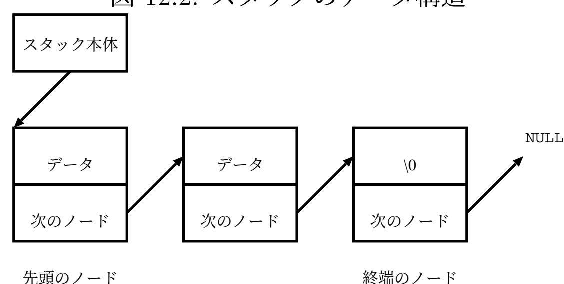
後述するように、データの読みだし時 (pop) に、一番上のデータ要素を取り出して、内部に保存されているデータを返す必要があります。今回のプログラムでは、文字をデータとして保存するので、今回は、スタックの一番下のノードにはデータ\0 を保存することにします。

このようにスタックの場合は、スタックの底の記号を定義するのが一般的です。

Program 12.2.3 stack.h

```
//*** stack.h **********
//
// 構造体 (第 11 回講義資料)
// 構造体を使ったスタックのヘッダファイル
// 作成日:2000/6/19
//
//*************************
#include "node.h"

typedef struct {        //スタック
  Node* root;           //スタックの先頭のノード
} Stack;

// プロトタイプ宣言
void init_stack(Stack &s);
void push(Stack &s,char data);
char pop(Stack &s);
```

図 12.2: スタックのデータ構造



12.3　スタックの操作

次にスタックの操作を定義しましょう。スタックの操作は、新しいデータ要素を先頭に追加する push と、一番上のデータ要素を取り出し、そこに保存されたデータを返す pop の二つです。

まず、新しいデータの追加 push を考えましょう。新しいデータの追加のためには、データを保持する新しいノードを new を使って作成し、データを新しいノードに格納します。次に、新しいノードから古い先頭ノードへポイントし、スタック本体から新しいノードへポイントし、スタックの新しい先頭ノードとします。

先頭ノードの取り出し pop では、先頭ノードのデータを読み出します。このとき、先頭ノードを削除することに注意して下さい。これに伴って、次のノードをスタックの先頭ノードとします。

これまで定義したスタックは、文字をデータとして保持する一般的なものです。これを使って、コマンドラインオプションとして指定された文字列を反転するプログラムを作成しましょう。

main 関数は、スタックを表す構造体を知らなければなりません。そこで構造体とその操作をする関数のプロトタイプ宣言を含む stack.h をインクルードします。

コマンドオプションとして与えられる文字列は、main の引数 argv[1] として main 関数に渡されます。そこで、argv[1] の先頭から終端\0 まで、スタックに push します。入力した文字列の先頭はスタックの底に、文字列の終端はスタックの上に保存されます。

このスタックを上から pop していけば、文字列は反転されます。スタックの底には\0 が書かれていますから、そこまで pop を繰り返します。

今回のプログラムは、スタックの構造体の定義及びスタック操作の関数のプロトタイプを宣言するヘッダファイル stack.h、スタックの操作の実装を記述するプログラムファイル stack.cc、及びスタックを使って文字列を反転するメインプログラムファイル stack-main.cc から構成されています。このように複数のプログラムファイルからなるプログラムをコンパイルするには幾つかの方法があります。

最も単純な方法は

```
g++ -o reverse stack.cc stack-main.cc node.cc
```

とソースファイル名を列挙してコンパイルする方法です。-o revserse は、実行形式ファイルの名前を reverse に指定します。しかし、この方法では、毎回全てのソースファイルをコンパイルすることになり、プログラムが大きくなると非常に非効率的です。ソースファイルごとにコンパイルすることでこの非効率は避けることが出来ます。

```
g++ -c stack.cc
g++ -c stack-main.cc
g++ -c node.cc
g++ -o reverse stack.o stack-main.o node.o
```

オプション-c はコンパイルのみを実行し.o ファイルを生成することを指定します。しかし、この方法でも、ソースファイル数が多くなると非常に大変です。また、コンパイルオプションがあったり、ライブラリ指定があれば更に作業が困難になります。

そこで、プログラム 12.3 のような Imakefile を使うと便利です。Imakefile が出来たら

```
xmkmf -a
```

を実行することで、Makefile を作成します。その後に

```
make
```

を実行すると、コンパイルが行われます。

12.4　モジュール化

本章のスタックのプログラムでは、ノードを表すプログラム (node.cc)、スタックを表すプログラム (stack.cc) 及びスタックを使うプログラム (stack-main.cc) に分けてプログラミングを行いました。

このようにプログラムを独立性の高い部品 (モジュール) に分けてプログラムすることは、プログラム開発の効率を上げ、更に各モジュールを再利用可能とします。

モジュール化されたプログラムをコンパイルするのを支援する道具が Imakefile です。

Program 12.3.1 stack.cc

```
//*** stack.cc **********
//
// 構造体 (第 11 回講義資料)
// 構造体を使ったスタックの実装部
// 作成日:2000/6/19
//
//*************************
#include <stdlib.h>
#include <iostream.h>
#include "stack.h"
//*************************************************
void init_stack(Stack &s)//スタックの初期化
{
  s.root= createnode('\0');// スタックの底の記号
}

void push(Stack &s,char data)
{
  Node* newroot = createnode(data);  //新しいノードの生成
  newroot->next=s.root;      //新しいノードをスタックの先頭に
  s.root=newroot;
#ifdef DEBUG
  cerr <<data<<" is push to "<<newroot<<"\n";
#endif
}

char pop(Stack &s)
{
  char c=s.root->a;            //スタック先頭の文字
  Node* newroot=s.root->next;
  delete s.root;               //スタックの先頭のノードの消去
#ifdef DEBUG
  cerr <<c<<" is pop from "<<s.root<<"\n";
#endif
  s.root=newroot;              //次のノードをスタックの先頭へ
  return c;
}
```

Program 12.3.2 stack-main.cc

```
//*** stack-main.cc **********
//
// 構造体 (第 11 回講義資料)
// 構造体を使ったスタックのメイン
// 作成日:2000/6/19
//
//*************************
#include <stdlib.h>
#include <iostream.h>
#include "stack.h"
//*************************************************
int main(int argc,char** argv)
{
  if(argc!=2){//使い方の説明
    cerr<<"使い方 : "<<argv[0]<<" 文字列\n";
    return 0;
  }

  char* s=argv[1];
  Stack stack;
  init_stack(stack);
  while(*s!='\0'){push(stack,*s);s++;}   //文字列をスタックへ保存
  cout <<"\n";

  char c;
  while((c=pop(stack))!='\0')cout  <<c;  //文字列をスタックから読み出す
  cout <<"\n";
}
```

Program 12.3.3 Imakefile

```
/* Imakefile for

作成者：只木進一
作成日：1999/6/27


xmkmf -f で新しい Makefile を作成し
make でコンパイル実行
*/
SRCS=stack-main.cc stack.cc node.cc  /* ソースファイル一覧 */

CXXEXTRA_DEFINES=   /* コンパイラに渡す define */
CXXEXTRA_INCLUDES=  /* コンパイラに渡す include path */

/* 作成するターゲットの一覧*/
AllTarget(reverse)
/* ソースファイルからオブジェクトファイルへのコンパイル方法の指定 */
NormalCplusplusObjectRule()
/* 実行形式ターゲットのコンパイル */
NormalCplusplusProgramTarget(reverse,stack.o  stack-main.o node.o,\
NullParameter,NullParameter,NullParameter)
/* ファイルの依存関係の確認 */
DependTarget()
```

第13章　構造体(その3)

13.1　目的

構造体を使ったまとめのプログラムとして、英文テキストを読んで、出現単語の一覧を、それぞれの出現回数を含めて調べるプログラムを作成しましょう。

まず、どのような作業が必要かを考えましょう。第一に、英文テキストを標準入力から読み込んで、それを単語に切り出す必要があります。第 10 章のプログラム 10.3.1 を参考にします。実際の英文は、スペース以外に、ピリオドやカンマなどの区切り記号がありますので、その点に注意して、単語を区切ります。

第二に、切り出した単語を保存する方法を考えましょう。デタラメに現れるデータをある決まった順序に並べる方法として、二分木を使う方法が知られています。クイックソート (quick sort) と呼ばれる方法です。今回はその方法を使って、単語を出現時に二分木に保存していきましょう。

二分木を使ったクイックソートは、非常に一般的な方法です。そこで、その部分を独立したモジュールとして構築します。そのため、プログラムは、複数のファイルから構成されます。そこで、分割コンパイルが可能なように構築します。

13.2　二分木

節 (node) を辺 (edge) で連結したものをグラフ (graph) と呼びます。グラフのなかで、次のような性質を有するものを特に木 (tree) と呼びます。

- 根 (root) と呼ばれる唯一の節を持つ。
- 根から根以外の節への道筋 (path) は唯一である。根に近い方向を上、反対側を下とよぶ。
- 根は親 (parent) と呼ばれる上の節を持たない。
- 根以外の節は、上に唯一の親を持つ。
- 節の下の節を子 (child) と呼ぶ。
- 子を持たない節を葉 (leaf) と呼ぶ。

木では、節を左から右に順序付けることができます。木のうち、葉の数が 2 以下の特別な木を二分木 (binary tree) と呼びます。

二分木の各節に、整数を保存することを考えましょう。葉以外の節は、左右に子を持ちますが、

$$\text{左の節の保持する値} < \text{中央の節の保持する値} < \text{右の節の保持する値}$$

となるように、値を保持することにしましょう (図 13.1)。結果の例を図 13.2 に示します。この二分木を左の深い節から読み出せば、整数を小さい順に並べることができます。

図 13.1: 節への値の保存 (再帰的操作)。ある節 n に値 a を保存する場合。

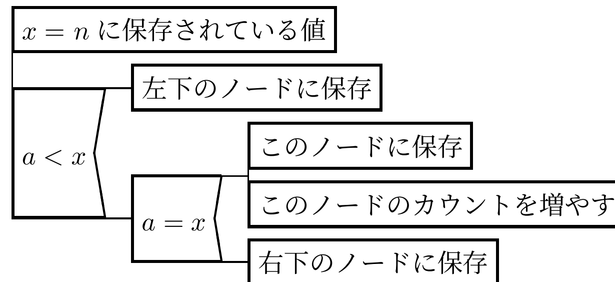
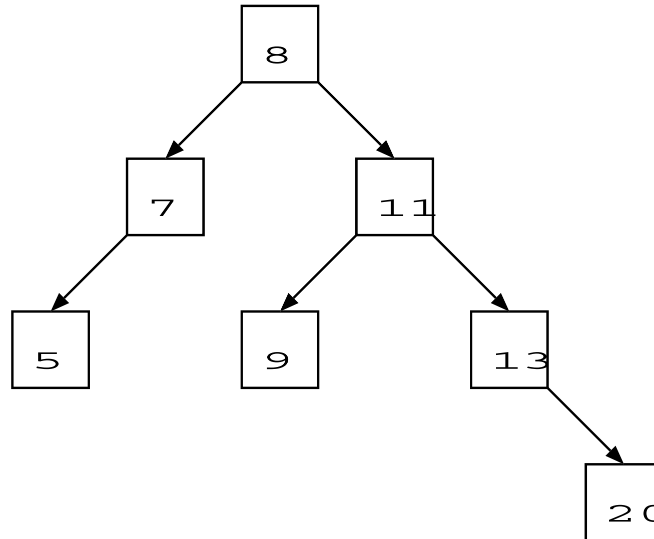


図 13.2: 二分木に保存された整数の例



13.3　二分木に単語を保存する

二分木の各節に、単語を保存することにしましょう。各節を表す構造体 `Node` には、保存する文字列 `str`、その文字列の出現回数を表す `num`、そして左下及び右下の節へのポインタがメンバとして定義されます。

二分木全体を表す構造体 `Btree` は、二分木の根に相当する節へのポインタだけを保持しています。

外部から操作する必要のある関数は、二分木の初期化 `init_btree`、二分木への単語の追加 `add_data`、二分木から単語の読みだし `read_tree`、及び二分木の削除 `delete_tree` の四種類です。

Program 13.3.1 b-tree.h

```
//*** b-tree.h **********
//
// 構造体を使った二分木
// 作成日:2001/7/4
//
//*************************
struct node {           //一つのデータを保持するノード
  char* str;
  int num;
  struct node* left;
  struct node* right;
};

typedef struct node Node;

typedef struct {        //二分木
  Node* root;           //二分木の root のノード
} BTree;

// プロトタイプ宣言
void init_btree(BTree &s);
void add_data(BTree &t,char*);
void read_tree(BTree s);
void delete_tree(BTree &s);
```

節及び二分木の構造体の定義及び上記四種類の関数のプロトタイプ宣言を含むヘッダファイルを b-tree.h としておきます。

関数の実装部分をプログラム 13.3.2 に示します。先頭では、内部で使用する関数がプロトタイプ宣言されています。

二分木の初期化は、関数 init_btree で行われます。この段階では、根の節には何も保存されません。

節への値の保存は、関数 add_data を通じて行います。関数 add_data では、二分木に始めてデータを保存する場合には、根の節にデータを保存します。それ以外の場合には、関数 add_data_sub に根の節を引数として渡します。

関数 add_data_sub は、節と保存する値を引数とする再帰的関数です。保存する文字列が、その節に保存されている文字列よりも小さい (辞書ならびで先に出て来る) 場合には、左下の節に保存しようとします。左下の節が無ければ (ポインタが NULL) ならば、関数 create_new_node で新たに節を生成します。そうでなければ、左下の節を引数にして関数 add_data_sub を呼びます。保存する文字列が、その節に保存されている文字列よりも大きい場合には、右下の節に保存しようとします。保存する文字列が、その節に保存されている文字列と同じ場合には、その文字列の出現回数カウンタを一つ増やします。

Program 13.3.2 b-tree.cc

```
//*** b-tree.cc **********
//
// 構造体を使った二分木
// 作成日:2001/7/4
//
//************************
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include "b-tree.h"

// プロトタイプ宣言
void add_data_sub(Node* node,char*);
Node* create_new_node(char*);
void read_tree_sub(Node* node);
Node* delete_tree_sub(Node* node);


//*************************************************
void init_btree(BTree &t)//二分木の初期化
{
  t.root=NULL;
}

void add_data(BTree &t,char* s)//二分木へデータを追加
{
  if(t.root==NULL){
    t.root=create_new_node(s);
    return;
  }
  add_data_sub(t.root,s);
}

void add_data_sub(Node* node,char* s)
//二分木へデータを追加する再帰手続き
{
  if(strcmp(s,node->str)<0){
    if(node->left==NULL)node->left=create_new_node(s);
    else                add_data_sub(node->left,s);
  } else {
    if(strcmp(s,node->str)==0){
      node->num++;}
    else {
      if(node->right==NULL)node->right=create_new_node(s);
      else                 add_data_sub(node->right,s);
    }
  }
}
```

Program 13.3.3 b-tree.cc(その 2)

```
Node* create_new_node(char* s)//新しいノードを作る
{
  Node* newnode = new Node;
  newnode->str=strdup(s);
  newnode->num=1;
  newnode->left=NULL;
  newnode->right=NULL;
  return newnode;
}

void read_tree(BTree t)//二分木を読む
{
  read_tree_sub(t.root);
}

void read_tree_sub(Node* node)//二分木を読む再帰手続き
{
  if(node!=NULL){
    if(node->left!=NULL)read_tree_sub(node->left);
    cout <<node->str<<" "<<node->num<<"\n";
    if(node->right!=NULL)read_tree_sub(node->right);
  }
}

void delete_tree(BTree &t){
  t.root=delete_tree_sub(t.root);
}

Node* delete_tree_sub(Node* node)
{
  if(node==NULL)return NULL;
  node->left=delete_tree_sub(node->left);
  node->right=delete_tree_sub(node->right);
  delete node->str;
  delete node;
  return NULL;
}
```

節を新たに作成する関数 create_new_node では少し注意が必要です。保存しようとする文字列はポインタで渡されます。つまり、実体は、二分木を呼び出した関数の中で定義されています。従って、二分木を呼び出した関数の中で変更されてしまうかもしれません。実際、今考えているプログラムでは、次々の行が読まれていきますから、ポインタで指していた文字列は変化します。そこで、文字列を保存する際には、一旦 strdup でコピーを作成し、そのコピーの方を保存することにします。このコピーは、この節でだけ使われます。

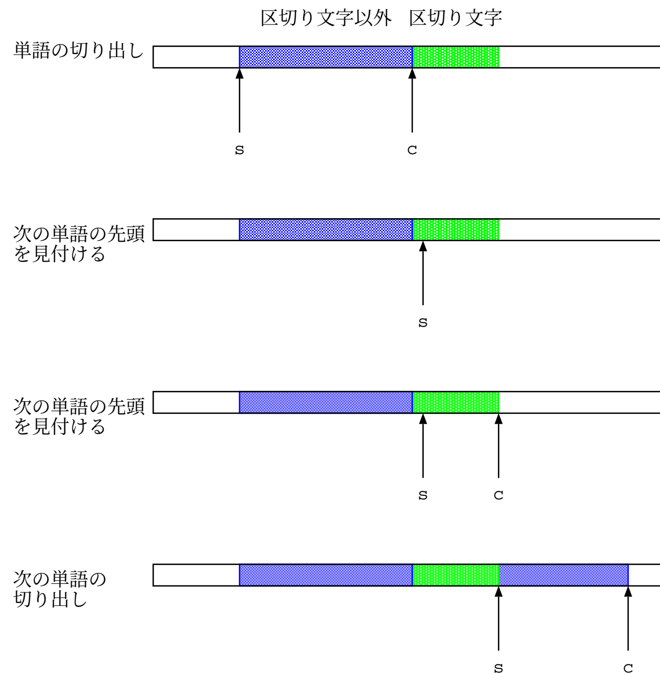
二分木に保存された文字列をその出現回数とともに出力するのが、関数 read_tree 及びその副関数 read_tree_sub です。左側の節、その節そのもの、右側の節という順番で再帰的に出力することで、左側の最も深い所に保持された値から右側の最も深い所に値へ、順番に出力が行われます。

二分木を削除する関数 delete_tree とその副関数 delete_tree_sub も忘れずに作成しましょ

う。左右の節を再帰的に削除した後、その節が保持している文字列を削除し、自分自身を削除します。

13.4　単語切り出し

図 13.3: 文字列の切り出し



次に、単語切り出しの部分について考えましょう。テキストは、標準入力からリダイレクトなどの方法で読み込まれるとしましょう。`cin.getline` を使って一行ずつ読み込みます。

一行読み込んだのち、行の先頭にある区切り文字を読み飛ばします。区切り文字の判定は関数 `isdelimitter` で行います。この段階で、文字列の終端`\0` に到達した場合には、次の行を読むようにします。その行の処理が終ったことの判定は、文字列へのポインタが `NULL` になったことで行うとします。

最初に区切り文字以外が現れた位置へポインタ `s` を指し、最初に区切り文字以外が現れる場所を探し、ポインタ `c` を置きます。ポインタ `c` が指している文字が`\0` であれば、行末に到達したことになりますから、ポインタ `s` が指している文字列を二分木へ保存します。ポインタ `c` が指している文字が`\0` 以外であれば、後ろに文字列が続く可能性があるので、ポインタ `c` が指している場所に`\0` を一旦置き、ポインタ `s` が指している文字列を二分木に保存し、ポインタ `c` が指している次の文字にポインタ `s` を指し、次の文字列操作に移ります。

ファイルを終端まで読み終えたら、二分木に保存した文字列を印刷し、二分木を削除して終了し

ます。

13.5 分割コンパイル

最後に、コンパイルのための Imakefile を示します。ソースファイルは main.cc と b-tree.cc から構成されています。これらをマクロ SRCS に設定します。これらがコンパイルされたオブジェクトを OBJS に設定します。

Imakefile ができたら

```
xmkmf -a
```

を実行し、Makefile を生成します。更に

```
make
```

を実行することで、実行形式ファイル words を生成します。

Program 13.4.1 main.cc

```
//**** main.cc ***********************
//
// 標準入力からテキストを読み、単語の出現回数を数える
// 作成:2001/7/4
//***********************************
#include <iostream.h>
#include "b-tree.h"

int isdelimitter(char c);//文字 c は区切り文字か否か

int main(int argc, char** argv)
{
  const int bufsize=1024;
  char buf[bufsize];
  BTree tree;
  init_btree(tree);

  while(cin.getline(buf,bufsize)){// 一行ずつファイル終端まで読み込む
    char* s=buf;
    while(s!=NULL){
      char* c=s;
      //行の先頭の区切り文字を読み飛ばす
      while(*c!='\0' && isdelimitter(*c))c++;
      if(*c=='\0')s=NULL;
      else s=c;

      if(s!=NULL){
        while(!isdelimitter(*c))c++;// 最初の区切り文字を見付ける
        if(*c=='\0'){//行の終端
          add_data(tree,s);// 単語登録
          s=NULL;
        } else {//それ以外
          *c='\0';
          add_data(tree,s);// 単語登録
          s=c+1;
        }
      }
    }
  }
  read_tree(tree);
  delete_tree(tree);
  exit(0);
}
```

Program 13.4.2 main.cc(その 2)

```
int isdelimitter(char c){ //文字 c は区切り文字か否か
  const int nd=16;
  //区切り文字の定義
  char delimitters[nd]={' ','\t','\n','\r','\0','.',',',':',';','-',
                  '(',')','{','}','[',']'};
  int id=0;
  for(register int i=0;i<nd;i++)if(c==delimitters[i])id=1;
  return id;
}
```

Program 13.5.1 Imakefile

```
/* Imakefile for

作成者：只木進一
作成日：1999/6/27


xmkmf -f で新しい Makefile を作成し
make でコンパイル実行
*/
SRCS=main.cc b-tree.cc  /* ソースファイル一覧 */
OBJS=main.o b-tree.o
CXXEXTRA_DEFINES=   /* コンパイラに渡す define */
CXXEXTRA_INCLUDES=  /* コンパイラに渡す include path */

/* 作成するターゲットの一覧*/
AllTarget(words)
/* ソースファイルからオブジェクトファイルへのコンパイル方法の指定 */
NormalCplusplusObjectRule()
/* 実行形式ターゲットのコンパイル */
NormalCplusplusProgramTarget(words,${OBJS},NullParameter,NullParameter,NullParameter)
/* ファイルの依存関係の確認 */
DependTarget()
```