C言語プログラミング入門

Nishio, Quick

http://karetta.jp/book-cover/c-for-beginners

http://www.usamimi.info/ guiprog/

はじめに

C 言語は，1972 年に AT&T ベル研究所の Dennis M. Ritchie と Brian Wilson Kernighan によって開発されたプログラミング言語です．この言語は，もともとオペレーティングシステムのプログラムを書くために作られたので，実行速度はプログラミング言語の中でも 1,2 を争う程の早さです．また，言語使用が小さいため，小さな資源でも動作させることができます．このような理由から，80 年代，90 年代には，多くのプログラムが C 言語で作られました．

90 年代後半に入ると，コンピュータが各家庭に普及し，処理性能が大幅に増加しました．また，Java，Visual Basic 等のプログラミング言語が登場しました．これらの言語は，90 年代には処理速度が遅い等の理由でほとんど利用されていませんでしたが，2000 年代に入り，コンピュータの処理性能が急速に向上したことで，爆発的に普及していきました．Java 言語の場合，どのような環境でも動く，標準機能が豊富，GUI（グラフィカルユーザインターフェース）の作成が容易等の特徴があります．また，メモリの管理などの低レベルな部分は言語が自動的に処理してくれます．

標準機能が少なく，また利用者がコンピュータの仕組みを十分に理解していないとプログラムを作成するのが困難であった C 言語は，上記の便利で使いやすい言語に取って代わられ，使われることがだんだん少なくなってきました．

ではなぜ，今 C 言語を学ぶのでしょうか．それは，実は C 言語は JAVA や，その他 C++や C#言語等の元になった言語なのです．よって，これらの言語は C 言語に非常によく似ています．また，言語仕様が小さいので，覚えることが非常に少なく，初めてプログラミングを学ぶには適した言語なのです．もし，C 言語をきちんとマスターできれば，他の言語に移行した場合でもすんなり理解することができるでしょう．

本書について

本書は，C 言語を始めて学ぶ方を対象とした書籍です．読者の方のほとんどは，プログラミング言語自体を始めて学ぶ方だと思います．そういった方でも理解できるように書いたつもりです．

目 次

第 1 章　C 言語プログラミングの基礎 1

1.1 コンパイラ 1
1.2 開発環境を整える 2
1.2.1 Visual C++ 2008 Express Edition のダウンロード 2
1.2.2 プロジェクトの作成 3
1.3 画面に文字を表示する 8
1.3.1 プログラムを作る前に 8
1.3.2 Hello World プログラム 9
1.3.3 printf 関数の書き方について 9
1.3.4 エスケープシークエンス 10
1.4 コメント 11
1.5 例題 12
1.5.1 コメントを含んだプログラム 12
1.5.2 長文を表示するプログラム 13
1.5.3 ダブルクォーテーションを表示するプログラム 13
1.6 演習問題 14

第 2 章　変数について 17

2.1 型の種類 17
2.2 変数の作り方 18
2.3 変数命名の際の注意 20
2.4 変数への値の代入 21
2.5 変数値を表示する 21
2.6 n 進数 24
2.7 変数使用の際の注意事項 26
2.8 const 修飾子 26
2.9 演算 27
2.9.1 商を求める際の注意 28
2.9.2 その他の演算 28
2.9.3 略した式 29
2.9.4 べき乗の計算の注意 29

2.10 キャスト . . . 30
2.11 scanf 関数 . . . 31
2.12 配列 . . . 32
2.12.1 一次元配列 . . . 32
2.12.2 配列への値の代入 . . . 33
2.12.3 配列の初期化 . . . 34
2.12.4 文字配列 . . . 35
2.12.5 キーボードから文字列の入力 . . . 36
2.12.6 scanf 使用時の注意点 . . . 37
2.13 文字コード . . . 38
2.13.1 文字コード表 . . . 38
2.13.2 数字の注意点 . . . 40
2.13.3 日本語について . . . 40
2.13.4 文字化けについて . . . 40
2.14 例題 . . . 40
2.14.1 メートルをフィートへ変換するプログラム . . . 40
2.14.2 空白を読み込む scanf . . . 41
2.15 演習問題 . . . 42

第 3 章 制御文 45

3.1 条件分岐 if else 構文 . . . 45
3.1.1 if else 文の書き方 . . . 45
3.1.2 非 0 の判定式 . . . 47
3.1.3 関係演算子 . . . 48
3.1.4 論理演算子 . . . 49
3.2 条件分岐 switch 構文 . . . 51
3.2.1 break を書かない switch 構文 . . . 53
3.3 繰り返し do while for 構文 . . . 54
3.3.1 do while 構文 . . . 54
3.3.2 while 構文 . . . 55
3.3.3 for 構文 . . . 57
3.3.4 break を使ったループの脱出 . . . 58
3.3.5 continue 文 . . . 59
3.4 ビット演算 . . . 60
3.4.1 ビットごとの演算子 . . . 61
3.4.2 シフト演算子 . . . 63
3.5 例題 . . . 64
3.5.1 文字列をコピーするプログラム . . . 64

3.5.2 最大値を求めるプログラム . . . 65
3.5.3 九九表を作成するプログラム . . . 67
3.5.4 文字コード表を表示するプログラム . . . 68
3.5.5 足し算，引き算を行うプログラム . . . 68
3.5.6 合格者判定プログラム . . . 69
3.5.7 ネットワークアドレスを求めるプログラム . . . 71
3.6 演習問題 . . . 71
3.7 応用演習問題 . . . 73

第 4 章 関数 77
4.1 関数とは . . . 77
4.2 関数を作ってみる . . . 77
4.3 プロトタイプ宣言 . . . 80
4.4 配列を引数として渡す . . . 81
4.5 main 関数 . . . 83
4.6 標準関数 . . . 84
4.6.1 文字列関係の処理 . . . 84
4.6.2 入出力関係の処理 . . . 88
4.6.3 数学関係の処理 . . . 92
4.6.4 その他の関数 . . . 94
4.7 例題 . . . 96
4.7.1 自作 strcpy 関数の作成 . . . 96
4.7.2 自作 strlen 関数の作成 . . . 97
4.7.3 自作 strcat 関数の作成 . . . 98
4.7.4 べき乗を求めるプログラム . . . 99
4.8 演習問題 . . . 100

第 5 章 有効範囲とプリプロセッサ 101
5.1 有効範囲 . . . 101
5.1.1 変数寿命 . . . 101
5.1.2 static . . . 103
5.2 プリプロセッサ . . . 104
5.2.1 #define . . . 104
5.2.2 #undef . . . 105
5.2.3 引数付マクロ . . . 105
5.2.4 #include . . . 106
5.3 演習問題 . . . 107

第6章　ポインタ　109
6.1 ポインタとは . . . 109
6.2 ポインタへの値の代入 . . . 111
6.3 ポインタによる変数値の参照 . . . 113
6.4 NULL ポインタ . . . 116
6.5 引数としてのポインタ . . . 117
6.6 ポインタの型 . . . 120
6.6.1 sizeof . . . 121
6.6.2 ポインタの型の大きさ . . . 122
6.7 配列とポインタ . . . 125
6.7.1 配列とポインタ入門 . . . 125
6.7.2 配列とポインタ . . . 130
6.7.3 引数に配列へのポインタを渡す . . . 131
6.7.4 ポインタ配列 . . . 133
6.8 文字列定数とポインタ . . . 135
6.9 ポインタのポインタ . . . 138
6.10 例題 . . . 140
6.10.1 自作 strlen 関数の作成再び . . . 140
6.10.2 自作 strcat 関数の作成再び . . . 141
6.11 演習問題 . . . 141

第7章　構造体　145
7.1 構造体 . . . 145
7.2 構造体へのポインタ . . . 147
7.3 typedef 指定子 . . . 149
7.4 例題 . . . 150
7.4.1 2点間の距離を求めるプログラム . . . 150

第8章　メモリ管理とファイル入出力　153
8.1 動的メモリ割り当て . . . 153
8.2 ファイル入出力 . . . 156
8.2.1 ストリーム . . . 156
8.2.2 ファイル開閉 . . . 157
8.2.3 ファイル読み込み . . . 158
8.2.4 ファイル書き込み . . . 162
8.2.5 ファイル終端の判定とエラーチェック . . . 164
8.3 標準ストリーム . . . 167
8.4 セキュアプログラミング . . . 168

8.5 例題 169
8.5.1 ファイルサイズを求めるプログラム 169
8.5.2 平均点を求めるプログラム 171
8.6 演習問題 172

第1章　C言語プログラミングの基礎

1.1　コンパイラ

C 言語でプログラムを作成した後，実行するにはコンパイルと呼ばれる作業が必要になります．なぜ，このような作業が必要なのでしょうか．

例えば，日本人である太郎さんとアメリカ人であるジョンが会話をするとしましょう．太郎さんは日本語しか話すことができず，ジョンは英語しか理解できません．太郎さんはジョンに仕事を頼もうとしましたが，日本語で話したのではジョンには伝わりません．そこで登場するのが，通訳さんです．この通訳さんは日本語を英語に翻訳してくれます．これで，太郎さんはジョンに仕事を頼むことができるわけです．

プログラミング言語とコンピュータの関係も同じです．コンピュータは 1 と 0，つまり ON と OFF からなる言語しか理解することができません．この言語の事を機械語といいます．私たちはコンピュータに命令をして、何か処理をしてもらう必要がありますが，私たちは機械語を理解することができません．

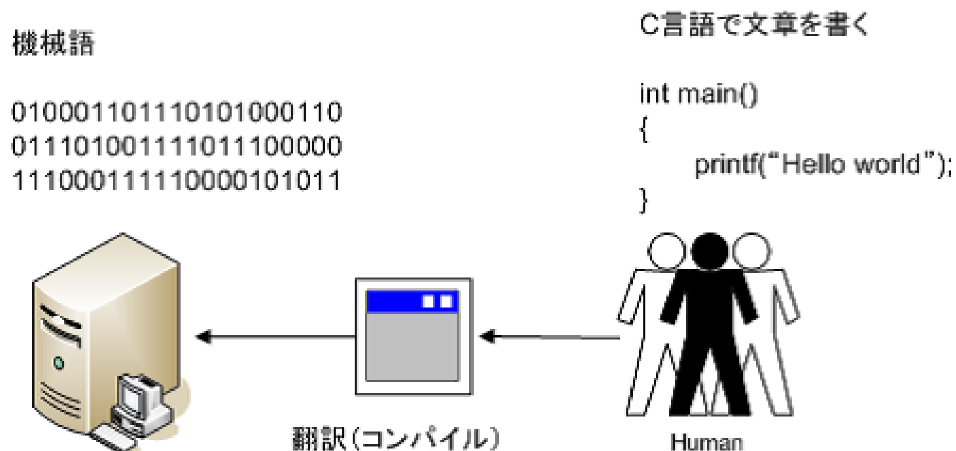

図 1.1: コンパイラの役割

そこで登場するのがコンパイラです．コンパイラは私たちがプログラミング言語で書いたことを機械語に翻訳してくれます．この翻訳者のことをコンパイラと

いい，プログラミング言語の文章を機械語に変換することをコンパイルするといいます．

C 言語でプログラムを書いたのなら，コンパイルを行わなければ実行することができません．

1.2 開発環境を整える

1.2.1 Visual C++ 2008 Express Edition のダウンロード

C 言語で書かれたプログラムを実行するためには，コンパイラが必要です．今回は無償で利用できる Visual C++ 2008 Express Edition を利用しましょう．なお，このコンパイラは Windows 環境でしか実行することができません．Linux 等の他の OS をお使いの方は，gcc 等の別のコンパイラを利用してください．

まずは VC++2008 をダウンロードしましょう．google 等の検索エンジンで Visual C++ 2008 Express Edition と検索をかければ一番先頭にダウンロードページにヒットするはずです（図 1.2)．

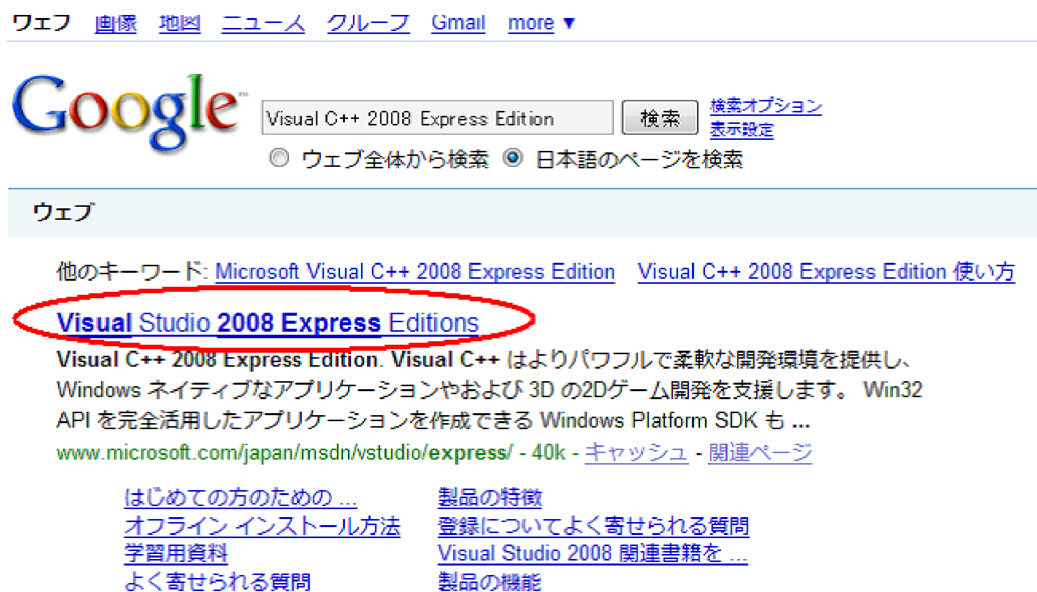

図 1.2: Visual C++ 2008 Express Edition の検索

次に，ダウンロードページから Visual C++ 2008 Express Edition の Web インストールをクリックしてください（図 1.3）．

ダウンロードが完了したら，手順に従ってインストールを行ってください．

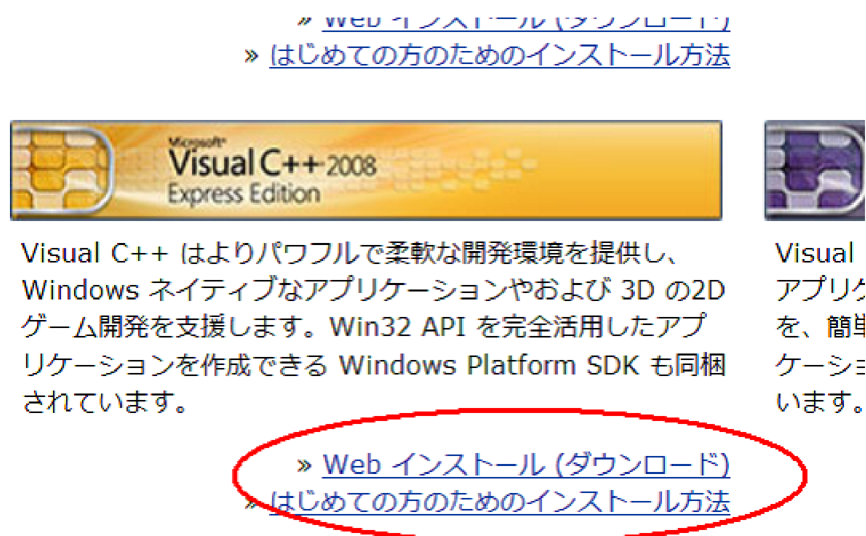

図 1.3: Visual C++ 2008 Express Edition のダウンロード

1.2.2 プロジェクトの作成

インストールが完了したら，プロジェクトを作成してみましょう．VC++では，プログラムをコンパイルするためにプロジェクトと呼ばれるものを作成する必要があります．

まずは Visual C++ 2008 を起動してください（図 1.4）．初回起動時は，多少設定が必要になるかと思います．手順に従って設定を行ってください．

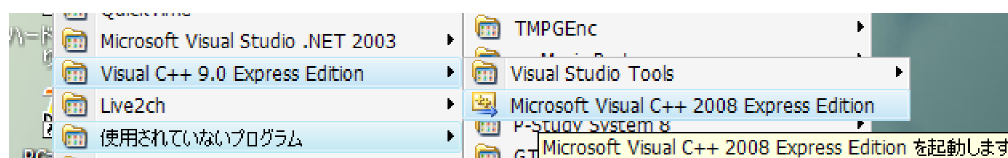

図 1.4: Visual C++ 2008 Express Edition の起動

次にメニューバーからファイル → 新規作成 → プロジェクトを選択します（図 1.5）．

プロジェクトの種類を全般とし，空のプロジェクトを選択してください（図 1.6）．プロジェクト名は好きな名前を付けてください．ここでは testProject としました．また，ファイルの保存場所を指定することもできます．

プロジェクトの作成が完了したら，次はソリューションエクスプローラ内のソースファイルフォルダを右クリックしてください（図 1.7）．そして，追加 → 新しい項目をクリックします．

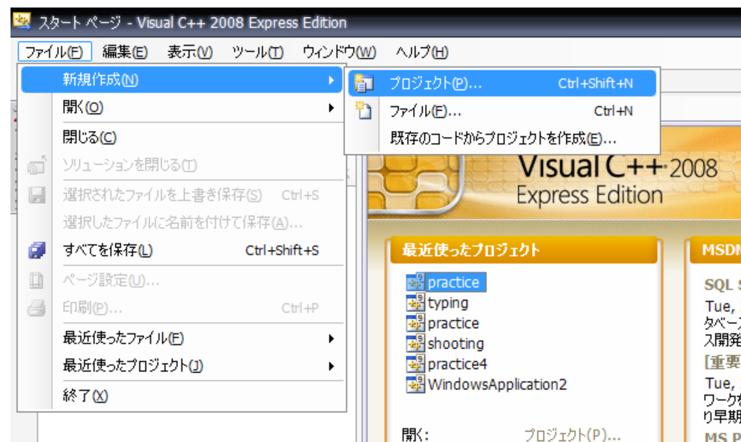

図 1.5: プロジェクトの新規作成

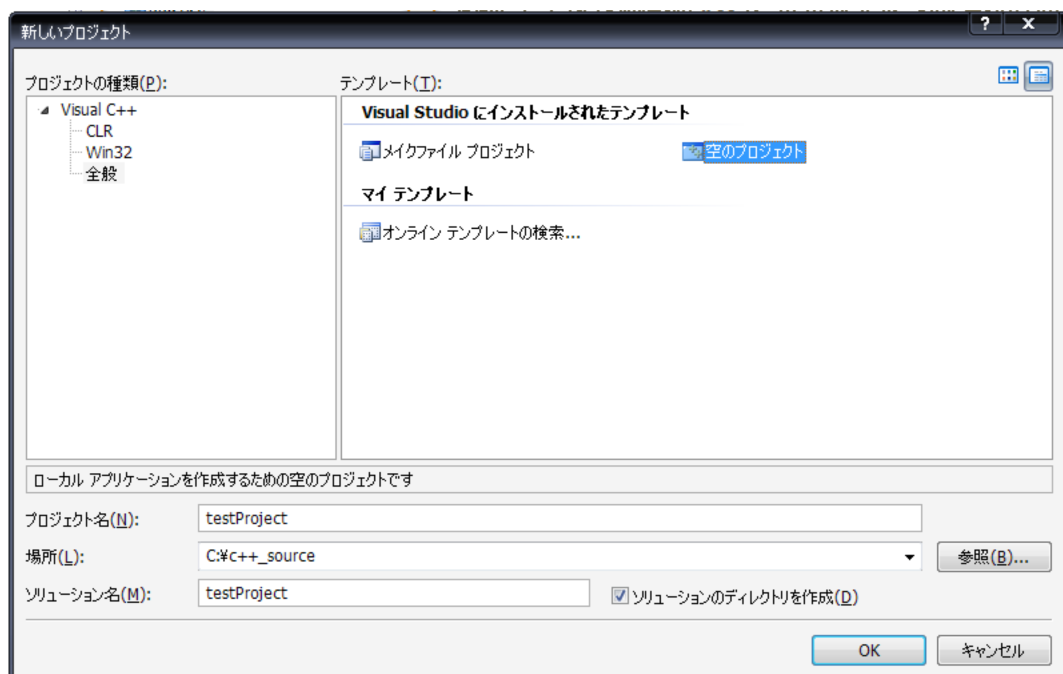

図 1.6: 空のプロジェクトの新規作成

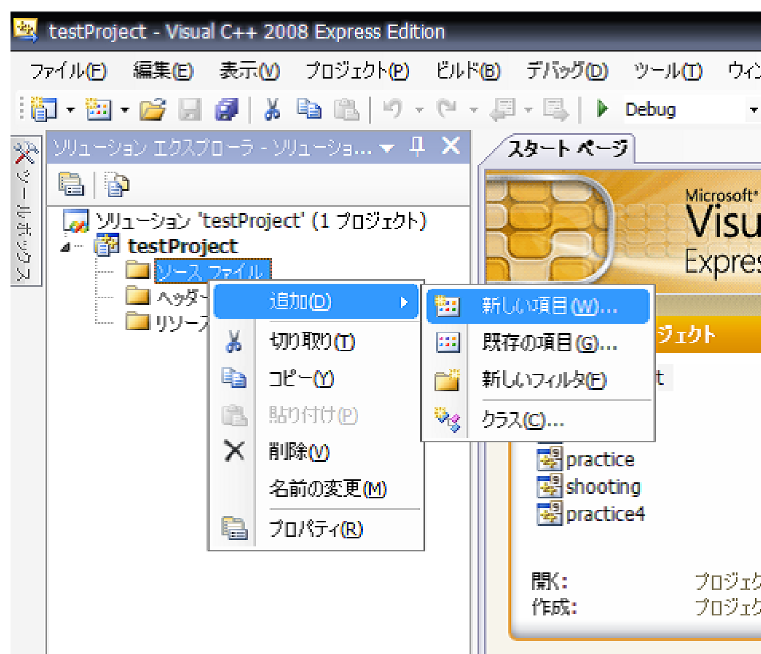

図 1.7: cpp ファイルの追加

テンプレート内の C++ファイルを選択します（図 1.8）．ファイル名は好きな名前を付けてください．ここでは test としました．

作成された test.cpp 内にプログラムを書いていきます（図 1.9）．図のようなプログラムを書き込んでください．

次にプログラムのコンパイル及び実行作業に入ります．メニューバーからデバッグ → デバッグなしで開始 をクリックします（図 1.10）．

無事にコンパイルが成功すれば，図 1.11 の画面が表示されるはずです．

もし，エラーが発生した場合は，原因が出力結果に書いてあります．例えば，6 行目の命令にセミコロンが無かった場合，図 1.12 のように，エラーの原因が出力されます．

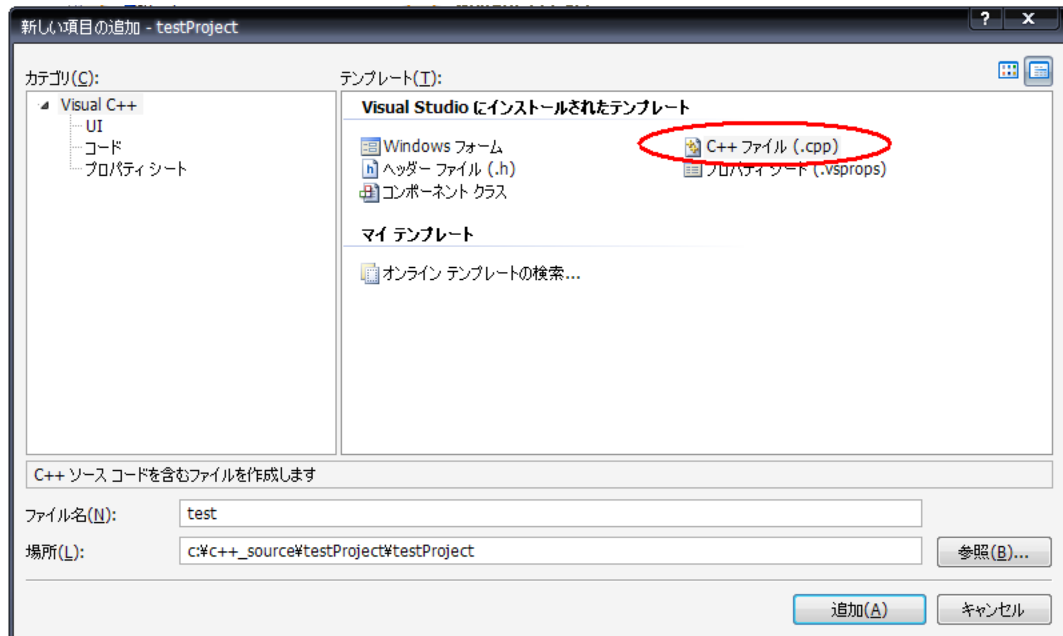

図 1.8: cpp ファイルの追加

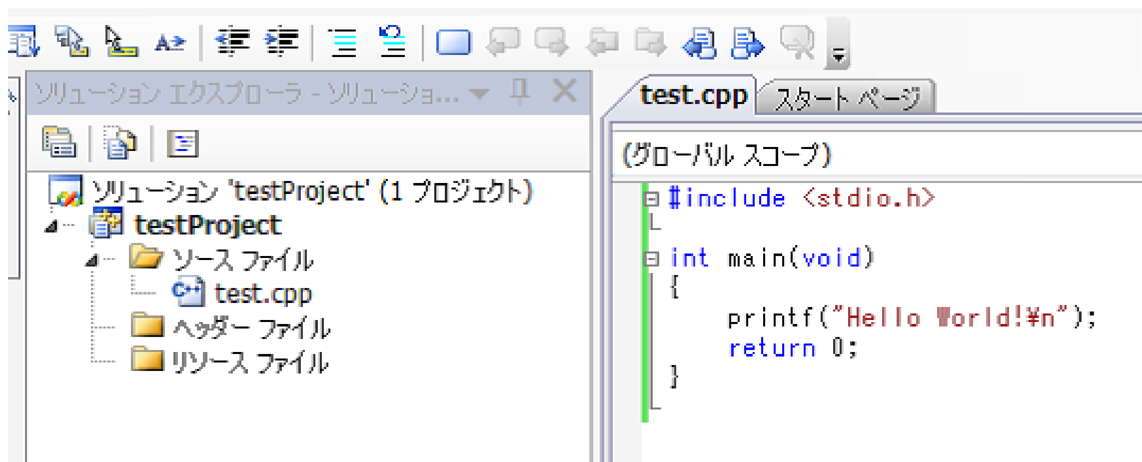

図 1.9: プログラムの入力

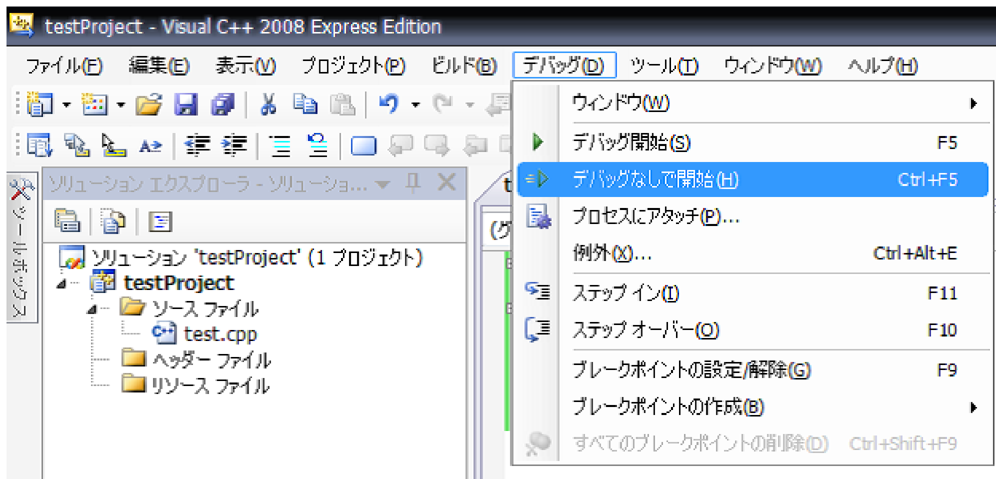

図 1.10: プログラムのコンパイルと実行

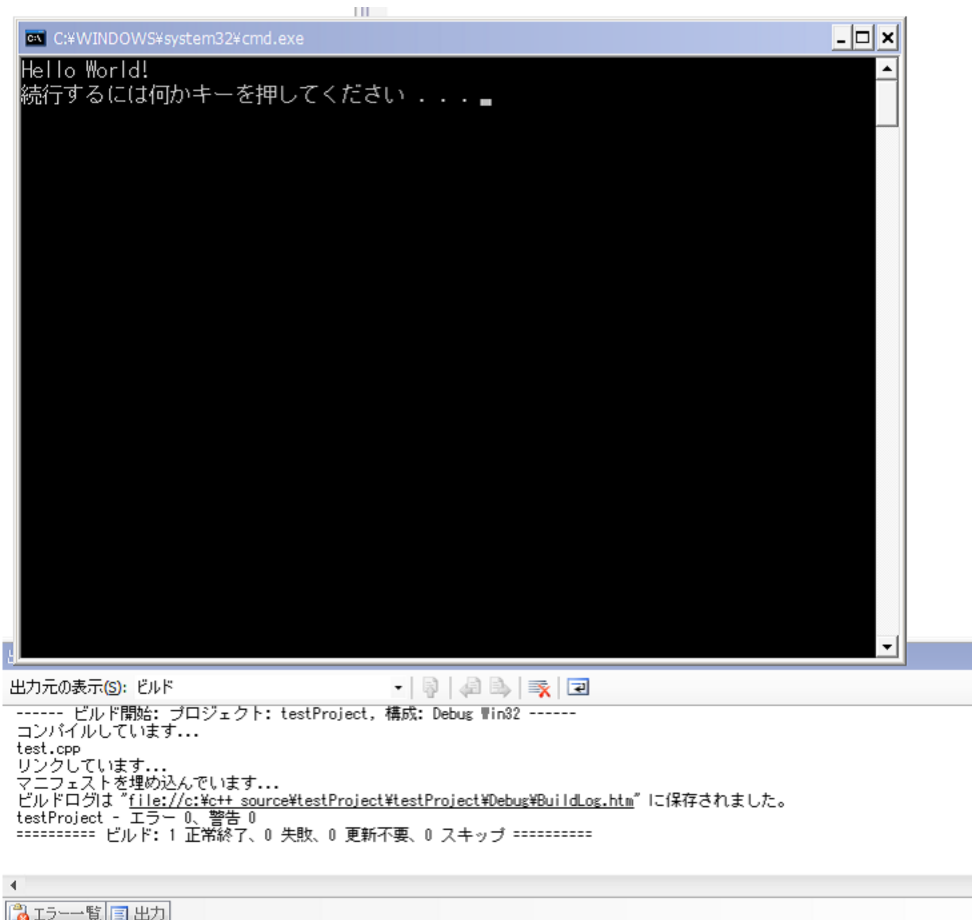

図 1.11: プログラムの実行

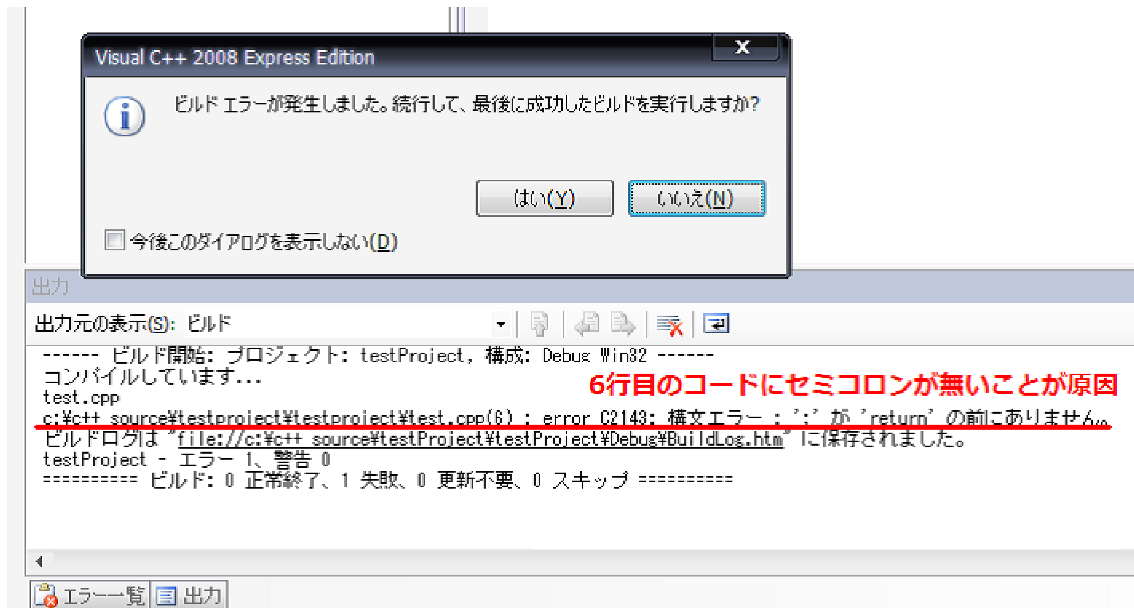

図 1.12: プログラムのエラー

1.3　画面に文字を表示する

1.3.1　プログラムを作る前に

次の文章をまずはエディタに貼り付けてください．ちなみにプログラミング言語で書かれた文章はソースコードと呼ばれています．これからは C 言語の文章の事をソースコード又は単にソースと呼ぶ事にします．

```
1  #include <stdio.h>
2  int main(void)
3  {
4      /* ここにプログラムを書いてね */
5      return 0;
6  }
```

C 言語においていくつか重要な点があります．まず，C 言語で書かれたプログラムには関数と呼ばれるものが必ず一つ以上存在しています．そして，関数の中にはいろいろな命令が詰め込まれています．つまり，関数は命令の塊なのです．

C 言語のプログラムには必ず main と呼ばれる関数が存在しています．この main 関数から C 言語のプログラムは実行されます．つまり，始めの段階では main 関数の中にプログラムを書いていく事になります．

上記の説明は今の段階では，よく分からなくてもよいです．まずは main 関数内の「/* ここにプログラムを書いてね */」というところに実際のプログラムを書いていくということを覚えておいてください．

現段階では上記のプログラムは，おまじないのようなものだと思って丸暗記してください．ただし，`#include <stdio.h>`の stdio.h の部分には注意してください．初心者の方の多くが，stdio をスタジオ (studio) と覚えてしまい，書き間違えてしまうことがあります．stdio は STanDard Input Output（標準入出力）の略であり,「エスティーディーアイオー」とか「スタンダードアイオー」等と読みます．

1.3.2　Hello World プログラム

では画面に文字を表示させてみましょう．

```
printf("Hello World!");
```

というソースコードを，/*ここにプログラムを書いてね*/ という部分に貼り付けてください．

```
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("Hello World!");
5      return 0;
6  }
```

このプログラムをエディタに書いて，コンパイルしてみてください．画面上に「Hello World!」と表示されたら成功です．printf 関数については，次節で説明します．

1.3.3　printf 関数の書き方について

画面上になにか文字を表示したい場合には，printf という関数を使います．printf() のカッコの中に表示したい文章を”(ダブルクォーテーション）で囲んで書きます．

書式

```
printf(" なにか表示したい文字をここに書く ");
```

ここで注意したいのがprintf関数の最後にセミコロン「;」が付いているということです．C言語では一つの命令の最後にセミコロンが付きます[1]．始めのうちはこのセミコロンを忘れがちなので注意してください．

1.3.4 エスケープシークエンス

printf関数を使い，改行を含んだ文字列を表示したい場合があります．しかし，C言語では次のように記述することはできません．

```
printf("Hello
World");
```

改行は\nという特殊な文字を使って行います.「\」と「n」2つの文字で改行という意味です．[2]

```
printf("Hello\nWorld!\n");
```

このようにして使います．改行文字のほかに，表1.1にある特殊文字を使うことができます．これらは，エスケープシークエンスと呼ばれています．

表1.1: エスケープシークエンス一覧

記号	意味
\n	改行
\a	BEEP音（警告音）
\0	NULL文字
\t	タブ
\b	バックスペース
\?	疑問符
\'	シングルクォーテーション
\"	ダブルクォーテーション
\\	バックスラッシュ（日本語の端末の場合, 円マーク）

[1]例外もいくつか存在します.
[2]日本語端末の場合バックスラッシュは円マーク¥を使います.

表 1.1 には，警告音を鳴らす為の文字 (\a) や，タブ[3]を表す文字 (\t) 等があります．NULL 文字[4]というのは文章の最後に付く文字のことで，文字列の終わりを表すものです．例えば,

```
printf("Hello world!");
```

は

```
printf("Hello world!\0");
```

と同じです．前者のように文字列の最後に NULL 文字を書かなくても，C 言語は後者のように文章の最後に勝手に NULL 文字を付加します.

表 1.1 のエスケープシークエンスは覚えにくいものです．\n だとか\b 等の略語の語源を知っていると，少しは覚えやすくなるかもしれません.

- \n　New line(改行)
- \a　Alert(警告音)
- \t　Tab(タブ)
- \b　Back Space(バックスペース)

1.4 コメント

コメントは，ソースコード（自分が C 言語で書いた文章の事）の中で，そのソースが何を意味しているかを書いておきたいときに利用します．たとえば次のようにしてコメントを書くことができます.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("Hello world!"); /* Hello World!と画面上に表示する */
5      return 0;
6  }
```

[3] タブとは，スペースをいくつかあわせたものの事です．多くの場合，スペース 4 文字分です.
[4] ヌル文字，又はナル文字と読みます.

この「/* Hello World!と画面上に表示する */」という部分がコメントです．/* からはじまって　*/で閉じたところまでがコメントとなります．このコメントの書き方は，ANSI C と呼ばれる C 言語の規格のものです．

複数行にまたがってコメントを書くこともできます．例えば，

```
/*
この
文章を
画面上に
表示する
*/
```

といったように使います．

上記のコメントの書き方以外に,「//」というコメントも存在しています.「//」以降，その行末までがコメントとなります．この書き方は，C99 と呼ばれる C 言語の新しい（ANSI C よりも後に登場した）規格でのコメントです．多くのコンパイラが C99 をサポートしていますが，一部のコンパイラでは**正常にコンパイルできない**可能性もあります．よって，本書では ANSI C 標準のコメントを採用することとします．

プログラムを作成したら，コメントはできるだけ書き込むようにしてください．あとでソースコードを見直した際に，プログラムの説明がかかれていない場合，何を処理しているのか分からなくなってしまいます．

1.5 例題

1.5.1 コメントを含んだプログラム

コメントが含まれているプログラムの例を見ていきましょう．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      /* 改行文字が入っていないので次の行と同じラインに表示される */
6      printf("This is a test program ");
7      printf("without indented. ");
8      /*
9      この部分はコメントアウトされているので表示されない
10     print("This line is not displayed. ");
```

```
11        */
12        return 0;
13    }
```

このプログラムで注意したい点は「This line is not displayed.」という部分が実行結果には表示されていないことです．コメントアウトされている部分はたとえ命令であっても無視されます．

実行結果

This is a test program without indented.

1.5.2 長文を表示するプログラム

printf 関数で表示する文字数が，1 行あたりで多い場合，コードが読みにくくなってしまいます．この場合の修正方法を学んでいきましょう．

printf 関数を一回だけ使い, There are 10 types of people in this world. Those who understand binary and those who don't. Which one are you? という文章を表示するとしましょう．しかし同じライン上で書くとコードが読みにくくなってしまいます．この場合以下のようにコードを書くこともできます．

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        /* このようにコードを書くこともできる */
6        printf("There are 10 types of people in this world."
7          "Those who understand binary and those who don't."
8          "Which one are you?\n" );
9        return 0;
10   }
```

実行結果

There are 10 types of people in this world.Those who understand binary and those who don't.Which one are you?

1.5.3 ダブルクォーテーションを表示するプログラム

次のような文章を表示するプログラムを作りたいとします．

He said, ”How dare you do to me like that!”

そこで，次のようなコードを書いてみましたが，うまくいきませんでした．

```
printf("He said, "How dare you do to me like that!"");  /* error */
```

どのようにすれば”（ダブルクォーテーション）を表示できるのでしょうか．それは，先ほども登場したエスケープシークエンスを使うことで解決できます．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("He said \"How dare you do to me like that!\"");
6      return 0;
7  }
```

このようにできることも覚えておきましょう．

実行結果

He said, ”How dare you do to me like that!”

1.6 演習問題

問題 1

自分の名前を画面上に表示するプログラムを作成せよ．

問題 2

ビープ音を鳴らすプログラムを作成せよ．

問題 3

次のプログラムの実行結果を予測せよ．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello \0 world");
6      return 0;
7  }
```

問題 4

printf() 関数を一度だけ使い次の文章を表示するプログラムを作成せよ．ただしソースコードが読みやすくなるように工夫すること.

The ICMP source quench message is the TCP/IP equivalent of telling another computer: ”I can’t keep up with all the traffic you’re sending me - slow down, please.” [5]

[5] Firewalls FOR DUMMIES by Brain Komar, Ronald Beekelaar, and Joern Wettern, PhD より引用

第2章　変数について

変数とは，情報を保存することができる箱のようなものです．この箱には，何か一つだけ情報を入れることができます．この箱に値を入れる作業は代入と呼ばれます．

箱にはいくつかの種類が存在しています．例えばパソコンを入れるための大きな箱と，お菓子を入れる為の小さな箱があるとします．パソコン専用の箱にはパソコンを入れることができ，同様にお菓子を入れる専用の箱にはお菓子を入れることができます．ここでお菓子を入れる箱にパソコンを入れようとすれば，パソコンが大きいため，残念ながら入れることはできません．逆に，パソコンを入れる箱にお菓子を入れると，お菓子は小さいので，箱には余分なスペースが残ってしまい，スペースの無駄遣いとなってしまいます．

このように，用途によって箱を使い分けますが，C 言語でも同様に用途によって変数（箱）を使い分けます．この箱の種類の事を変数の型といいます．

2.1　型の種類

具体的に変数にはどのような型があるのかをみていきましょう．C 言語では表 2.1 にある変数の型が利用できます．なお，表にかかれている数値は，処理系依存であり，開発環境によって異なる場合があります．

short 型の変数には，-32768 〜 +32767 までの整数値を格納することができます．よって，1000000 等の大きな値や，少数値を代入することはできません．int 型の場合は，表 2.1 より-2147483648 〜 +2147483647 までの整数値を保存することができます．

long 型は昔は int 型よりも大きな範囲を扱うことができる変数でしたが，現在では int 型と変わりないものとなっています．今後は long 型が int 型よりもさらに大きくなる可能性があります．

unsigned とついているものは，符号付の数値を保存することができないものです．例えば，unsigned int では，-1 などの値を表現することができません．

char 型は少し特殊な型で，主に整数値ではなく一文字を保存するために利用されます．

float,double は少数値まで扱うことができるものです．double は float よりも細かい精度を表現することができます．

たいていは，整数を扱う場合は int 型を，少数を扱う場合は double 型を使います．文字を扱う場合の変数は特殊で char 型を利用します．

表 2.1: 変数の型一覧（処理系依存）

型の名前	範囲
short	-32768 〜 +32767
int	-2147483648 〜 +2147483647
long	-2147483648 〜 +2147483647
unsigned short	0 〜 +65535
unsigned int	0 〜 +4294967295
unsigned long	0 〜 +4294967295
char	-128 〜 +127 又は 0 〜 +255
unsigned char	0 〜 +255
float	3.4 * 10^-38 〜 3.4 * 10^+38
double	1.7 * 10^-308 〜 1.7 * 10^+308

C 言語コンパイラの仕様は，結構いい加減なところが多く存在しています．例えば，表 2.1 にある char 型の値の範囲を見てください．「-128 〜 +127 又は 0 〜 +255」となっています．これは，コンパイラ側がどちらを採用しても良い事になっています．また，int 型の値の範囲は「-2147483648 〜 +2147483647」となっていますが，昔は「-32768 〜 +32767」でした．このように，コンパイラの仕様で厳密に定められておらず，利用するコンパイラや OS によって定義が変わってきてしまうものは処理系依存と呼ばれています．

2.2 変数の作り方

では，実際に変数を作ってみましょう．int 型の変数を作りたい場合，次のように宣言します．

```
int hoge;
```

これは，hoge という名前の int 型の箱を作れという意味です．箱には名前をつけることができます．同様に double 型の piyo という名前の箱を作りたい場合は，

```
double piyo;
```

とします．

また，変数を複数同時に宣言する場合は，次のように記述することもできます．

```
double foo, bar;
```

上記の宣言は

```
double foo;
double bar;
```

と全く同じ意味になります．

ここで注意しなければならないことは，**変数はブロックの始めに宣言する**ということです．ブロックとは，{から}の範囲のことをいいます．例えば次のようなコードがあったとします．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int hoge;
6      printf("Hello");
7      printf("WORLD");
8
9      return 0;
10 }
```

このコードは問題ありません．しかし，以下のようなコードは，ANSI C では書くことができません．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      /* このコードはエラーが発生する */
```

```
 6        printf("Hello");
 7        int hoge;
 8        printf("WORLD");
 9
10        return 0;
11    }
```

変数は ブロックの一番初め，つまり { (中カッコ) のすぐ後ろに宣言しなければならないので，上記のようなコードは書くことができません．[1]

2.3 変数命名の際の注意

変数名を付ける際に，次のようなルールがあります.

- 先頭は英字か_(アンダースコア）でなければならない
  数字ではダメです.
- 同じ名前の変数が 2 つ存在する事は基本的にはできない
  例外も存在します．詳しくは有効範囲の章をご覧下さい.
- 予約語は使えない
  予約語 (表 2.2）とは，int や double 等，コンパイラがあらかじめ定義しているキーワードのことです.

表 2.2: 予約語一覧 (一部のみ掲載）

auto	double	int	struct	break	else	long	switch
case	enum	register	typedef	char	extern	return	union
const	float	short	unsigned	continue	for	signed	void
defalut	goto	sizeof	volatile	do	if	static	while

[1] C 言語の新しい規格である C99 では，変数はどこでも宣言できるようになりました.

2.4 変数への値の代入

箱に何か値を入れてみましょう．例えば，10 という値を hoge という変数に入れたい場合，次のように記述します．

```
int hoge;
hoge = 10;
```

この = という記号は，式の右側の値を左側に代入せよ，という意味です．数学でいうイコール（左辺と右辺は等しい）ではないので注意してください．

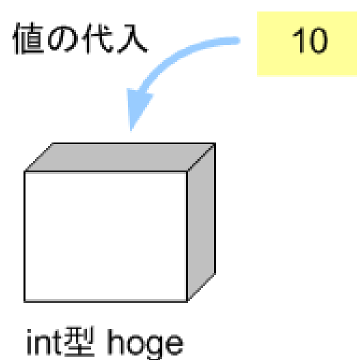

図 2.1: 値の代入

ここで箱を作った時点で値を入れた状態にしたい場合（これを初期化という）は，`int hoge = 10;` とします．初期化と代入は似ていますが少し意味が違います．初期化は最初から値が入った状態にすることであり，代入とはもともと作られている箱に値を格納することです．

余談ですが，数学でいう x=10（x は 10 である）といった本当にイコールの意味を使いたい場合はどうすればよいでしょうか．これは，制御文である if 文という所で解説します．今軽く説明しておくと，==という記号を使います．イコールを二つ繋げて書くと，数学でいうイコールという意味になります．

2.5 変数値を表示する

変数内の値を画面上に出力したい場合，どのようにしたらよいでしょうか．例として，int 型の hoge という箱に 10 という値を代入して，その箱の中身を画面上に表示させてみます．

```
int hoge;
hoge = 10;
printf("%d",hoge);
```

```
int hoge;
hoge = 10;

printf( "%d",  hoge  );
```

%dを10に置き換える

実引数

図 2.2: 変数値の表示方法

%d の部分は，続く実引数を整数の 10 進法で表示せよ，という意味です．実引数とは,「,」の後に続く部分のことをいいます．このプログラムを実行すると，次のように画面上に表示されます．

実行結果

```
10
```

10 進数で表示以外にも，8 進数や 16 進数で表示というものも存在します（表 2.3）．コンピュータサイエンスを勉強したことのない読者の方は，もしかしたら 2 進数は 16 進数にはなじみがないかもしれませんね．でも安心してください．次節で 2 進数や 16 進数について簡単にですが説明をします．

小数値を表示したい場合（double 型や float 型の場合）には，%f を利用します．この%で始まる部分を変換指定子といいます．他にもいろいろ試して見ましょう．

表 2.3: 変換指定子

記号	意味
%d	整数の 10 進数として出力
%u	符号なし整数の 10 進数として出力
%o	整数の 8 進数として出力
%x	整数の 16 進数として出力
%f	小数値表示（double や float の場合に使う）
%c	1 文字出力
%s	文字列を出力
%p	ポインタの値を出力
%%	%を出力

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int hoge = 10;
5       int piyo = 20;
6       double foo = 30.256;
7
8       printf("hoge の値は%d です\n",hoge);
9       printf("piyo の値は%d です\n",piyo);
10      printf("値は%d です\n", 50);
11      printf("hoge は%d piyo は%d です\n",hoge,piyo);
12      printf("foo の値は%f です\n",foo);
13      printf("hoge の値は%f  です\n",hoge);
14      printf("foo の値は%d です\n",foo);
15      return 0;
16  }
```

実行結果

```
hoge の値は 10 です
piyo の値は 20 です
値は 50 です
hoge は 10 piyo は 20 です
foo の値は 30.256000 です
hoge の値は-11188256525479525000000.000000 です
foo の値は 927712936 です
```

13,14 行目の結果が異常な数値になっています．これは，13 行目は値は int 型なのに小数として表示させているからです．また，14 行目は double 型なのに整数で表示させているのでこのような結果となってしまいました．

では, 小数の値を整数として表示させるためにはどうすればよいでしょうか．その方法は，後に登場するキャストを使うことにより解決できます．

表 2.3 の変換指定子はとても覚えにくいものです．%d だとか%p 等の略語の語源を知っていれば，少しは覚えやすくなるかもしれません (英語になじみの無い方は覚えにくいかもしれませんが)．ここでは，一部の略語の語源を紹介します.

- %d　Decimal Number(デシマルナンバー)，10 進数の事
- %o　Octal Number(オクタルナンバー)，8 進数の事
- %x　heXadecimal Number(ヘクサデシマルナンバー)，16 進数の事
- %f　Floating Point Number(フローティングポイントナンバー)，浮動少数の事
- %c　Character(キャラクター)，文字の事
- %s　String(ストリング)，文字列の事
- %p　Pointer(ポインタ)，ポインタの事

2.6 n 進数

ここでは 2 進数と 16 進数の考え方について簡単に説明します.

我々が普段よく使っている数の表し方は 10 進数です．これは，0 から 9 までの 10 個の数字を使って数を表すものです.

2 進数とは，0 と 1 の 2 つの数字だけを用いて全ての数を表すものです．例えば，10 進数で 1 は 2 進数でも 1 です．では，10 進数での 2 は，2 進数でどのように表すのでしょうか．0,1 ときて，次の数は 2 ですか?いえいえ，2 進数では 2 という数字は使えないのですよね．正解は 10 です．このように桁上がりが発生します.

16 進数も同様に 16 個の文字を使って数を表します．16 進数は，0 から 9 までの数字と，A から F までのアルファベットを使って数を表現します.

10 進数と 2 進数，16 進数の関係は表 2.4 となります．この 2 進数と 16 進数は，プログラミングにおいては頻繁に用いられる概念なので，よく理解しておいてください.

10 進数	2 進数	16 進数
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11

表 2.4: 2 進,10 進,16 進数の関係

2.7 変数使用の際の注意事項

変数を初期化，または値を代入しないまま，中身を表示してみることとします．例えば，

```
int hoge;
printf("hogeの値は%dです",hoge);
```

としたら，どのような結果となるでしょうか．

実は出力結果はわかりません．もしかしたら0とでるかもしれませんし，分けのわからない数字が出力されるかもしれません．また Visual C++ 2008 Express Edition においてはこのようなコードはエラーを起こし実行することができません．このように，変数を作った時点では中にはゴミが入っています．そのため，変数は必ず何か値を代入，又は初期化してから使うようにしましょう．

2.8 const 修飾子

変数の型の前に const[2]を付けると，変数の中身を書き換えることができなくなります．ただし，初期値を与えることはできます．const 修飾子は，読み込み専用の変数を宣言したい際に利用します．

例えば，次のように初期値を与え，その値を読み取ることはできます．

```
1  #include <stdio.h>
2  int main(void)
3  {
4      const int i = 5;
5      printf("%d", i);
6      return 0;
7  }
```

しかし，const 指定した変数値に，値を代入することはできません．例えば，以下のようなプログラムを書くことはできません．

```
1  #include <stdio.h>
2  int main(void)
3  {
4      const int i = 5;
5      i = 10; /* コンパイルエラー */
6      return 0;
7  }
```

[2]const は CONSTant(不変の，一定のという意味）の略です．

2.9 演算

数学でもお馴染みの四則演算，つまり「足し算」「引き算」「掛け算」「割り算」をC言語でも行うことができます．その方法は簡単です (表2.5)．

表 2.5: 四則演算

記号	意味
+	足し算
−	引き算
*	掛け算
/	割り算

ここで，簡単な計算の例をお見せします.

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int x, y;
5
6       x = 10;
7       y = 20;
8       printf("x+y は%d です\n", x + y);
9       y = y - x;
10      printf("y の値は%d です\n", y);
11      return 0;
12  }
```

このプログラムの実行結果は次のようになります.

実行結果

```
x+y は 30 です
y の値は 10 です
```

2.9.1 商を求める際の注意

商，つまり割り算の答えを求める際に注意すべき点があります．たとえば次の計算を見てください．

```
int x = 7;
int y = 3;

printf("x/yの値は%dです", x / y);
```

この計算では，答えは 2 と表示され，小数点以下は切り捨てられることになります．なぜなら，int 型は整数しか箱に値を入れることができないからです．小数点以下の値ももとめたい場合は，double 型を用いればよいのです．

```
double x = 7;
double y = 3;

printf("x/yの値は%fです", x / y);
```

また，y の値を 0 にして，0 で何か値を割ってはいけません．その点は数学のルールと同じです．

2.9.2 その他の演算

値の符号の反転をしたい場合は次のようにします．

```
int x = 10;
printf("反転した値は%dです\n", -x);
```

剰余，つまり割り算の余りを求めたい場合は，次のようにします．

```
int x = 10, y = 3;
printf("余りは%dです\n",x % y);
```

この剰余の計算は，**結構利用することが多い**ので，是非覚えておきましょう．

2.9.3 略した式

次のように記述することができます.

```
int x = 10;
x += 10;
```

この x+= 10 というのは，x = x + 10 を略した書き方です．これらの計算はよく使われるので，こうした略が存在します．他の演算も同様に

```
x -= 10;
x /= 10;
```

などと記述することができます.

また，x に 1 を足したい場合があります．この 1 を足すという作業は，特によく使われるので，更に略して

```
x++;
```

という書き方をしても良い事になっています．これを**インクリメント**といいます．同様に，**デクリメント**，つまり値を 1 引くという書き方も存在します.

```
x--;
```

C 言語を拡張した言語に C++言語があります．この言語の名前の由来は，C 言語を++した，つまり C 言語より一歩進んだ言語ということで，このような名前が作られました．ちなみに，C 言語は B 言語を元にして作られました．また，C 言語より後に作られた言語として，D 言語とか C#言語なんてものもあります．C#の名前の由来は C++++, つまり C++言語よりも一歩進んだ言語という意味です．++を縦に 2 つならべてみてください.

2.9.4 べき乗の計算の注意

べき乗とは，x の 2 乗，つまり x*x などのことをいいます．よく，コンピュータでべき乗を表現する際に，x^2 という表現が使われますが，C 言語ではこれでは 2 乗という意味になりません．^は xor（排他的論理和）という意味になってしまい

ますが，今はあまり気にしなくてよいです．２乗の計算をしたい場合は，素直に x*x としてください．ここで，３乗，４乗 ··· と増えていくと，x*x*x*x* ··· となってしまいます．これらを回避する方法として，pow という関数が存在しますが，これも今は気にしないでください．簡単に説明しますと，pow は次のように使います．

```
pow( x , 4.0 );
```

これは x の 4 乗という意味です．

2.10 キャスト

C 言語では，計算を行う際には，必ず型が同じでなければならないというルールが存在します．例えば次のようなコードを書いたとします．

```
double dx = 10.0, dz;
int iy = 3;

dz = dx / iy;
```

このように int と double 型との違う型同士の計算をした場合は，格上げという暗黙の型変換が自動的に行われています．ここでは，iy は型が**一時的に**double 型に変換されます．暗黙の型変換ではより大きな型に合わせて変換されます．たとえば，int と double では，int は整数だけを収納できるのに対し，double は整数も収納できるし，少数も収納することができます．よってこの場合は，int は double に型変換されることになります．

暗黙の型変換ではなく，明示的に型変換をしたい場合次のようにします．

```
double dy = 10.0, dz;
int iy = 3;

dz = (int)dy / iy;
```

このように dy の型を int 型に一時的に変えたい場合，dy の前に (int) と書きます．同様に double 型にしたい場合は，(double) とします．こういった型変換を**キャスト**と呼びます．

2.11 scanf関数

キーボードから何か値を入力して，変数の中に代入したい場合があると思います．そういった場合に使うのが scanf 関数です．次のソースをコンパイルして実行すると，キーボードから読み込んだ文字を画面上に出力します．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x;
6
7      printf("数字を入力してください:");
8      scanf("%d", &x );
9      printf("キーボードから %d という数字を受け取りました\n" , x);
10     return 0;
11 }
```

scanf は printf と書き方が似ていますが，**x の前に&が付いていること**に注意してください．この&の意味はポインタのところで詳しく説明しますので，今は&をつけなければならないという事を忘れないでください．scanf 文の中に書いてある%d は printf の時と同様，10 進法の値をキーボードから読み込んで，その値を x に収納せよということです．ただし，double 型を読み込みたい場合，%f ではなく%lf となることに注意してください．

現段階ではキーボードからの入力を受け取る場合 scanf 関数を使っていますが，この関数は実際にソフトを開発する際には使用しないでください．
では初めから scanf 関数を使わずに説明を進めれば，と思われる方もいるかもしれませんが，scanf を使わずにキーボードから入力を受け取るのはやや難しいので，今の段階では scanf を使うこととします．
なぜ scanf を使用してはいけないのかという理由は今は難しいので，ここでは説明はしません．興味のある方はセキュアプログラミングの節をご覧下さい．C 言語には scanf 関数の他にも gets 関数，sprintf 関数など実際のソフトウェア開発では使わないほうがいい関数がいくつも存在しています．

2.12 配列

変数とは，一つの値を入れることができる箱と説明しました．では，ここで変数を 5 個作って見ましょう．

```
int hoge1;
int hoge2;
int hoge3;
int hoge4;
int hoge5;
```

同じような目的で使う変数を 5 個作ってみました．しかし，ここで 100 個の変数が必要としましょう．そうしたらどうなるでしょうか．

```
int hoge1;
int hoge2;
int hoge3;
    .
    .
    .
```

と宣言だけで大変な事になってしまいます．そういったときに，同じ種類の変数をひとまとまりのデータとして扱うために，配列というものが存在します．つまり，たくさんの変数を一気に作ってしまうものを配列といいます．

2.12.1 一次元配列

では，まずは変数を 5 個，配列として作るにはどのようにしたらよいでしょうか．配列の宣言は次のようになります．

```
int hoge[5];
```

変数の宣言と似ていますが，変数名[3]の後に [5] と書かれています．これは変数を５個作れという命令です．よって，100 個変数を作りたい場合は

```
int hoge[100];
```

とすればよいわけです．

書式

データの型　配列名 [要素の数];

[3]箱の名前の事．ここでは hoge の事です．

2.12.2 配列への値の代入

配列を作った後は値を代入してみましょう．ここで， `int hoge[5];` の配列のイメージは図 2.3 のようになります．

図 2.3: 配列のイメージ

ここで，配列の一番最初の変数に 10 という値を代入したい場合，

```
hoge[0] = 10;
```

とします (図 2.4)．

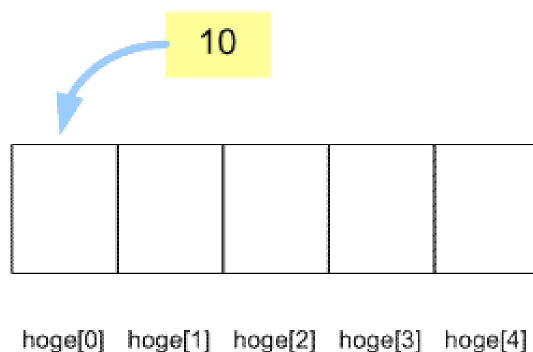

図 2.4: 要素 0 への値の代入

注意点として，図 2.3 を見てわかるように，配列は [0] は必ず 0 から始まります．よって代入できるのは

```
hoge[0]
hoge[1]
hoge[2]
hoge[3]
hoge[4]
```

です．hoge[5] は存在しないので値を代入できない事に注意してください．

2.12.3 配列の初期化

配列を初期化する場合は，つぎのような書き方をすることができます．

```
int hoge[5] = { 1,2,3,4,5 };
```

これは，次のように書いたのと「ほぼ」同じです．[4]

```
hoge[0] = 1;
hoge[1] = 2;
hoge[2] = 3;
hoge[3] = 4;
hoge[4] = 5;
```

箱の中身は図 2.5 となります．

図 2.5: 配列のイメージ

また，次のようにしたらどうなるでしょうか．

```
int hoge[5] = {1,2};
```

残り 3 つはどうなるでしょうか．この足りない部分は 0 が自動的に代入されます．つまり，

```
int hoge[5] = {1,2,0,0,0};
```

と同じです．さて，今度は次のようにしたらどうなるでしょうか．

```
int hoge[] = {1,2,3,4,5};
```

今度は，[] の中の数字を省略しました．このように書くと，コンパイラが要素数を勝手に設定してくれます．つまり，[] の中に 5 を勝手に入れてくれます．しかし，次のようなコードを書くことはできません．

```
int hoge[];
```

これは要素数が 0 なので， `int hoge[0];` という事になりますが，変数を 1 つも作らないというのは意味がありません．こういったことはできないので注意してください．

[4]完全に同じではないです．なぜなら初期化と代入は違うからです．

2.12.4 文字配列

変数にはもちろん文字を入れることもできます．文字を入れる場合は char 型を使うと説明しました．ここで，実際 char 型に文字を入れてみましょう．入れることができるのは英字 1 文字だけです．

```
char str;
str = ’A’;
```

ここで，英字 1 文字を入れる場合，’ ’で囲む事に注意してください．” ”ではないです．

では，次に配列を使ってみましょう．

```
char str[6] = {’H’ , ’E’ , ’L’ , ’L’ , ’O’ , ’\0’ };
printf("%c%c%c%c%c%c",str[0],str[1],str[2],str[3],str[4],str[5]);
```

str[5]，つまり最後の要素に見慣れない文字が含まれています．これは NULL 文字[5]といい，文字列の最後には必ずいれなければならないものです．この記号があると，文章はここで終わりという意味になります．

次に文字配列の中身を printf で表示させています．1 文字表示する場合は%c を使うんでしたね．しかし，上記の書き方，非常に面倒です．そこで，他に方法がないかと探してみると，%s というものが存在します．これは，文字列を表示する際に使用するものです．使い方は次のようになります．

```
char str[6] = {’H’ , ’E’ , ’L’ , ’L’ , ’O’ , ’\0’ };
printf("%s",str);
```

printf 関数で利用している str には添え字，つまり [0] などが付いていませんね．実はこれ，とても難しいことをしているのです．これを理解するには，ポインタの知識が必要です．簡単に説明すると，str は &str[0] の略した書き方なのですが，ぜんぜん訳がわかりませんね．今は文字列を表示したい場合このようにするのだということを覚えておいてください．

さて，気を取り直して，今度は配列の初期化の部分を更に簡単にしてみましょう．上記のコードは，

```
char str[6] = {"HELLO"};
```

と書き直すことができます．さらに略して，

```
char str[6] = "HELLO";
```

[5]ナル，またはヌルと読む

とすることもできます．最後に\0 が付いていませんが，このように書くと文字列の場合，最後に勝手に\0 が付け加えられます．ここで気をつけなければならないことがあります．それは次のような場合です．

```
char str[5] = "HELLO";
```

さきほど説明したように，文字列の最後に\0 が勝手に付け加えられるので，これでは要素が 1 つ足りなくなってしまいます．こういった事に注意してください．

2.12.5 キーボードから文字列の入力

キーボードから文字列を受け取る場合にも scanf 関数を利用します．次のプログラムを見てください．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char str[6];
6
7      printf("何か文字を入力してください：");
8      scanf("%s", str);
9      printf("あなたは%s と入力しましたね.\n", str );
10     return 0;
11 }
```

このプログラムを実行すると，次のようになります．ためしに HELLO と入力してみることとします．

実行結果

何か文字を入力してください：HELLO
あなたは HELLO と入力しましたね.

このように%s を使うことで文字列の代入も簡単にできます．

ただ，文字列を扱う場合には注意が必要です．scanf 関数を注意深く見てください．**str の部分に&が付いていません**．また，

```
scanf("%s", &str[0]);
```

としても, 実は全く同じ意味になります．この部分もポインタという概念を正しく理解していないと分からない部分です．とりあえず今はこういうものだと理解しておいてください．

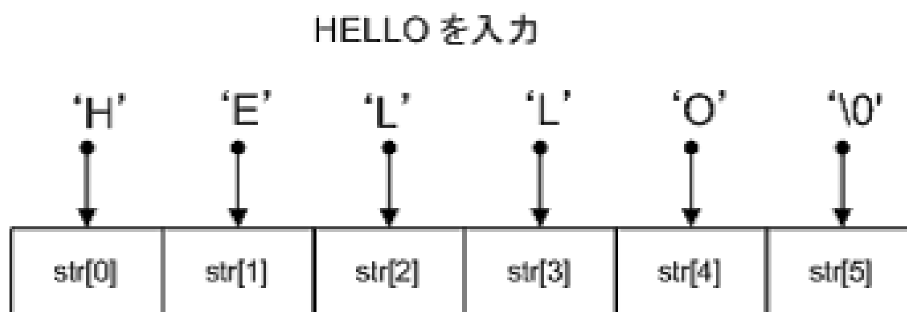

図 2.6: キーボードから文字列の入力

2.12.6 scanf 使用時の注意点

scanf 関数の%s はスペース，改行，タブなどがくると読み込みを終了してしまいます．例えば，次のようなプログラムがあったとします．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char str[100];
6
7      printf("何か文字を入力してください：");
8      scanf("%s", str);
9      printf("受け取った文章:%s.\n", str );
10     return 0;
11 }
```

そしてキーボードから HELLO WORLD と入力してみます．どのように実行されるのでしょうか．

実行結果

何か文字を入力してください：HELLO WORLD
受け取った文章：HELLO

これは予想外の結果です．このように HELLO の後ろに空白があるため読み込みを終了してしまいます．空白も一緒に読み込みたい場合は gets 関数などを使用します．

2.13 文字コード

C 言語には文字コードと呼ばれるものが存在します．文字コードとはなんのことでしょうか．前に 1 文字を保存するために使う変数は char 型だと説明しました．そして char 型の中に入れることができる値は整数の -128 ～ 127（処理系依存）だと言いました．つまり

```
char hoge;
hoge = 97;
```

は問題ないです．これは hoge に 97 という値を代入するということです．しかし，char 型は次のように使っていました．

```
char hoge;
hoge = 'a';
```

さきほど char 型に収納できる値は整数だといいましたが，ここでは文字を代入しているように見えますね．実はこれら 2 つのコードは全く同じ意味です．

始めに説明したとおり，コンピュータは 1 と 0 しか理解することができません．この 1 と 0 を組み合わせると整数を表現することができます．例えば，10 進法で 10 を 2 進法に直すと 1010 となります．しかし 1 と 0 ではいくら頑張っても文字を表現することができません．ここで登場するのが文字コードとなります．

つまり，文字コードとは単なる整数を決められた文字に置き換える役割を果たします．

2.13.1 文字コード表

では，文字コードとはどのようになっているのでしょうか．次に表す表 (図 2.7) は Windows 日本語版で一般的に使われている SHIFT_JIS コード表です．縦軸が 10 の位を，横軸が 1 の位を表しています．

文字コード表では 16 進法で表されています．この文字コード表より，'a' は 16 進法で 61，10 進法で 97 ということがわかります．この '(シングルクウォーテーション）で囲まれた 1 文字が文字コード表により整数に置き換えられます．

ちなみに，C 言語において値を 16 進法で代入する場合，数字の前に 0x をつけます．

```
char hoge = 0x61;
```

図 2.7: JIS コード表

2.13.2 数字の注意点

文字コード表を良く眺めてください．文字コードには数字が含まれています．例えば,

```
char hoge = '1';
```

と書いたとしましょう．これは,

```
char hoge = 1;
```

とは違います．’(シングルクォーテーション) で囲むと文字コード表の整数に置き換えられるという点に注意してください．つまり，’1’ は文字コード表より 0x31 になります.

2.13.3 日本語について

文字コード表を見ると全部で 256 通りの文字しか表現することができません．しかし日本語は漢字，ひらがな，カタカナとどう考えても 256 文字では足りません．日本語の場合は特殊で，2 つ (またはそれ以上) の char 型で一文字を表現します．これらの日本語の話はとても複雑なので省略します.

2.13.4 文字化けについて

文字コードは 1 種類ではありません．文字コードにはいろいろな種類が存在します．これらの文字コードの配置の違いにより，適切な文字コードを選択しないと文字化けという現象が起こります.

2.14 例題

2.14.1 メートルをフィートへ変換するプログラム

仕様

メートルからフィートへ変換するプログラムを作成する．メートルに 3.2 をかけることによってフィートは求めることができる.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       double meter;
6       double answer;
7
8       printf("メートルを入力してください :");
9       scanf("%lf",&meter);
10      answer = meter * 3.2;
11      printf("%f メートルは%f フィートです\n", meter, answer);
12      return 0;
13  }
```

実行結果

メートルを入力してください :27
27.000000 メートルは 86.400000 フィートです

2.14.2 空白を読み込む scanf

2.12.6 において「HELLO WORLD」と入力し Enter キーを押すと「HELLO」しか表示されませんでした．これを改良するのに gets() 関数を使うほか，scanf() には, スキャン集合という非常に面白い機能があります．たとえば%[0-9] と指定すると数字のみを読み込むことになります．この性質を利用することで空白も一緒に読む込むことができます．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       char str[80];
6
7       printf("enter :");
8       scanf("%[a-zA-Z ]",str);
9       printf("answer :%s\n",str);
10      return 0;
11  }
```

実行結果

enter :HELLO WORLD
answer :HELLO WORLD

2.15 演習問題

問題 1

分を時間に変換するプログラムを作成しなさい．例えば，ユーザが 180 と入力したら 3 と出力するようにせよ．

問題 2

セ氏温度 (C) を華氏温度 (F) に変換するプログラムを作成せよ．セ氏温度 (C) と華氏温度 (F) には次のような関係がある．

$$F = \frac{9}{5}C + 32$$

問題 3

次のプログラムの出力結果を予測せよ．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 3, j = 5;
6      printf("%d", i / j);
7      return 0;
8  }
```

問題 4

次のプログラムの問題点を指摘しなさい．

```
1  #include <stdio.h>
2
3  int main(void)
```

```
4   {
5       char str[5] = "HELLO WORLD";
6       printf("%s", str);
7       return 0;
8   }
```

問題 5

2.5 のプログラムにおいて，不適切な数値が表示された．このプログラムをキャストを使うことによって，正しい動作をするプログラムに書き換えよ．

第3章　制御文

この章では，条件分岐，及び繰り返し構文を学んでいきます．これらの構文を使わないでプログラムを作ることは不可能です．とても大切な文法なので，是非マスターしましょう．

3.1　条件分岐 if else 構文

if else 構文は，条件分岐の際に利用する構文です．if は英語で「もし〜だったら」，else は「そうでなかったら」という意味です．例えば，もしテストの点数が 60 点以上だった場合には合格時の処理を，そうでなかったら不合格時の処理をするといった場合に使います（図 3.1）．

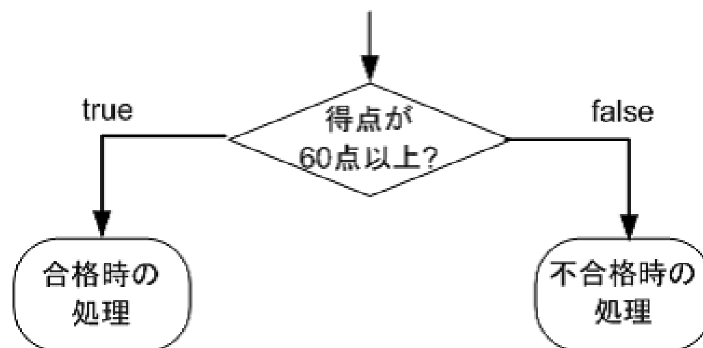

図 3.1: if else のイメージ

3.1.1　if else 文の書き方

実際に if else 構文を使ってプログラムを書いてみましょう．int 型の hoge という変数があるとします．この hoge の値が 10 の場合 A の処理を，そうでない場合 B の処理をするプログラムを書いてみます．

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

```
2
3  int main(void)
4  {
5      int hoge = 10;
6
7      if( hoge == 10 ) {
8          /* 処理 A */
9      } else {
10         /* 処理 B */
11     }
12 }
```

if 構文の場合，if( ) のカッコの中には条件を書きます．カッコの中の条件式が真 (true) である場合は処理 A を，偽 (false) である場合は処理 B を行います．上記の例の場合,hoge が 10 なので，処理 A を実行します．

ここで注意して欲しいのは,hoge = 10 ではなく,hoge == 10 であるということです．= とは代入という意味だと説明しました. 数学で言うイコール (左辺と右辺が等しい）という意味にしたい場合は, ==とイコールを二つつなげて書かなければならないことに注意してください.

さて, 処理 A しか必要が無い時には次のように書くことができます.

```
if( hoge == 10 ) {
    /* hoge が 10 の時の処理をここに書く */
}
```

また, 条件を複数書きたい場合, 次のようにします.

```
int hoge = 8;

if( hoge == 10 ){
    /* A の処理 */
}else if( hoge == 9) {
    /* B の処理 */
}else if( hoge == 8 ) {
    /* C の処理 */
}else{
    /* D の処理 */
}
```

上記のプログラムは，C の処理が実行されることになります．ここでもし hoge が 9 であるならば B の処理が行われますし,6 ならば D の処理が実行されることになります.

3.1.2 非 0 の判定式

次のようなプログラムを書くこともできます.

```
int hoge = 1;

if( hoge ){
    /* 何か処理を書く */
}
```

さて, カッコの中には何か条件式を書かなくてはなりませんが, 上記の例の場合は変数名の hoge としか書いてありません. これはどういう意味でしょうか.

実はこういった場合, カッコの中の値が 0 であるか 0 以外であるかで判定します. ここでは,hoge が 0 ではない場合 (true）処理を行い,0 と等しい場合（false）は処理を行いません．つまり, 上記の式は以下のように書き換えることができます.

```
if( hoge != 0 ) {
    /* 何か処理を書く */
}
```

ここで新しい表現 != というものが出現しました. これは，hoge が 0 でなかったらということを意味しています.

では別の例を挙げてみます.

```
if( 1 ) {
    /* A の処理 */
} else {
    /* B の処理 */
}
```

この場合，常に A の処理が実行されます. 先ほど説明したように 0 以外の数字の時は A の処理が実行されます．つまり，もしカッコの中の数字を 0 にしたら B の処理が実行されます.

次のプログラムを見てください．

```
if( hoge = 3 ) {
    /* hogeが3の場合，処理Aを実行 */
} else {
    /* hogeが3以外の場合，処理Bを実行 */
}
```

これは，C 言語でのプログラミング経験がある方は，一度は経験していると思われる典型的なミスです．条件式を見てください． `hoge == 3` ではなく， `hoge = 3` となっています．この場合，コンパイルは成功しますが，常に処理 A が実行されることとなります．なぜなら，hoge には 3 が代入され，非ゼロの判定式より，常に条件が true となるからです．こういったミスを起こさないようにするため，次のように条件式を書く人もいます．

```
if( 3 == hoge ) {
    /* hogeが3の場合，処理Aを実行 */
} else {
    /* hogeが3以外の場合，処理Bを実行 */
}
```

この場合は，例え `3 = hoge` と誤って書いてしまっても，数値に変数を代入することができないため，コンパイルエラーが発生します．よって，間違えを未然に防ぐことができます．ただ，最近のコンパイラは賢いので，if の条件に `hoge = 3` と書くと，警告を発してくれます．
余談ですが，JAVA 言語ではこのようなミスを起こさないように，非ゼロの判定式は書くことができないようになっています．

3.1.3 関係演算子

２つの変数を比較して,式が正しいか誤りかを求める時に使う演算子を関係演算子といいます. 例えば，表 3.1 のようなものが存在しています.

例えば，i と j の値を比較し，i が j より小さかったら真，そうでなかったら偽としたい場合，次のようなプログラムを書くことができます.

表 3.1: 関係演算子

`x <  y`	x が y より小さかったら真, そうでなかったら偽
`x >  y`	x が y より大きかったら真, そうでなかったら偽
`x <= y`	x が y 以下であれば真, そうでなかったら偽
`x >= y`	x が y 以上であれば真, そうでなかったら偽
`x == y`	x が y と等しければ真, そうでなかったら偽
`x != y`	x が y と等しくなければ真, そうでなかったら偽

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int i, j;
6
7       printf("i の値を入力してください :");
8       scanf("%d", &i); /* i を入力する */
9       printf("j の値を入力してください :");
10      scanf("%d", &j); /* j を入力する */
11
12      if( i < j ) {
13          /* もし i が j より小さかったら */
14          printf("i は j より小さい値です. \n");
15      } else {
16          printf("i は j より大きい値です. \n");
17      }
18  }
```

i には 10 を，j には 5 を入力したとすると，このプログラムの実行結果は次のようになります.

実行結果

```
i の値を入力してください : 10
j の値を入力してください : 5
i は j より大きい値です.
```

3.1.4 論理演算子

次のようなプログラムを書いてみます.

```
int hoge = 10, piyo = 20;

if( hoge == 10 ) {
    if( piyo == 20 ) {
        /* 処理 A を行う */
    }
}
```

さて, この式は hoge が 10 であり, かつ piyo が 20 である時, 処理 A が実行されるということです．ここで論理演算子 (表 3.2) を利用すると，一つの if 構文でこの条件を書き換えることができます.

表 3.2: 論理演算子

x == 10 && y == 20	x が 10 かつ y が 20
x == 10 || y == 20	x が 10 または y が 20
!x	x の否定

先ほどのプログラムを論理演算子を使って書き直すと，次のようになります.

```
if( hoge == 10 && piyo == 20){
    /* 処理 A */
}
```

表 3.2 にある x の否定とはどういうことを意味しているでしょうか．例えば，次のように書くことができます.

```
int hoge = 10;
if( !hoge ){
    /* 処理 A */
}
```

条件が !hoge ではなく hoge であった場合，非ゼロの判定式より，処理 A が実行されます．しかし，この場合は hoge が 0 以外の場合は条件が偽となり，処理 A が実行されません．つまり，非ゼロの判定式の真偽をひっくり返すこととなります.

3.2 条件分岐switch構文

if構文をマスターしたところで, 次はswitch構文を学んでみましょう．switch構文は，条件分岐が複雑になる場合に用いられる構文です．ここで,if構文を使ったプログラムの例をお見せします.

```
1   int hoge = 8;
2
3   if( hoge == 10 ){
4       /* Aの処理 */
5   }else if( hoge == 9) {
6       /* Bの処理 */
7   }else if( hoge == 8 ) {
8       /* Cの処理 */
9   }else{
10      /* Dの処理 */
11  }
```

if構文の節でも登場したプログラムです．特に問題の無いプログラムです．このプログラムのイメージは図3.2です.

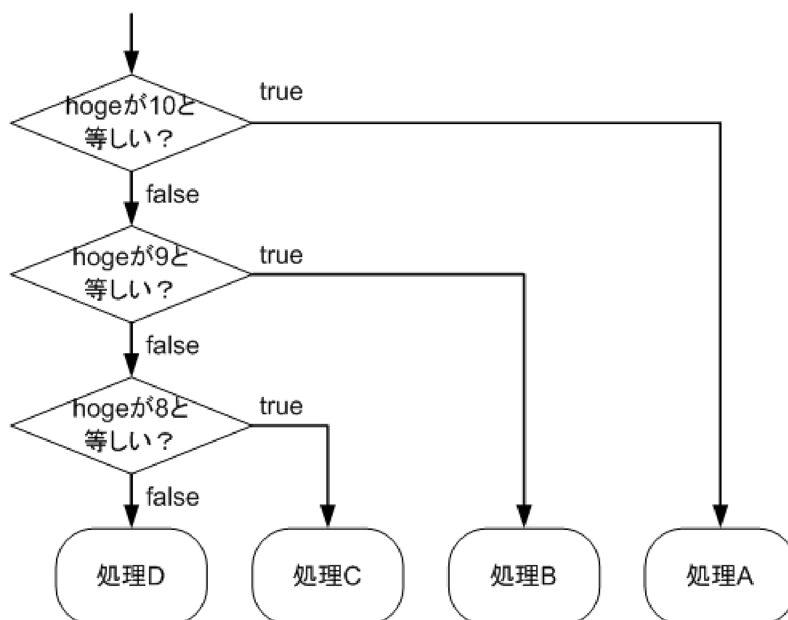

図 3.2: if elseによる条件分岐

しかし, ここで hoge が 7 である場合,hoge が 6 である場合 ··· と考えていくと, なんだか見難い書き方になってしまいます.[1] これを解消するのが switch 構文です. ここで switch 構文を使って書き直してみましょう.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int hoge = 8;
6       switch( hoge ) {
7       case 10: /* 処理 A */
8           break;
9       case 9 : /* 処理 B */
10          break;
11      case 8 : /* 処理 C */
12          break;
13      default: /* 処理 D */
14          break;
15      }
16      /* 処理 E */
17  }
```

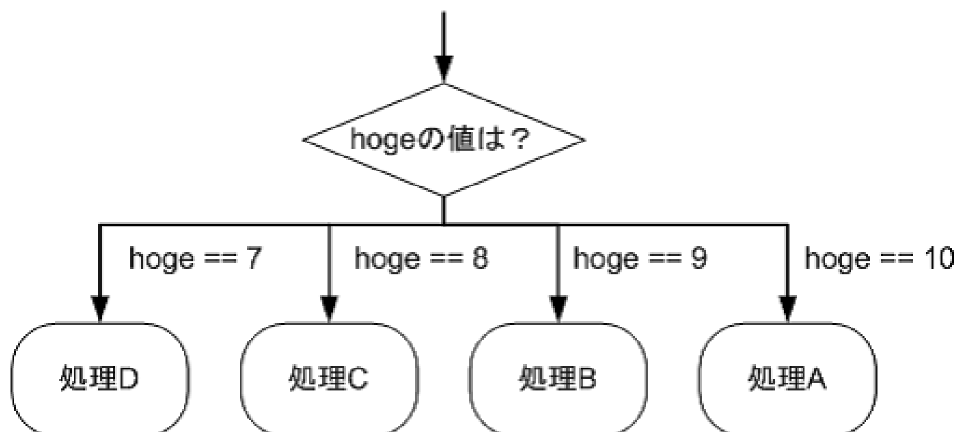

図 3.3: switch による条件分岐

switch 構文のイメージ図は図 3.3 です．if-else は上から順に比較を行い，条件が正しいかを判定していきますが，switch の場合は，条件の場所に飛んでいく感じです.

[1] 見難いだけでなく，実は処理的にも遅くなる場合があります.

switch に続くカッコの中には何か変数を書きます．このカッコ内に書くことができるのは，**整数を持った値のみ**です．この場合 hoge と書いただけなので,hoge の中の値によって処理を分けます．例えば hoge の値が 8 であった場合,case 8: という場所に飛びます．途中 break というものが存在しますが,break とは switch 構文から飛び出るということを意味しています．例えば case 8: に飛んだ場合，処理 C を実行した後処理 E が実行されます．

default というのはどの条件にも当てはまらなかった時に実行される構文です．if 文でいう else のような意味を表しています．

3.2.1 break を書かない switch 構文

switch 構文の中で break を書かないとどうなるでしょうか．次のコードを実行してみましょう．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int hoge = 9;
6       switch( hoge ) {
7       case 10: printf("10\n");
8
9       case 9 : printf("9\n");
10
11      case 8 : printf("8\n");
12
13      default: printf("7\n");
14
15      }
16      return 0;
17  }
```

実行結果

```
9
8
7
```

C 言語は上から下へコードが実行されるので，case 9:に飛んだ後はそのまま下のコード，つまり case 8:のコードが実行されます．

このようなテクニック (fall through と呼ばれる) が使われることも稀にありますが，基本的には使わないようにしてください．もしくは，使う場合はコメントで fall through と書き込むようにしましょう．ただ単に break を書き忘れたのだろうと勘違いされる場合があります.

3.3 繰り返し do while for 構文

do while for 構文は，繰り返しの処理を行いたい場合に利用されます.

3.3.1 do while 構文

Hello world と 10 回表示するプログラムを do while を使って作ってみます.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int hoge = 10;
5  
6      do{
7          printf("%d 回目:Hello world\n", 11 - hoge);
8          hoge--;
9      }while( hoge > 0);
10 
11     return 0;
12 }
```

このプログラムの実行結果は以下のようになります.

実行結果

```
1 回目:Hello world
2 回目:Hello world
3 回目:Hello world
4 回目:Hello world
5 回目:Hello world
6 回目:Hello world
7 回目:Hello world
8 回目:Hello world
9 回目:Hello world
10 回目:Hello world
```

do while 構文の書式は次のようになっています.

書式

```
do { 処理 } while( 条件式 );
```

do{ }の中に何か処理を書きます. そして条件式には条件を書きます. この条件が真の場合は, また do{ の部分に戻って処理を繰り返します（図 3.4）.

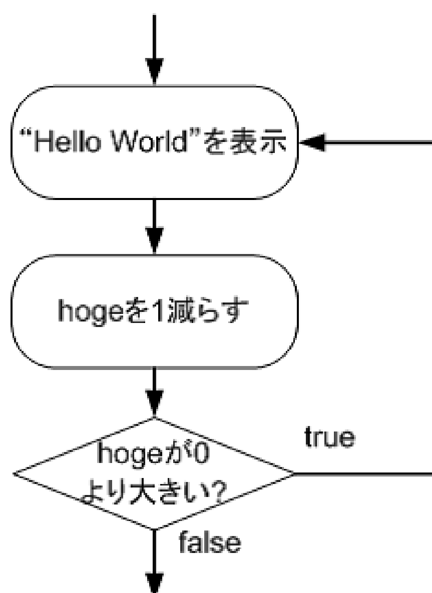

図 3.4: do while による繰り返し

よく忘れるのですが,**while の後にセミコロンを付ける**ことに注意してください.

3.3.2 while 構文

do while 構文より while 構文だけの方がよく使われます. do while と while の違いは,条件式を最初に判定するか後で判定するかの違いだけです (図 3.5）.

また, do while 構文は, ループを脱出するかの判定が最後にあるため, 処理が最低 1 回は必ず実行されますが, while 構文の場合は, 処理が一回も実行されないことがあります.

Hello world と 10 回表示するプログラムを while を使って書き直すと次のようになります.

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

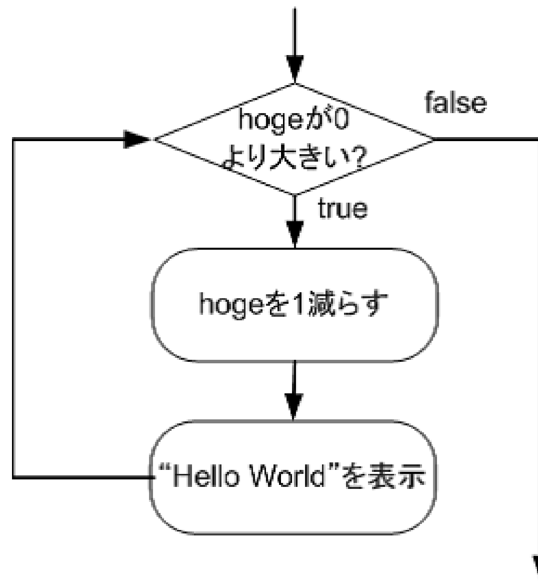

図 3.5: while による繰り返し

```
 2  int main(void)
 3  {
 4      int hoge = 10;
 5      while( hoge > 0 ) {
 6          printf("%d 回目:Hello world\n", 11 - hoge);
 7          hoge--;
 8      }
 9      return 0;
10  }
```

while 構文の書式は次のようになります.

書式

```
while( 条件式 ) { 処理 }
```

条件式は if 構文と同じように書くことができます．同じということは，次のように書いたらどのような実行結果となるでしょうか.

```
while( 1 ) {
    printf("Hello world\n");
}
```

非ゼロの判定式を思い出してください．これは, 常に真ということを表しています. つまり無限ループに陥ります.

3.3.3 for 構文

for 構文は while 構文と同様，繰り返しの処理をしたい場合に利用されます．前節の while を使ったプログラムを，for を使って書き直してみます.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int hoge;
6      for( hoge = 10; hoge > 0 ; hoge--) {
7          printf("%d回目:Hello world\n",11-hoge);
8      }
9      return 0;
10 }
```

for 構文の書式は以下のようになっています.

書式

```
for( 初期条件 ; 条件式 ; 後始末 ) { 処理 }
```

初期条件には変数の初期化などを書きます．条件式は if と同じです．後始末とは処理を最後まで実行した後，その部分が実行されます．例では 〜回目：Hello world と表示した後に後始末の部分 `hoge--` が実行されるということです.

初期条件や後始末, 条件式などは省くことができます. 例えば,

```
int hoge = 10;
for( ; hoge > 0 ; hoge--) {
    printf("%d回目:Hello world\n",11-hoge);
}
```

とすることもできます.

では，for と while を比較してみましょう.

while の場合

```
処理A;
while( 処理B ) {
    なにかしたい処理
    処理C;
}
```

これを for で書き直すと，以下のようになります.

for の場合

```
for( 処理A ; 処理B ;  処理C ) {
    なにかしたい処理
}
```

for を使った無限ループも while 同様，作ることができます. それは条件式になにも書かなければよいのです.

```
for(;;){ なにか処理 }
```

3.3.4 break を使ったループの脱出

繰り返し処理を実行している間に break を使うと，いつでもループから脱出することができます．例えば次のようなプログラムを書いたとします.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6
7      for( i = 0; i < 10; i++ ) {
8          printf("iの値:%d\n", i);
9          if( i == 5 ) {
10             break;
11         }
12     }
13     printf("脱出\n");
14     return 0;
15 }
```

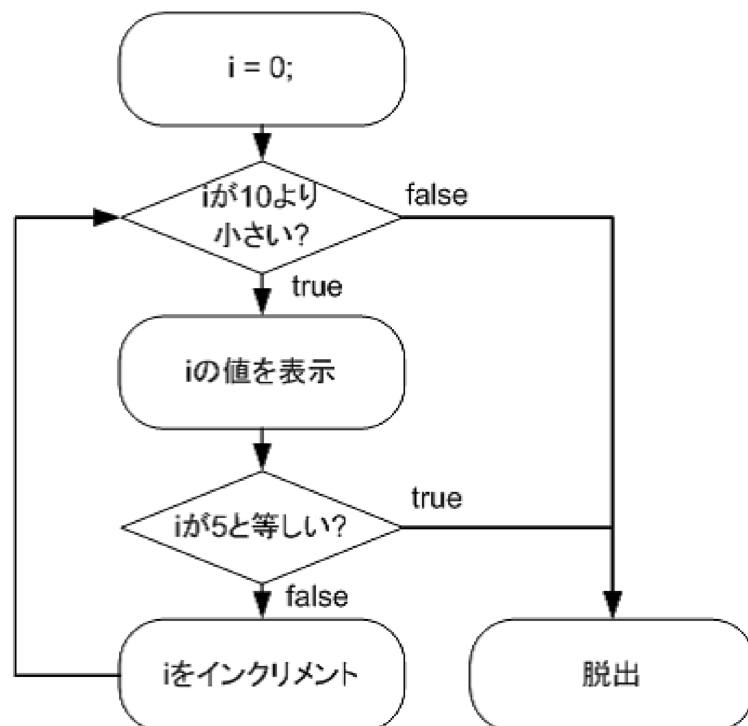

図 3.6: break によるループの脱出

このプログラムの流れ図は図 3.6 となります．また，実行結果は以下のようになります．

実行結果

```
i の値:0
i の値:1
i の値:2
i の値:3
i の値:4
i の値:5
脱出
```

このように break を使うことで，通常の終了判定を経ずにループを脱出することも可能です．

3.3.5 continue 文

continue はループの次の繰り返しをすぐさま実行したい場合に使用されます．次のコードを見てください．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6      for( i = 0; i < 5; i++ ) {
7          printf("iの値:%d\n", i);
8          if( i <= 2 ) {
9              continue;
10         }
11         /* i <= 2の場合，この下の行は実行されない */
12         printf("LOOP-----\n");
13     }
14     return 0;
15 }
```

このプログラムの実行結果は以下のようになります．

実行結果

```
iの値:0
iの値:1
iの値:2
iの値:3
LOOP——
iの値:4
LOOP——
```

continue も break 同様，for do while などの繰り返しの中で使用することができます．

3.4 ビット演算

ビット演算は，ビット単位で値を操作する演算です．この演算に関しては，2進数，16進数の概念を把握していないと理解することができません．実際のソフトウェア開発においては，ビット演算はよく使われますが，初心者の内は，ビット演算を知らなくてもさほど問題は無いと思います．よって，難しいと感じたら，この節は飛ばしてしまってもかまいません．

3.4.1 ビットごとの演算子

C 言語にはビット単位で演算を行う特別な演算子として, 次の 4 つが用意されています．ビットとは 0 か 1 で表されるコンピュータが扱うことのできるデータの最小単位のことです.

記号	意味
&	ビットごとの AND(論理積)
|	ビットごとの OR（論理和）
^	ビットごとの XOR（排他的論理和）
~	NOT（否定，ビットを反転させる）

AND は，すべてのビットが 1 である場合に 1 が出力されます．例えば，次のような結果となります.

```
0 & 0 => 0
0 & 1 => 0
1 & 0 => 0
1 & 1 => 1
```

2 つのビットが 1 の場合に 1 が出力されます．AND は，日本語で論理積と呼ばれています．つまり，掛け算だと思えばよいわけです．0 に何をかけても 0 となります．すべての値が 1 である場合のみ，1 が出力されるわけです.

OR は，いずれかのビットが 1 である場合に 1 が出力されます．例えば，次のような結果となります.

```
0 | 0 => 0
0 | 1 => 1
1 | 0 => 1
1 | 1 => 1
```

OR は日本語で論理和といわれています．つまり，足し算だと思えばよいわけです．$0 + 0$ は 0 ですが，それ以外の場合，1 つでも 1 があれば出力が 1 となるわけです.

XOR(又は EOR) は日本語で排他的論理和とも呼ばれています．これはやや難しい考えです．XOR は 2 つのビットが与えられた場合，1 つだけが 1 である場合に，出力が 1 となります．例えば，次のような結果となります.

```
0 ^ 0 => 0
0 ^ 1 => 1
1 ^ 0 => 1
1 ^ 1 => 0
```

NOT は 1 の補数とも呼ばれ，単にビットを反転させるだけです．例えば，次のような結果となります.

```
~0 => 1
~1 => 0
```

では，これらのビット演算を使った例をお見せします.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      unsigned char and,or,xor,not;
6      unsigned char hoge,piyo;
7
8      hoge = 187; /* 10111011 */
9      piyo = 86;  /* 01010110 */
10
11     and = hoge & piyo;
12     or = hoge | piyo;
13     xor = hoge ^ piyo;
14     not = ~piyo;
15
16     printf("hoge AND piyo = %d\n",and);
17     printf("hoge  OR piyo = %d\n",or);
18     printf("hoge XOR piyo = %d\n",xor);
19     printf("     NOT piyo = %d\n",not);
20
21     return 0;
22 }
```

10 進数での 187 は，2 進数で 10111011 と直すことができます．この 10 進数から 2 進数へ直す方法についてはここでは説明しません．興味のある方はインターネット等で調べてみてください．このプログラムの実行結果は以下のようになります.

実行結果

hoge AND piyo = 18
hoge OR piyo = 255
hoge XOR piyo = 237
NOT piyo = 169

```
    hoge   1 0 1 1 1 0 1 1 (187)       1 0 1 1 1 0 1 1 (187)
AND piyo   0 1 0 1 0 1 1 0 (86)   OR   0 1 0 1 0 1 1 0 (86)
    --------------------------      ----------------------
           0 0 0 1 0 0 1 0 (18)        1 1 1 1 1 1 1 1 (255)


    hoge   1 0 1 1 1 0 1 1 (187)
XOR piyo   0 1 0 1 0 1 1 0 (86)   NOT  0 1 0 1 0 1 1 0 (86)
    --------------------------      ----------------------
           1 1 1 0 1 1 0 1 (237)       1 0 1 0 1 0 0 1 (169)
```

3.4.2 シフト演算子

C 言語にはビットをシフトさせるビットシフト演算子があります．例えば，

```
00000011
```

を左に 2 つシフトすると，

```
00001100
```

となります．ビット全体を指定した分だけ左にずらします．ビット右側からは 0 が挿入されます．

ビットシフト演算子の一般的な形式は次のとおりです．

値 << ビット数	左にビット数だけシフト
値 >> ビット数	右にビット数だけシフト

左シフトは単純ですが，右シフトは左シフトほど単純ではありません．[2] ここでは，右シフトの説明は省略することとします．

ビットシフトの例をみてみましょう．

```
1  #include <stdio.h>
2  int main(void)
3  {
4      unsigned char hoge = 4;
5
6      hoge = hoge << 3;
7      printf("hoge を左に 3 シフトさせると :%d",hoge);
8      return 0;
9  }
```

[2] 右シフトには，論理シフトと算術シフトと呼ばれるものが存在します．

実行結果

hoge を左に 3 シフトさせると :32

このように hoge に 8 をかけたのと同等の値になります

```
hoge       0 0 0 0 0 1 0 0 (4)
-------------------------------
hoge << 3  0 0 1 0 0 0 0 0 (32)
```

3.5 例題

3.5.1 文字列をコピーするプログラム

仕様

キーボードから英単語を受け取り, その英単語を別の配列にコピーするプログラムを作成する.

```
1  /* 文字列コピープログラム 1 */
2  #include <stdio.h>
3
4  int main(void)
5  {
6      char source[100] , str[100];
7      int i;
8
9      printf("英単語を入力してください:");
10     scanf("%s", source );
11
12     for( i = 0; source[i] != '\0' ; i++ ){
13         str[i] = source[i];
14     }
15     str[i] = '\0';
16
17     printf("コピーした文字列は %s です",str);
18     return 0;
19 }
```

文字列の最後には NULL 文字があることを思い出してください．for 構文で source の 0 番目からの要素を str にコピーしていきます．source[i] に NULL 文字が登場したら，ループを脱出します．

別の方法も存在します. 解答は一通りとは限りません.

```
1  /* 文字列コピープログラム２ */
2  #include <stdio.h>
3
4  int main(void)
5  {
6      char source[100] , str[100];
7      int i;
8
9      printf("英単語を入力してください:");
10     scanf("%s", source );
11
12     i = 0;
13     while( source[i] != '\0' ) {
14         str[i] = source[i];
15         i++;
16     }
17     str[i] = '\0';
18
19     printf("コピーした文字列は %s です",str);
20     return 0;
21 }
```

実行結果

英単語を入力してください:HELLOWORLD
コピーした文字列は HELLOWORLD です

3.5.2　最大値を求めるプログラム

仕様

キーボードから 10 個の数字を受け取り, その最大値を表示するプログラムを作成する.

```
1   /* 最大値を求めるプログラム */
2   #include <stdio.h>
3
4   int main(void)
5   {
6       int array[10];
7       int i,j,max;
8
9       printf("10個の数字を入力してください\n");
10
11      for( i = 0 ; i < 10 ; i++ ) {
12          printf("%d回目:",i+1);
13          scanf("%d", array );
14      }
15
16      for( j = 1 , max = array[0] ; j < 10; j++) {
17          if( max < array[j] ) {
18              max = array[j];
19          }
20      }
21
22      printf("一番大きな数字は%dです",max);
23      return 0;
24  }
```

実行結果

```
1回目:5
2回目:8
3回目:13
4回目:4
5回目:90
6回目:54
7回目:30
8回目:52
9回目:70
10回目:1
一番大きな数字は90です
```

3.5.3 九九表を作成するプログラム

仕様

九九を表示するプログラムを作成する. 表示結果は以下のようにする.

```
  1   2   3   4   5   6   7   8   9
  2   4   6   8  10  12  14  16  18
  3   6   9  12  15  18  21  24  27
  4   8  12  16  20  24  28  32  36
  5  10  15  20  25  30  35  40  45
  6  12  18  24  30  36  42  48  54
  7  14  21  28  35  42  49  56  63
  8  16  24  32  40  48  56  64  72
  9  18  27  36  45  54  63  72  81
```

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int i,j;
6
7       for( i = 1 ; i < 10 ; i++ ) {
8           for(j = 1 ; j != 10 ; j++ ) {
9               printf("%3d ",i * j);
10          }
11          printf("\n");
12      }
13      return 0;
14  }
```

printf の中に%3d というものを使っていますが，これは 3 文字分のスペースを確保して表示するという意味です. このようにすると綺麗に表示することができます. 5 文字分のスペースを確保したい場合は,%5d とすればよいわけです.

3.5.4 文字コード表を表示するプログラム

仕様

０～２５５までの文字コードを表示するプログラムを作成せよ．10 進法の値，16 進法の値，文字を表示することとする．

```
0 0
1 1
```

（略）

```
65 41 A
66 42 B
```

（略）

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int hoge;
6
7      for( hoge = 0; hoge < 256 ; hoge++ ) {
8          printf("%3d %3x %c\n",hoge,hoge,hoge);
9      }
10     return 0;
11 }
```

3.5.5 足し算，引き算を行うプログラム

仕様

利用者の選択により足し算,引き算ができるプログラムを作成する.

```
1  #include <stdio.h>
2
3  int main(void)
```

```
4  {
5      int hoge, piyo;
6      int select;
7
8      printf("1 :足し算を行う\n"
9             "2 :引き算を行う\n");
10     printf("番号を入力してください :");
11
12     scanf("%d",&select);
13     printf("hoge :"); scanf("%d",&hoge);
14     printf("piyo :"); scanf("%d",&piyo);
15
16     switch(select) {
17     case 1: printf("%d + %d = %d\n",hoge,piyo, hoge + piyo);
18         break;
19     case 2: printf("%d - %d = %d\n",hoge,piyo, hoge - piyo);
20         break;
21     }
22     return 0;
23 }
```

実行結果

1 :足し算を行う
2 :引き算を行う
番号を入力してください :1
hoge :15
piyo :10
15 + 10 = 25

3.5.6 合格者判定プログラム

仕様

5 人の受験者の得点を受け取り,60 点以上の合格者の番号と得点を表示させるプログラムを作成する.

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

```
2
3  int main(void)
4  {
5
6      int student[5];
7      int i;
8
9      for(i = 0; i < 5; i++){
10         printf("受験番号%d番の得点 :",i + 1);
11         scanf("%d",&student[i]);
12     }
13     for(i = 0; i < 5; i++){
14         if(student[i] > 60) {
15             printf("受験番号%d番  %d点で合格\n",i+1, student[i]);
16         }
17     }
18     return 0;
19 }
```

実行結果

```
受験番号1番の得点 :68
受験番号2番の得点 :57
受験番号3番の得点 :30
受験番号4番の得点 :90
受験番号5番の得点 :88
受験番号1番　68点で合格
受験番号4番　90点で合格
受験番号5番　88点で合格
```

3.5.7 ネットワークアドレスを求めるプログラム

仕様

つぎの IP アドレスのネットワークアドレスを求めるプログラムを作成する．
ネットワークアドレスは IP アドレスとサブネットマスクとの AND を取る事により求めることができる．
IP アドレス 192.168.12.9
サブネットマスク 255.255.255.0

```
1   #include <stdio.h>
2   int main(void)
3   {
4       unsigned char ip_address[4] = {192,168,12,9};
5       unsigned char subnetmask[4] = {255,255,255,0};
6       unsigned char net_add[4] = {0};
7       int i;
8
9       for(int i = 0; i < 4; i++){
10          net_add[i] = ip_address[i] & subnetmask[i];
11      }
12      printf("network_address is %d.%d.%d.%d\n",
13          net_add[0],net_add[1],net_add[2],net_add[3]);
14      return 0;
15  }
```

実行結果

network_address is 192.168.12.0

3.6 演習問題

問題 1

キーボードから数字を受け取り, その絶対値を表示するプログラムを作成せよ. 絶対値とは数字 0 からの距離のことである．例えば，-7 ならば 0 からの距離は 7 なので絶対値は 7 である．

問題 2

3 つの数字を入力し, 最小値を求めるプログラムを作成せよ.

問題 3

複数の値を入力し，その平均値を求めるプログラムを作成せよ.

問題 4

キーボードから英単語を受け取り, その文字数を数えるプログラムを作成せよ.
ヒント：文字列の終わりには必ず NULL 文字（\0) がある.

問題 5

*****
***
*
*
***
*****

この形を作るプログラムを書け.

問題 6

10 人の生徒の得点を受け取り，もっとも点数の高かった人，低かった人の番号と得点を表示させるプログラムを作成せよ.

問題 7

Fizz Buzz は英語圏で長距離ドライブ中や飲み会の時に行われる言葉遊びである. 遊び方はいたってシンプルである．まず，プレイヤーは円状に座る．最初のプレイヤーは「1」と数字を発言する．次のプレイヤーは直前のプレイヤーの次の数字を発言していく．ただし，3 で割り切れる場合は 「Fizz」，5 で割り切れる場合は「Buzz」，両者で割り切れる場合は 「Fizz Buzz」を数の代わりに発言しなければならない．発言を間違えた者やためらった者は脱落となる [wikipedia より引用].

ゲームは以下のとおりに発言が進行する.

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz Buzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz, Buzz, 26, Fizz, 28, 29, Fizz Buzz, 31, 32, Fizz, 34, Buzz, Fizz, ...

1〜100 までの整数を出力するプログラムを作成しなさい．ただし，3 の倍数の時は数字ではなく Fizz と，5 の倍数の時は Buzz と出力し，3 と 5 両方の倍数の場合には FizzBuzz と出力すること．また，剰余（modulo）演算子 `%` を使ってプログラムを作成しなさい．

問題 8

Fizz Buzz 問題のプログラムを剰余演算子を使わずに作成せよ．

3.7 応用演習問題

応用演習問題には，いままで説明していない知識を使う問題が含まれています．余裕があったら考えてみてください．

応用問題演習 1

次のプログラムにおいて hoge, piyo, foo, bar はそれぞれ何回表示されるか予測せよ．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5           int hoge = 0, piyo = 0;
6           int foo = 0, bar = 0;
7
8           do{
9                   printf("hoge\n");
10          }while( hoge++ < 3);
11
12          while( piyo++ < 3){
13                  printf("piyo\n");
14          }
15
```

```
16          do{
17                  printf("foo\n");
18          } while(++foo < 3);
19
20          while(++bar < 3){
21                  printf("bar\n");
22          }
23          return 0;
24  }
```

応用問題演習 2

IPAddress 124.33.214.39, Subnetmask 255.255.255.224 においてこの所属サブネットと所属するサブネットのブロードキャストアドレスを求めるプログラムを作成せよ.

応用問題演習 2 解答

所属サブネット　124.33.214.32
ブロードキャストアドレス　124.33.214.63

応用問題演習 3

企業の常時接続を利用して今度 WWW サーバ、メールサーバなどを公開することになった. ISP にグローバルアドレスをくれるよう申請したところ配布されるグローバルアドレスは「122.18.130.93 /29」であった. いくつのサーバに固定 IP アドレスを振り分けられるか求めるプログラムを作成せよ.

応用問題演習 3 解答

```
1   #include <stdio.h>
2   int main()
3   {
4           unsigned char ip_address[4] = {210,175,166,160};
5           unsigned char subnetmask[4] = {255,255,255,248};
6           unsigned char net_add[4] = {0};
7           unsigned char bro_add[4] = {0};
8           int host_num = 0;
```

```
 9          int i;
10
11          for(int i = 0; i < 4; i++){
12                  net_add[i] = ip_address[i] & subnetmask[i];
13          }
14
15          /* ブロードキャストアドレスを求める */
16          for(int i = 0; i < 4; i++){
17                  if(subnetmask[i] != 255){
18                          bro_add[i] = net_add[i] + ~subnetmask[i];
19                  }else{
20                          bro_add[i] = ip_address[i];
21                  }
22          }
23
24          printf("network_address is %d.%d.%d.%d\n",
25                  net_add[0],net_add[1],net_add[2],net_add[3]);
26          printf("broadcast_address is %d.%d.%d.%d\n",
27                  bro_add[0],bro_add[1],bro_add[2],bro_add[3]);
28
29          for(int i = 0; i < 4; i++){
30                  host_num += (bro_add[i] - net_add[i]) << 24 - 8 * i;
31          }
32
33          /* はじめと最後のアドレスは使えないので */
34          host_num = host_num - 1;
35          printf("利用可能なホストの数は %d です\n",host_num);
36
37          return 0;
38  }
```

第4章　関数

4.1　関数とは

C 言語に欠かせないものの一つとして関数というものが存在します．実は今まで関数を何回も使っていました．例えば printf ですが，これは関数です．scanf も関数です．関数とはいくつかの命令文を集めて一つの塊としたものです．

4.2　関数を作ってみる

2 つの値を入力して，その和を求める関数を作ってみましょう．

```
1   #include <stdio.h>
2
3   int Plus(int x, int y)
4   {
5           int result;
6           result = x + y;
7           return result;
8   }
9
10  int main(void)
11  {
12          int hoge,piyo;
13          int n;
14
15          printf("2 つの整数を入力してください\n");
16          printf("1 つ目:"); scanf("%d",&hoge);
17          printf("2 つ目:");  scanf("%d",&piyo);
18
19          n = Plus(hoge,piyo);
20          printf("足し算の結果は%d です\n",n);
21
```

```
22          return 0;
23   }
```

このプログラムの実行結果は次のようになります．例として，1つ目には数値5を，2つ目には10を入力したとします．

実行結果

```
2つの整数を入力してください
1つ目:5
2つ目:10
足し算の結果は15です
```

さて，少しずつ解説していきましょう．まずC言語における関数の書き方は次のようになります．

関数の書き方

```
型 関数名（引数）{ 関数の中身 }
```

まずmain関数の前にPlus関数を書かなければならない事に注意してください．なぜそのようにしなければならないかは後で登場するプロトタイプ宣言というところで説明します

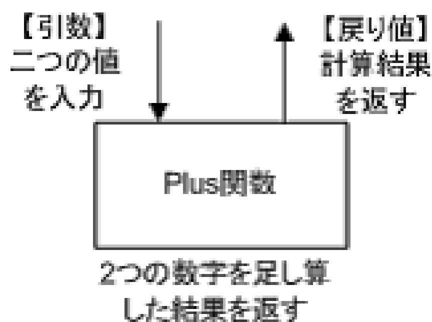

図 4.1: 引数

関数には好きな名前をつけてください．ただし，すでに使われている関数名や変数名は使わないでください．例えば，printfなどの名前をつけないでください．

関数名の前に型とあります．これはその関数が値を返すときの型となります．例えば，Plus関数の場合，main文の中で `n = Plus( hoge , piyo);` となっています．これはPlus関数で処理した内容をnの中に代入する際に，nの型は関数の型と一致していなければなりません．つまりnはint型でなければなりません．

さて，関数を呼び出す際に Plus(hoge, piyo); としました．関数を呼び出す際には， 関数名 (引数) といった感じに記述します．

引数とはどういったものでしょうか．図 4.2 を使ってその説明をします．

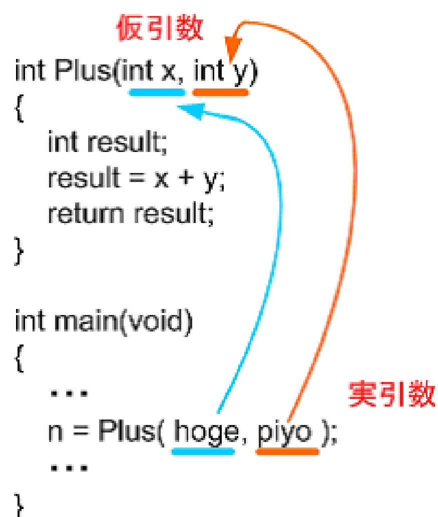

図 4.2: 引数

図 4.2 は引数の様子を示した図です．呼び出した側の引数を実引数といい，呼び出された側の引数を仮引数といいます．このとき，実引数の値が仮引数の値にコピーされます．つまり，

```
void Test(int x)
{
   x = 10;
}

int main(void)
{
  int hoge;
  Test( hoge )

  return 0;
}
```

としても x は hoge のコピーなので，x の中身をいくら変更しても hoge には影響を及ぼしません．また，Test 関数ですが，値を返す必要がないときは関数の型を

void とします．

引数に関して，もしかしたら何も受け取る必要が無い場合があるかもしれません．そういった場合は，引数の中に void，つまり空という宣言を書きます．

```
void Test2(void)
{
    printf("TEST FUNCTION\n");
}

int main(void)
{
    Test2();
    return 0;
}
```

最後に return ですが，Plus 関数の例では result という変数の中身の値を返せという意味です．関数から値を返す場合に使用します．

4.3 プロトタイプ宣言

C 言語で書いたコードは上から下へ実行されます．例として次のようなコードを書いたとします．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          Test2();
6          return 0;
7  }
8
9  void Test2(void)
10 {
11         printf("TEST FUNCTION\n");
12 }
```

このコードをコンパイルするとエラーが発生するでしょう．なぜなら，main 文の中で Test2 関数を呼び出していますが，この時点ではまだ Test2 関数が本当に存在しているかどうかを判断することができません．このような場合に使われるのが，プロトタイプ宣言とよばれるものです．

```
1   #include <stdio.h>
2
3   void Test2(void); /* プロトタイプ宣言 */
4
5   int main(void)
6   {
7           Test2();
8           return 0;
9   }
10
11  void Test2(void)
12  {
13          printf("TEST FUNCTION\n");
14  }
```

プロトタイプ宣言を行ってみました．こんどは main 文より前にあらかじめ Test2 という関数が存在しますよ，ということを知らせるためにこのような書き方をします．Test2() のあとにあるセミコロン (;) に注意してください．

4.4 配列を引数として渡す

引数には配列を渡すこともできます (この説明は正確ではありません．実際は渡せているように見えるだけです)．

次のコードを見てください．

```
1   #include <stdio.h>
2
3   void show(int array[5], int n)
4   {
5           int i;
6           for( i = 0; i < n ; i++ ) {
7                   printf("%d\n", array[i]);
8           }
9   }
10
11  int main()
12  {
13          int array[] = {1, 2, 3, 4, 5};
```

```
14          show( array, 5 );
15          return 0;
16  }
```

このプログラムは array 配列の中身を表示するプログラムです.

実行結果

```
1
2
3
4
5
```

show 関数の引数に array 配列を渡しているように見えます．しかし，show 関数を次のように書き換えても問題ありません.

```
void show(int array[], int n)
{
    int i;
    for( i = 0; i < n ; i++ ) {
        printf("%d\n", array[i]);
    }
}
```

一見 array は配列のように見えますが，実はポインタと呼ばれるものです．ポインタはかなり厄介なものです．次のコードはポインタを用いて show 関数を書き直したものです.

```
void show(int *array, int n)
{
    int i;
    for( i = 0; i < n ; i++ ) {
        printf("%d\n", array[i]);
    }
}
```

いままで出てきた関数はすべて等しいコードです．上記のコードの詳しい説明は今は難しいのでしません．今のうちはこういうコードもかけるんだなということを覚えて置いてください.

4.5 main 関数

ここは余談ですので読み飛ばしてもらってもかまいません．

私たちが一番よく使っている関数に main 関数があります．main 関数はプログラムの一番初めに実行される関数です．と書いておきながら，本当は一番初めではありません．実は main 関数もどこか別のところから呼び出されています．メイン関数が呼び出されている所をみてみましょう．

```
int BP_SYM( … )
{

    …

    result = main(argc,argv,__environ);

    …

    exit(result);
}
```

こんな感じに呼び出されています．これをみてわかるとおり，main は値を返してその値の結果を元に exit という関数を呼び出して処理を終了しています．exit 関数は引数に 0 を渡すと正常終了，0 以外の値を渡すと異常終了としてプログラムを終了させます．このことより，main 文で異常が発生した場合,

```
int main(void)
{
    return 1;
}
```

などとすればよいです．また，よくある間違いですが，main 関数を作る際には

```
void main(void)
{
    /* 何か処理 */
}
```

では駄目です．理由はいままでの説明を読んでいただければわかると思います．しかし最近のコンパイラでは `void main(void)` と書いてもエラーは発生しないようです．

4.6 標準関数

C 言語には標準関数という関数が存在します．これは，よく使われる処理があらかじめ関数として用意されています．例えば printf 関数や scanf 関数などが標準関数にあたります．この printf 関数などは stdio.h というファイルにあらかじめ作られています．そして，`#include <stdio.h>` という部分でこのファイルを読み込む事によって printf 関数を自分で定義しなくても今まで使うことができていました．

ここで，標準で用意されている関数のほんの一部を紹介します．

4.6.1 文字列関係の処理

文字列の処理に用いる関数は，string.h に用意されています．今から紹介する関数を使用するためには，`#include <string.h>` と定義しなければなりません．

strcpy 関数

定義

```
char *strcpy(char* str1, const char* str2);
```

strcpy 関数は文字列をコピーする際に使います．一つ目の引数に二つ目の引数の内容をコピーします．引数に $*$ マークが付いています．これはポインタというものですが，今の段階では気にしないでください．また，const とは引数として渡した変数を関数内で変更することを禁止するキーワードです．

```
1   #include <stdio.h>
2   #include <string.h>
3
4   int main(void)
5   {
6           char source[20] = "hello";
7           char dest[20];
8
9           strcpy( dest , source );
10          printf("コピーされた文字列は %s です\n", dest);
11          return 0;
12  }
```

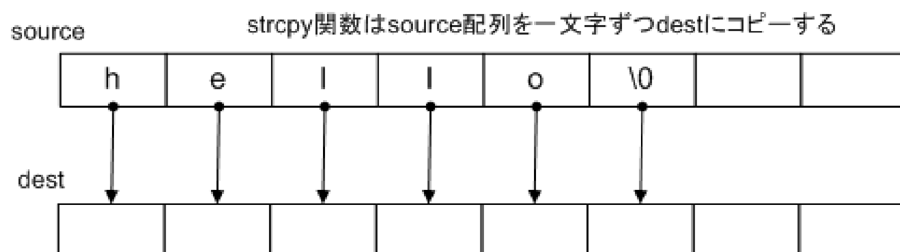

図 4.3: strcpy の動作

実行結果

コピーされた文字列は hello です

strncpy 関数

定義

```
char *strncpy(char* str1, const char* str2, size_t n);
```

一つ目の引数に二つ目の引数の内容を n 文字だけコピーします．ただし，str2 の長さが n 以上のときには n 文字をコピーするが, このときに NULL 文字の自動付加は行われない．str2 の長さが n より少ない場合には,str1 の中の後ろの部分に NULL 文字で埋められる．

要するに文字列の最後に NULL 文字が自動付加されない場合があるということです．strcpy 関数ではこのようなことは起こりませんでしたが，strncpy 関数の場合は自分で NULL 文字を文字列の最後に付け加える必要があるということです．

```
1   #include <stdio.h>
2   #include <string.h>
3
4   int main(void)
5   {
6           char source[20] = "helloworld";
7           char dest[20];
8
9           strncpy( dest , source , 5);
10          dest[5] = '\0'; /* これを忘れないように */
11          printf("コピーされた文字列は %s です\n", dest);
12          return 0;
```

```
13  }
```

実行結果

コピーされた文字列は hello です

strlen 関数

定義

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

strlen 関数は引数 str の文字数を返す関数です．ただし文字列の最後の NULL 文字はカウントに含まれません．size_t というのは unsigned int とほぼ同じです．[1]

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6          char source[20] = "hello";
7          printf("文字列の長さは %d です\n", strlen(source) );
8          return 0;
9  }
```

実行結果

文字列の長さは 5 です

strcat 関数

定義

```
char* strcat(char* str1, const char* str2);
```

strcat 関数は文字列 str1 の後ろに str2 の文字列を連結させる関数です．

```
1  #include <stdio.h>
2  #include <string.h>
3
```

[1]環境依存なので unsigned int とは限りません．

```
4   int main(void)
5   {
6           char source[80] = "world";
7           char dest[80] = "hello";
8
9           strcat(dest, source);
10          printf("文字列：%s\n", dest );
11          return 0;
12  }
```

実行結果

文字列：helloworld

strncat 関数

定義

```
char* strncat(char* str1, const char* str2, size_t n);
```

strncat 関数は文字列 str1 の後ろに str2 の文字列を n 文字分連結させる関数です.

```
1   #include <stdio.h>
2   #include <string.h>
3
4   int main(void)
5   {
6       char dest[20] = "ABC";
7       char source[20] = "DEFGH";
8
9       strncat(dest, source, 3);
10      printf("文字列：%s\n", dest);
11      return 0;
12  }
```

実行結果

文字列：ABCDEF

strcmp 関数

定義

```
int strcmp(const char* str1, const char* str2);
```

strcmp 関数は，str1 と str2 の内容を比較し，その結果を戻り値として返します．

もし，二つの文字列の内容が等しい場合戻り値は 0，str1 より str2 が大きい場合戻り値は負の値を返します．また str1 より str2 が小さい場合は正の値を戻り値に返します．

要するに二つの文字列が等しい場合は 0 を，そうでない場合は 0 以外の数値を返す関数です．

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6          char source[80] = "hello";
7          char dest[80] = "hello";
8
9          if( strcmp(source, dest) == 0 ) {
10                 printf("二つの文字列は等しいです．\n");
11         } else {
12                 printf("二つの文字列は等しくないです．\n");
13         }
14         return 0;
15 }
```

実行結果

```
二つの文字列は等しいです．
```

4.6.2 入出力関係の処理

画面への文字列の出力，及びキーボードからの入力などの処理を行う関数を紹介します．

putchar 関数

定義

```
int putchar(int c);
```

c で指定した 1 文字を表示します．この関数は stdio.h の中に含まれています．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          putchar('a');
6  }
```

実行結果

a

puts 関数

定義

```
int puts (const char *str);
```

str で指定した文字列を表示します．printf と違い%d などの変換指定示を使うことはできません．また，文字列の最後には\n が自動的に付加されます．

この関数は stdio.h の中に含まれています．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          puts("hello world");
6  }
```

実行結果

hello world

getchar 関数

定義

```
int getchar(void);
```

キーボードから一文字受け取ります.
この関数は stdio.h の中に含まれています.

```
1  #include <stdio.h>
2
3  int main()
4  {
5          char ch;
6
7          printf("文字を入力してください：");
8          ch = getchar();
9
10         printf("受け取った文字:%c\n", ch);
11         return 0;
12 }
```

例えば，キーボードから a を入力したとすると実行結果は以下の通りとなります.

実行結果

文字を入力してください：a
受け取った文字列：a

gets 関数

定義

```
char *gets (char *str);
```

キーボードから文字列を受け取ります．scanf 関数とは違い，文字列に空白が含まれていても空白も一緒に文字列として読み込みます.
この関数は stdio.h の中に含まれています.

```
1  #include <stdio.h>
2
3  int main()
```

```
4   {
5           char str[80];
6
7           printf("文字列を入力してください：");
8           gets( str );
9           printf("受け取った文字列:%s\n", str);
10          return 0;
11  }
```

例えば，キーボードから hello world を入力したとすると実行結果は以下の通りとなります.

実行結果

文字列を入力してください：hello world
受け取った文字列：hello world

fgets 関数

定義

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

stream から n - 1 文字分の文字列を受け取り，string に代入します．ここで stream とはファイル操作を扱う部分で登場するものです．stream の部分を stdin とすることで，キーボードから文字列を受け取ることができます.

この関数は stdio.h の中に含まれています.

```
1   #include <stdio.h>
2
3   int main()
4   {
5           char str[80];
6
7           printf("文字列を入力してください：");
8           fgets(str, 6, stdin);
9           printf("文字列:%s\n", str);
10          return 0;
11  }
```

例えば，キーボードから helloworld を入力したとすると実行結果は以下の通りとなります.

実行結果

文字列を入力してください：helloworld
受け取った文字列：hello

4.6.3 数学関係の処理

数学で用いる関数は，math.h に用意されています.

sin 関数

定義

```
double sin( double arg );
```

sin 関数は引数 arg の sin の値を返します．引数の単位はラジアンです.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6          double value = -1.0;
7          printf("%f の sin の値は%f です\n",value ,sin(value));
8
9          return 0;
10 }
```

実行結果

-1.000000 の sin の値は-0.841471 です

cos 関数

定義

```
double cos( double arg );
```

cos 関数は引数 arg の cos の値を返します．引数の単位はラジアンです.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6          double value = -1.0;
7          printf("%f の cos の値は%f です\n",value ,cos(value));
8
9          return 0;
10 }
```

実行結果

-1.000000 の cos の値は 0.540302 です

pow 関数

定義

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

pow 関数は x の y 乗を計算した結果を戻り値として返します.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6          double x = 2.0, y = 5.0;
```

```
7           printf("%f の %f 乗は %f です", x, y, pow(x, y) );
8
9           return 0;
10  }
```

実行結果

2.000000 の 5.000000 乗は 32.000000 です

4.6.4 その他の関数

標準関数のうち，いずれにも分類できない関数をその他の関数としました．

atoi 関数

定義

```
int atoi( const char *string );
```

string で指定した文字列を int 型の数値に変換した結果を戻り値として返します．文字列は整数を表すものでなければなりません．この関数を使うには stdlib.h を読み込まなくてはなりません．

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6     char str[]="150";
7
8     printf("文字列%s を数値%d に変換しました．\n", str, atoi(str) );
9     return 0;
10  }
```

実行結果

文字列 150 を数値 150 に変換しました．

toupper 関数

定義

```
int toupper( int c );
```

c で指定した文字が小文字である場合大文字に変換します．この関数を使うには stdlib.h を読み込まなくてはなりません．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6    char ch = 'a';
7    printf("%c を %c に変換しました\n", ch, toupper(ch) );
8
9    return 0;
10 }
```

実行結果

a を A に変換しました

tolower 関数

定義

```
int tolower( int c );
```

c で指定した文字が大文字である場合小文字に変換します．この関数を使うには stdlib.h を読み込まなくてはなりません．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
```

```
5    {
6      char ch = 'A';
7      printf("%c を %c に変換しました\n", ch, tolower(ch) );
8
9      return 0;
10   }
```

実行結果

A を a に変換しました

4.7 例題

4.7.1 自作 strcpy 関数の作成

仕様

strcpy と同等の機能の関数 my_strcpy を作成せよ.

```
1    #include <stdio.h>
2
3    void my_strcpy(char dest[], char source[])
4    {
5            int i;
6
7            for( i = 0; source[i] != '\0'; i++ ) {
8                    dest[i] = source[i];
9            }
10           dest[i] = '\0';
11   }
12
13   int main()
14   {
15           char source[100] = "hello";
16           char dest[100];
17
18           my_strcpy(dest, source);
19           printf("コピーした文字列は %s です\n", dest);
```

```
20          return 0;
21  }
```

本物の strcpy は戻り値を正しく返さなければなりませんが，現段階ではポインタを学習していないので，戻り値は無しとしました．

実行結果

コピーした文字列は hello です

4.7.2 自作 strlen 関数の作成

仕様

strlen と同等の機能の関数 my_strlen を作成せよ．

```
1   #include <stdio.h>
2
3   int my_strlen(char str[])
4   {
5           int i = 0;
6
7           for( i = 0; str[i] != '\0'; i++ )
8                   ;
9           return i;
10  }
11
12  int main()
13  {
14          char buf[100] = "hello";
15
16          printf("%s の文字数は %d です\n", buf, my_strlen(buf) );
17          return 0;
18  }
```

実行結果

hello の文字数は 5 です

4.7.3 自作 strcat 関数の作成

仕様

strcat と同等の機能の関数 my_strcat を作成せよ.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void my_strcat(char dest[], char source[])
5  {
6          int i;
7          int len = strlen( dest );
8
9          for( i = 0; source[i] != '\0'; i++ ) {
10                 dest[ i + len ] = source[i];
11         }
12         dest[i + len] = '\0';
13 }
14
15 int main(void)
16 {
17         char source[100] = "world";
18         char dest[100] = "hello";
19
20         my_strcat( dest, source );
21         printf("文字列：%s\n", dest );
22         return 0;
23 }
```

実行結果

文字列：helloworld

4.7.4 べき乗を求めるプログラム

仕様

べき乗を求めるプログラムを作成する．例えば 2 の 5 乗を求める関数を作る場合は，Power(2 , 5) となるようにする．

```
1  #include <stdio.h>
2
3  int Power(int x, int y); /* プロトタイプ宣言 */
4
5  int main(void)
6  {
7          int xi,yi;
8
9          printf("整数を入力してください:");
10         scanf("%d",&xi);
11         printf("何乗にしますか:");
12         scanf("%d",&yi);
13
14         printf("%d の%d 乗は%d です\n",xi,yi,Power(xi,yi));
15         return 0;
16 }
17
18 int Power(int x, int y)
19 {
20         int i,result;
21
22         for( i=1, result = 1; i <= y; i++) {
23                 result *= x;
24         }
25         return result;
26 }
```

実行結果

整数を入力してください：2
何乗にしますか：5
2 の 5 乗は 32 です

4.8 演習問題

問題 1

strncpy と同等の機能の関数を作成せよ.

問題 2

strncat と同等の機能の関数を作成せよ.

問題 3

指定した文字列を逆順に表示する関数 reverse を作成せよ.

問題 4

文字列 source から特定の文字 ch を取り除く関数を作成せよ.

問題 5

指定した文字列が回文であるかを判定する関数 palindrome を作成せよ. 回文とは前後どちらから読んでも同じになる語句のことである. radar や madamimadam(= Madam, I'm Adam) などが回文にあたる.

第5章　有効範囲とプリプロセッサ

5.1　有効範囲

5.1.1　変数寿命

次のようなコードを書いたとします．

```
1  #include <stdio.h>
2  void func(void);
3
4  int main(void)
5  {
6          int hoge;
7          （略）
8          return 0;
9  }
10
11 void func(void)
12 {
13         int hoge;
14 }
```

コードを見てわかるように，int 型の hoge という変数を作成しています．hoge という名前が 2 回使われていますが，これは問題ないのでしょうか．ここで出てくる考え方が変数寿命です．例えば，main 関数の中ででてくる hoge は main の終わり，つまり} の場所まで有効です．main 関数の終わりになると，その hoge という変数は消滅します．また，func 関数ででてくる hoge も func 関数の中で有効です．つまり，次のようなことはできません．

```
1  #include <stdio.h>
2  void func(void)
3
4  int main(void)
```

```
5  {
6          int hoge;
7          (略)
8          return 0;
9  }
10 void func(void)
11 {
12         hoge = 10;
13 }
```

hoge は main 関数の中で宣言されているので，main 関数の中でのみ有効です．では，次のコードはどうでしょうか．

```
1  #include <stdio.h>
2
3  int hoge; /* 1個目 */
4
5  void func(void);
6
7  int main(void)
8  {
9          int hoge; /* 2個目 */
10         hoge = 20;
11         (略)
12         return 0;
13 }
14 void func(void)
15 {
16         int x;
17         int hoge = 10; /* 3個目 */
18
19         for(x=0; x < 10 ; x++ ) {
20                 int hoge; /* 4個目 */
21                 hoge = x;
22         }
23 }
```

ここで，1 個目の hoge は関数の外で宣言されています．このようにすると，どこからでも 1 個目の hoge を使うことができます．次に main 関数の中で宣言した

hoge はどうなるでしょうか．ここで宣言された hoge は 1 個目の hoge とは名前は同じですが，違うものです．この 2 個目の hoge は main 関数の中でのみ有効です．ということは，main 関数の中で 1 個目の hoge は使えなくなります．

3 個目の hoge は同様に func 関数の中でのみ有効です．4 個目の hoge は for 文の中でのみ有効です．ただし，次のようなことはできません．

```
int main(void)
{
    int hoge;
    int hoge;

    return 0;
}
```

同じ有効範囲の場所に同じ名前の変数を作ることはできません．

5.1.2 static

次のコードを見てください．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          static int x = 10;
6          (略)
7          return 0;
8  }
```

ここで見慣れない static というものが付いています．この static のことを記憶域クラス指定子といいます．

static を付けて宣言された変数は，プログラム開始時に変数が作られ，プログラムが終了するときに消滅します．次のコードを見てください．

```
1  #include <stdio.h>
2
3  void func(void);
4
5  int main(void)
```

```
6   {
7           func();
8           func();
9           func();
10          return 0;
11  }
12
13  void func(void)
14  {
15          static int hoge = 10;
16          hoge++;
17          printf("hoge = %d\n",hoge);
18  }
```

このコードの実行結果は以下のようになります.

実行結果

```
hoge = 11
hoge = 12
hoge = 13
```

func 関数を通るたびに 10 が初期化されているわけではありません. static 変数では初期化はプログラム開始の前に一度行われるだけです.

5.2 プリプロセッサ

5.2.1 #define

#define とは, 何らかの文字列を何か別の文字列に置き換える時に使うものです.

書式

```
#define 文字列A 文字列B
```

#define は, A を B に置き換えるという役割を果たします.

```
1   #include <stdio.h>
2   #define NUMBER 10
3
4   int main(void)
```

```
5  {
6          printf("%d",NUMBER);
7          return 0;
8  }
```

上記のコードは，NUMBER という文字列を 10 という定数に置きかえるという意味です．NUMBER と全部大文字の名前をつけましたが，#define を使って作った文字列に名前をつけるときには，全部大文字にするようにしましょう．

また，よくある間違いですが，#define で定義したあと，セミコロンをつけないようにしましょう．そのセミコロンまで置き換えられてしまいます．

5.2.2 #undef

#undef とは，一度定義したマクロを無効にしたい時に使います．

書式

#undef 文字列 A

たとえば，次のようにして使います．

```
#undef NUMBER
#define NUMBER 11
```

これは，もし#undef 以前に NUMBER というマクロがすでに存在していた場合，今までのマクロを無効にして，新しく 11 という数字で置き換えろという意味です．

5.2.3 引数付マクロ

define のようなマクロ定義においても，関数と同じように引数を持つことができます．

```
1  #include <stdio.h>
2
3  #define SUM( x, y ) ( (x) + (y) )
4
5  int main(void)
6  {
7          int sum;
8
```

```
 9          sum = SUM(3,5);
10          printf("%d",sum);
11
12          return 0;
13  }
```

これは， `SUM(3,5)` の部分を ((3) + (5)) に置き換えるという意味です.
引数付きマクロは使い方に注意しないと，予期せぬ結果がでてくることがあります．例えば次のような場合です.

```
#define SQR(x) x*x
```

このマクロは，ある値を引数として受け取り，その二乗の値に置き換える役割を果たします．では，このマクロを使ってみましょう.

```
int x = SQR( a + b );
int y = SQR( a ) / SQR( b );
```

a と b には何か値が入っているとします．ここで，このマクロを展開してみると，次のようになります.

```
int x = a+b * a+b;
int y = a*a / b*b;
```

これではおかしな事になってしまいます．例えば，a が 3 で b が 4 だとしましょう．x の結果は本来，7 の二乗の 49 にならないといけません．しかし，3+4*3+4 は 3+12+4 となり 19 となってしまいます．また，y の値もおかしな事になってしまいます．これを回避するには，どのようにしたらよいでしょうか.
答えは次のようになります.

```
#define SQR(x) ((x)*(x))
```

このようにカッコをつける事によって回避することができます.

5.2.4 #include

#include という命令は，指定したファイルを取り込むということです．たとえば，いままで printf 関数などの関数を使ってきましたが，printf 関数はどこで定義されているでしょうか．printf 関数は stdio.h という名前のファイルに宣言されています．この stdio.h というファイルを取り込むことによって，printf や scanf など

の関数を自分で作らなくても使うことができます．このように，標準で用意されている関数のことを，標準関数と呼びます．

使い方

```
#include <ファイル名>
#include "ファイル名"
```

#include には 2 通りの宣言の仕方があります．まず，はじめの< >で囲む宣言の仕方ですが，これは標準で用意されているヘッダファイルを読み込む時に使います．” ”で囲む宣言は，自分で作ったファイルを読み込む時に使います．ヘッダファイルは自分で作ることができます．

5.3 演習問題

問題 1

次のプログラムの実行結果を予測せよ．

```
1   #include <stdio.h>
2
3   int num = 10;
4
5   void func(int number)
6   {
7           printf("NUM = %d\n", num);
8           number = 20;
9   }
10
11  int main(void)
12  {
13          int num = 3;
14
15          printf("NUM = %d\n", num);
16          func( num );
17          printf("NUM = %d\n", num);
18          return 0;
19  }
```

問題 2

次のプログラムの実行結果を予測せよ.
また, このプログラムの問題点を指摘し改善せよ.

```
1   #include <stdio.h>
2
3   #define TIMES( x, y ) ( x * y )
4
5   int main(void)
6   {
7           int num;
8
9           num = TIMES( 3 + 5 , 5 + 3 );
10          printf("%d\n",num);
11
12          return 0;
13  }
```

第6章　ポインタ

C 言語においてポインタは重要な項目であり，難しい項目の 1 つでもあります．ポインタが難しい原因の一つは，コンピュータのメモリについて意識しなければならない事にあると思われます．この章では，ポインタの基礎的な事について説明していきます．

6.1　ポインタとは

ポインタとは，メモリのアドレスを保存する変数です．

メモリとはなんでしょうか．例えば，ある変数を家と例えると，メモリは土地のようなものです．あなたが今，家を建てたいとします．家には大切なデータを保管することができます．また，家には名前を付けることができます．家の名前は hoge とでもしましょう．家を建てるには当然ながら土地が必要です．適当に空いている土地を探して，そこに建てることとしましょう．お金はどうするんだとか，住所は勝手に決めて問題無いのかとか，そういった事は気にする必要はありません．とにかく土地が空いていれば家は自由に建てられるのです．そういうことにしましょう．また，あなたが建てた家には住所が付いています．この住所は，他の人と情報をやり取りする際に利用します．

C 言語において，変数は上記の例でいう家と同じです．例えば，int 型の変数 hoge を次のように定義したとします．

```
int hoge;
```

hoge は int 型の値を保持する変数です．この宣言を実行すると，コンピュータはメモリ上に hoge を作成します．この hoge には当然住所が存在しています．この住所はプログラム側が勝手に割り当てます．

ここでパソコンのメモリ内の様子を見てみましょう．図 6.1 をご覧下さい．メモリという土地に変数 hoge が作成されているようです．hoge には住所が付いています．図では 100 番地となっているようです．

さて，この住所情報を保存することができる変数がポインタです．ここで，仮に `p_piyo` という名前のポインタを作成してみます．`p_piyo` は hoge の住所情報を持っているとしましょう（図 6.6）．当然ながら，ポインタ変数にも住所はあります．図では 150 番地となっているようです．

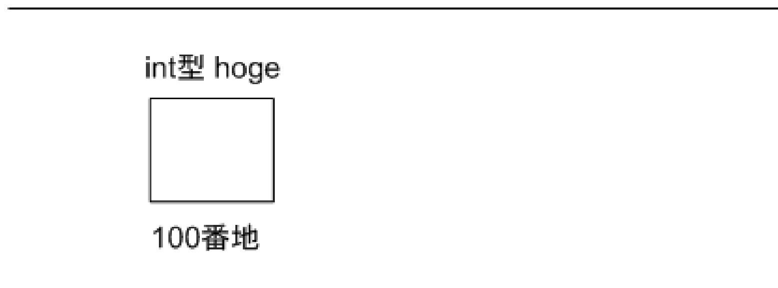

図 6.1: メモリ内部の様子

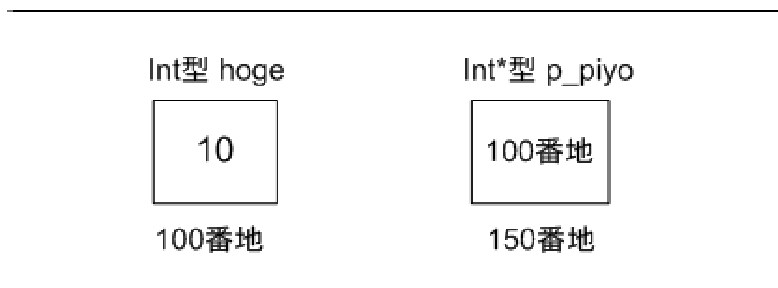

図 6.2: hoge の住所情報を持った p_piyo

ポインタは少し変わっています．ポインタは住所を入れることができる箱（変数）なわけですが，実はすべての変数の住所を入れられるわけではないのです．例えば，int 型の変数の住所を保管したければ，int 型のポインタを作成する必要があります．double 型の変数の住所を保管したければ，double 型のポインタを利用しなければならないわけです．double 型のポインタには int 型の変数の住所を保管しておくことはできないのです．

ポインタってのは住所を入れておくことができる変数なわけだから，別に型をわざわざ指定する必要はないのではないか，と思われる方もいるかもしれませんが，ポインタに型が無いと，後々面倒な問題が発生するわけです．いまの段階では，ポインタには型があるということだけを覚えて置いてください．

int 型のポインタ p_piyo を宣言する場合は，

```
int* p_piyo;
```

または

```
int *p_piyo;
```

と書いたりします．どちらで書いても同じ意味です．ただ，前者を使う場合には複数のポインタ変数を宣言するときに少し注意が必要です．たとえば後者のように，

```
int *p_hoge,*p_piyo
```

と宣言すると，p_hoge，p_piyo はどちらもポインタです．しかし，

```
int* p_hoge, p_piyo
```

とすると，p_hoge はポインタですが，p_piyo はポインタ型ではなく，ただの int 型になってしまいますので注意してください．

6.2 ポインタへの値の代入

```
int hoge;
```

という定義を書いてみました．これは，int 型の変数を作るということです．ここで，パソコンのメモリの様子を図 6.1 に表してみます．図 6.1 は，パソコンのメモリ内部に int 型の hoge という名前の箱（変数）が作られたということですね．そして，その箱が作られた場所の住所（アドレス）は 100 番地です．

では次に `int* p_piyo` という変数を作った様子を図 6.3 に表します．

さて，p_piyo という箱（ポインタ）を作ったので，次はその箱に値を代入してみましょう．次のようなコードを書いたとします．

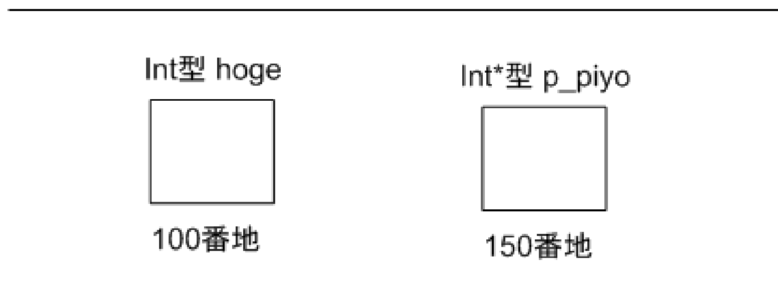

図 6.3: メモリ内部の様子

```
int hoge;
int* p_piyo;
hoge = 10;
p_piyo = &hoge;
```

ここで，int 型の hoge に代入できる値は int(整数）だけです．hoge に値を代入するときは， `hoge = 10;` と記述します．これは，hoge に 10 を代入するということですね．同様に，p_piyo に値を代入してみましょう．ここで， `p_piyo = hoge;` とするとどうなるでしょうか（図 6.4）．もちろんコンパイルエラーです．p_piyo は int*型です．int*型には (int 型の）アドレスしか入れることができません (値を代入するときは，代入する側とされる側の型が同じでなければなりません）．

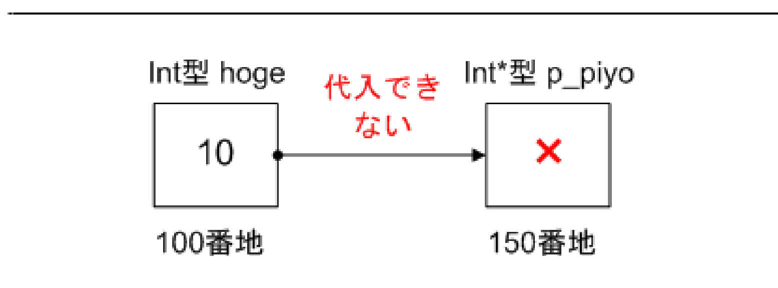

図 6.4: メモリ内部の様子

では，p_piyo に hoge の住所（アドレス）を代入するにはどうしたらよいでしょうか．ここででてくるのが&という記号です．ここで，`p_piyo = &hoge` とすると，図 6.5 のようになります．

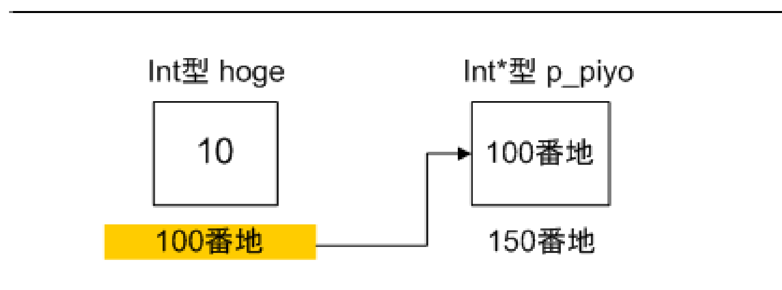

図 6.5: メモリ内部の様子

&は，指定した変数の住所を取得するためのものです．つまり，&hoge とは hoge の住所を取得せよ，ということです．

6.3 ポインタによる変数値の参照

前節では，変数に値を代入することを考えました．今度は値を表示することを考えて見ましょう．次のようなコードを書いたとします．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int hoge;
6      int* p_piyo;
7
8      hoge = 10;
9      p_piyo = &hoge;
10
11     printf("hoge address :%p\n", &hoge);
12     printf("hoge :%d\n", hoge);
13     printf("p_piyo :%p\n", p_piyo);
14     printf("*p_piyo:%d\n", *p_piyo);
15     printf("p_piyo address :%p\n", &p_piyo);
16 }
```

ここで，printf 関数を用いてアドレスを表示する時には%p という記号を使います．上の状態を図に表すと，図 6.6 のようになります．

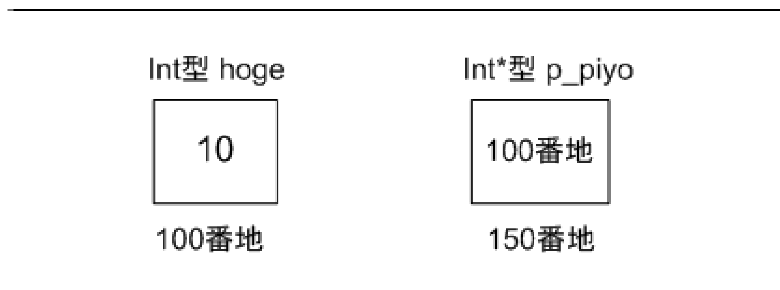

図 6.6: メモリ内部の様子

printf("hoge address :%p\n",&hoge); の部分は前節で説明した&を使っています．指定した変数の住所を取得せよという意味でしたね．よって，実行結果は

```
hoge address : 100番地
```

となります．[1]

printf("hoge :%d\n",hoge); は，hoge の中身が表示されますね．
printf("p_piyo :%p\n",p_piyo); も，p_piyo の中身が表示されます．

さて，ここからが本題です．printf("*p_piyo:%d\n",*p_piyo); という所です．ここで出てきた*p_piyo とはなんでしょうか．この部分を実行してみると以下のようになります．

```
*p_piyo : 10
```

これは，hoge の中身と同じですね．つまり*p_piyo とは，p_piyo が持っているアドレスにあるデータを持ってきます．もう少し詳しく説明しましょう．現在 p_piyo は 100 番地というデータを持っています．この，100 番地にあるデータ（この場合は hoge の中身）を取得したい場合，*p_piyo とするのです．メモリ内部の様子の図をよく眺めて考えてみてください．

ここで，ポインタを宣言するときにも*記号を使い，ポインタが持っている住所のデータを取得する場合にも*記号を使いましたが，両者は**全くの別物**です．次のコードをご覧下さい．

```
int* p_hoge; /* ポインタを宣言するための*記号 */
```

[1] 実際は 100 番地なんて表示されません．まず，変数がメモリ上のどこに割り当てられるかは OS が決めることであり，自分で決めることは出来ません．つまり，住所（アドレス）は勝手に割り当てられます．そして，〜番地のような文字もでてきません．ここでは分かりやすさを追求するために勝手に書き加えたものです．私の家でこのコードを実行したら，次のようになりました．hoge address :0012F580

```
int piyo = 10;

p_hoge = &piyo;

/* ポインタ p_hoge を通して piyo にアクセスするための*記号 */
printf("%d", *p_hoge);
```

ポインタ宣言時の*と，データアクセス時の*は別物です．このように，同じ記号なのに，使う場所によって意味が違ってくるのがポインタをややこしくしている原因でしょうか．

さて，話を戻して最後の， `printf("p_piyo address :%p\n", &p_piyo);` は，どのような実行結果となるでしょうか．分かりますよね．では，コード全体の実行結果を書きます．

実行結果

```
hoge address : 100 番地
hoge    : 10
p_piyo : 100 番地
*p_piyo : 10
p_piyo address : 150 番地
```

ここで，ポインタに型がある理由の一部をお話します．もし，どんな変数のアドレスも入れることができるポインタが存在したらどうでしょうか．C 言語には，そのようなポインタである void*型が存在しています．ここで，void*型だけを使ってプログラムを書いてみる事にしましょう．

```
1  int main(void)
2  {
3      int hoge = 10;
4      void* p_piyo = &hoge;
5
6      /* bar = *p_piyo はできない */
7      int bar = *( (int*)p_piyo );
8
9      return 0;
10 }
```

変数に値を代入する場合は，代入する側とされる側の値が同じでなければなりません．ここで，

```
int bar = *p_piyo;
```

と書いても，p_piyo がもっているアドレスにあるデータ (つまり hoge) の型はどんなものかが分からないわけです．int*型は int 型の変数のアドレスを持っていることをあらかじめコンパイラに知らせておけば，このような心配をする必要がなくなるわけです．

ポインタはあるオブジェクトのアドレスを入れるものだと §6.1 で述べました．さて，なぜアドレスを入れる箱のことをポインタ（矢印）というのでしょうか．前節のコードの様子を図 6.6 で示しました．ここで，図 6.6 は図 6.7 のように書き直すこともできます．p_piyo はアドレスを入れる箱ですが，これは hoge の場所を矢印で指し示していると考えることもできます．この矢印の先にあるデータにアクセスしたいときには，*p_piyo とします．これが，アドレスを入れる箱がポインタ（矢印）と呼ばれている理由です．

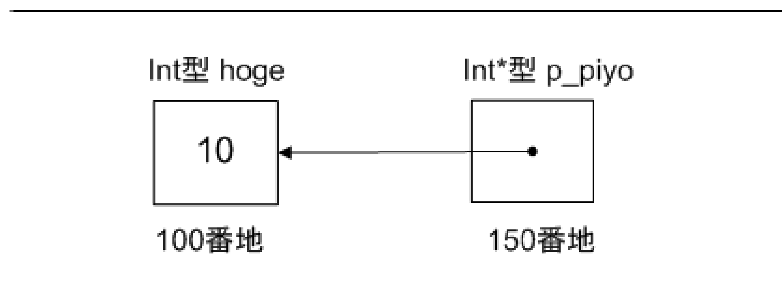

図 6.7: メモリ内部の様子

6.4 NULL ポインタ

int hoge; と定義して， printf("%d",hoge); とした場合はどうなるでしょうか．

答えは不定です．hoge を作った段階では中にどんな値が入っているかはわかりません．

同様に，int* p とした場合も，p の中に入っている値は不定です．しかし， p がどこも指していないという状態を作りたいとしたら，どのようにしたらよいでしょ

うか．そのような状態を作り出したい場合は， `p = NULL;` と p に NULL を代入します．これは `p` はどこも指していないという事を表しています．

この `NULL` [2]とはなんでしょうか． `NULL` は大抵は，

```
#define NULL 0
```

と定義されています．つまり， `p = NULL` は `p = 0` ということになります．だからといって， `p = 0` としてはいけません．もしかしたら

```
#define NULL ( (void*)0 )
```

と定義されているかもしれません．とりあえず，何も指していない場合は `NULL` を代入します．また， `int* p = NULL` として， `printf("%d",*p);` としてはいけません．これは， `p` がどこも指していないのに参照しようとしたのですから，当然といえば当然です．[3]

さて，int*型には数値を代入するとコンパイルエラーになると前に説明しました．例えば，次のようなコードを書くことはできません．

```
int* p_piyo = 10;
```

では，NULL が仮に 0 と定義されているのなら，次のようなコードを書くことはできないのではないかと疑問に思われる方もいるかもしれません．

```
int* p_piyo = 0;
```

しかし，これはコンパイルエラーとはなりません．C 言語において，0 は特別な意味を持っています．ポインタに 0 を代入した場合，コンパイル時に NULL ポインターに変換されます[4]．

6.5 引数としてのポインタ

ここではポインタはなぜ存在するのか，またどのようにして使ったらよいかを解説していきたいと思います．次のような swap 関数を作ったとします．

```
1   void swap( int x, int y )
2   {
3       int tmp = x;
```

[2]NULL はナル，またはヌルと読みます．

[3]余談ですが，Java 言語ではこのように何も指していない場所を参照したとき，NullPointerException という例外（エラー）が発生します．

[4]正直いって，0 が NULL ポインタに変換されるといった仕様はよくないと思います．JAVA 言語では，null という新しいキーワードが用意されています．

```
4       x = y;
5       y = tmp;
6   }
```

このコードでは，二つの変数を引数として受け取り，その中身を交換する関数を作った「つもり」です．次のようなコードを書き，実行させてみましょう．

```
1   #include <stdio.h>
2
3   void swap( int x, int y )
4   {
5       int tmp = x;
6       x = y;
7       y = tmp;
8   }
9
10  int main(void)
11  {
12      int hoge = 10, piyo = 5;
13
14      swap( hoge , piyo );
15      printf("hoge : %d, piyo : %d\n" ,hoge, piyo);
16      return 0;
17  }
```

実行結果

```
hoge : 10, piyo : 5
```

この swap 関数ではうまく値を交換できていません．なぜでしょうか.

ここで，関数の引数について思い出しましょう．§4.2 で，仮引数には実引数のコピーが渡されると説明しました．忘れている人は復習しておきましょう．

これらのコードの実行結果を図で表すと，図 6.8 となります．

仮引数である x と y をいくら書き換えても元の変数 hoge と piyo にはなんら影響を及ぼしません．しかしそれでは困ります．そこで登場するのがポインタです．ポインタを使ってもとの変数にアクセスすることで中身を書き換えることができます．次のようなコードを書いてみます．

```
1   #include <stdio.h>
2
```

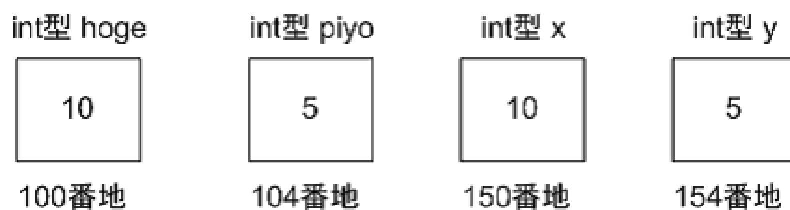

図 6.8: メモリ内部の様子

```
3   void swap(int* x, int* y); /* プロトタイプ宣言 */
4
5   int main(void)
6   {
7       int hoge = 10, piyo = 5;
8
9       swap( &hoge, &piyo );
10      printf("hoge : %d, piyo : %d\n",hoge,piyo);
11      return 0;
12  }
13
14  void swap(int* x, int* y)
15  {
16      int tmp = *x;
17      *x = *y;
18      *y = tmp;
19  }
```

このコードの様子を図で表したものが，図 6.9 です．

ここで，*x は hoge そのものであり，*y も piyo そのものです．よってこれらの値を入れ替えることで値を交換できることになります．実行結果は次のようになります．

実行結果

```
hoge : 5, piyo : 10
```

このように関数に hoge や piyo などの変数を渡し中身を変更したい場合，`&hoge` のようにしてアドレスを渡さなければいけないことが分かったと思います．scanf

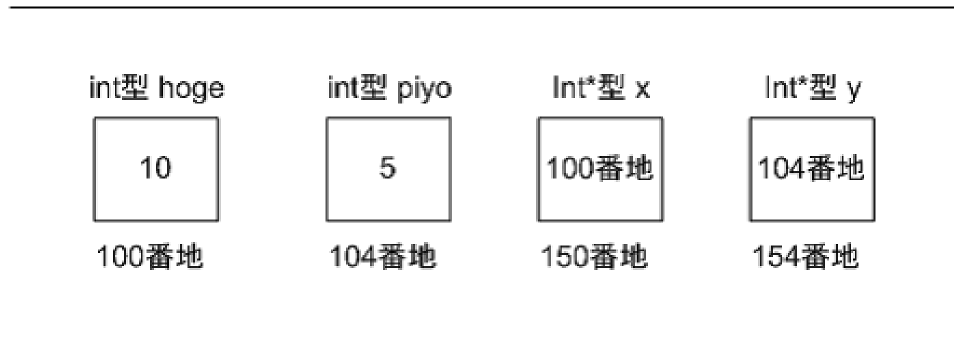

図 6.9: メモリ内部の様子

関数にいままで&を付けていたのはこの為です.

6.6 ポインタの型

ここでは，変数の型の大きさについて学んでいきます.

変数には型が存在するということはすでに勉強したと思います．例えば，int 型や char 型，double 型などが変数の型にあたります．2 章で紹介した表 6.1 を思い出してください.

表 6.1: 変数の型一覧（処理系依存）

型の名前	範囲
short	-32768 〜 +32767
int	-2147483648 〜 +2147483647
long	-2147483648 〜 +2147483647
unsigned short	0 〜 +65535
unsigned int	0 〜 +4294967295
unsigned long	0 〜 +4294967295
char	-128 〜 +127 又は 0 〜 +255
unsigned char	0 〜 +255
float	3.4 * 10^-38 〜 3.4 * 10^+38
double	1.7 * 10^-308 〜 1.7 * 10^+308

表をみてわかるように，int 型は-2147483648 〜 +2147483647 の値を，char 型は-128 〜 127 または 0 〜 255 となっています．この値はどのようにして決められているのでしょうか．

これらの型にはそれぞれ大きさが決まっています．char 型の場合は 1 バイト，int 型の場合は 4 バイトとなっています[5]．1 バイトとは 8 ビットのことです．そして 8 ビットを 2 進法で表すと 8 桁の値を持つことになります．

さて，8 桁で表すことができる値の範囲は,

00000000〜11111111

です．これを符号無しの char 型と考え，上記の数字を 10 進法に変換すると，

0〜255

となるわけです[6]．以上の結果より，char 型（1 バイト）で表すことのできる値の範囲は 0 〜255 と 256 通りの値を表すことができるとわかったと思います．

6.6.1 sizeof

ある型が何バイトであるかを取得する事ができます．それが sizeof です．sizeof は関数ではありません．一見関数のようにも見えますが，**int や return と同じ予約語**です．

次のようなコードを書いてみます．

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char hoge;
5      int piyo;
6
7      printf("char 型のサイズは%d です", sizeof(hoge) );
8      printf("int 型のサイズは%d です", sizeof(piyo) );
9      return 0;
10 }
```

このようにして，調べたい型の大きさを調べることができます．ちなみに実行結果は次のようになりました．[7]

[5]これらの値は環境によって異なる場合があります．
[6]ここでは話がややこしくなるので，-128〜+127 の場合は考えないこととします．
[7]環境によっては結果が違うかもしれません

実行結果

```
char 型のサイズは 1 です
int 型のサイズは 4 です
```

6.6.2 ポインタの型の大きさ

では次にポインタの型の大きさをしらべてみましょう.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char* p_hoge;
5      int* p_piyo;
6
7      printf("char*型のサイズは%dです\n", sizeof(p_hoge) );
8      printf("int*型のサイズは%dです\n", sizeof(p_piyo) );
9      return 0;
10 }
```

実行結果は次のようになりました.

実行結果

```
char*型のサイズは 4 です
int*型のサイズは 4 です
```

さて，この結果をみると，どうやらポインタの型の大きさは 4 バイトのようです.

ここでちょっと考えてみてください．ポインタとは住所（アドレス）を入れる箱でした．つまり，アドレスを入れるものなのだから，ポインタに型[8]はいらないんじゃないかという疑問を持たれた方もいるかもしれません．しかし，ポインタの型は重要な役割を果たしています.

例えば次のように配列を作ったとしましょう.

```
1  #include <stdio.h>
2  int main(void)
3  {
4          int array[5] = {5,10,15,20,25};
5          int* p;
```

[8]例えば int*や char*など

```
6           p = &array[0];
7
8           printf("p[0] = %d\n",*p);
9           p++;
10          printf("p[1] = %d\n",*p);
11  }
```

main 関数のはじめの 3 行を図に表してみると図 6.10 のようになります．つまり，図 6.11 のようになります．

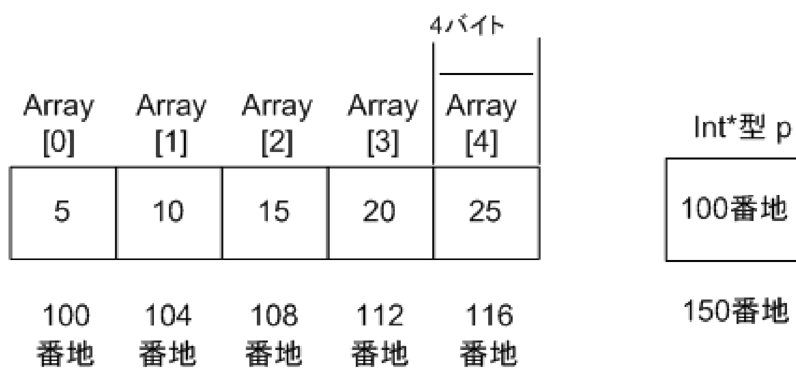

図 6.10: メモリ内部の様子

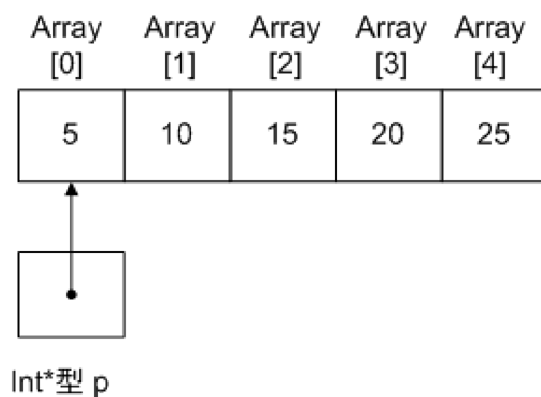

図 6.11: メモリ内部の様子

配列の場合，住所は連続して振り分けられます．例えば，Array[0] が 100 番地に割り当てられたとしましょう．int 型の大きさを 4 バイトとすると，Array[1] は 104 番地に割り当てられます．

ここで，

```
p++;
```

とすると，どうなるでしょうか．p に 1 を足すということだから，100 番地 + 1 番地 で 101 番地？と考える人がいるかもしれませんが，そうはなりません．この場合，int 型のサイズ（この場合は 4 バイト）だけポインタに足されます．つまり，100 + 4 番地で 104 番地ということになります．この，p++; を図で表してみると，図 6.12 という感じになります．

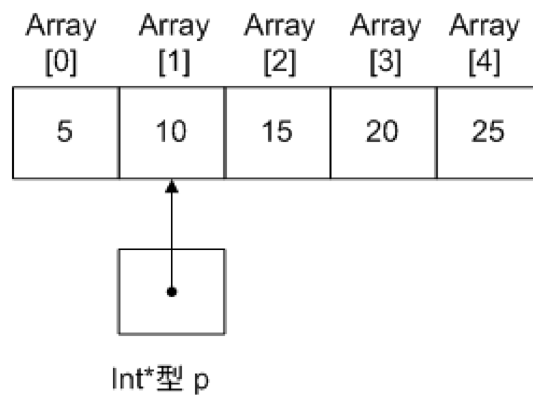

図 6.12: メモリ内部の様子

つまり，ポインタをインクリメントすると，次の要素を指し示すわけですね．この 4 バイト足すという情報は int*型だったからわかった情報です．もし int*ではなく char*としたら，char 型は 1 バイトなので，100 番地+1 番地=101 番地となってしまって，おかしな事になってしまいます．ですので，ポインタの型も重要な役割を果たしています．では一応プログラムの実行結果を書いておきます．

実行結果

```
p[0] = 5
p[1] = 10
```

6.7 配列とポインタ

配列とポインタは全く別の概念ですが，定義がよく似ているので混乱しやすい所です．ここでは，配列とポインタの違いについて解説していきます．

6.7.1 配列とポインタ入門

次のようなコードを書いたとします．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int array[5] = { 1, 2, 3, 4, 5 };
6       int* p;
7
8       p = &array[0];
9
10      printf("*(p+0) : %d\n", *(p + 0) );
11      printf("*(p+1) : %d\n", *(p + 1) );
12      printf("*(p+3) : %d\n", *(p + 3) );
13      return 0;
14  }
```

では，まず

```
int array[5] = { 1, 2, 3, 4, 5 };
int* p;
p = &array[0];
```

までの内容を図 6.13 に示します．

配列の場合，住所は連続して振り分けられます．例えば，array[0] が 100 番地に割り当てられたとしましょう．int 型の大きさを 4 バイトとすると，array[1] は 104 番地に割り当てられます．array は array[0]〜array[4] の合計 5 つの変数を作ったので，array の最後の要素 array[4] は 116 番地にありますね．

int*型である p に値を代入してみましょう．

```
p = array[0];
```

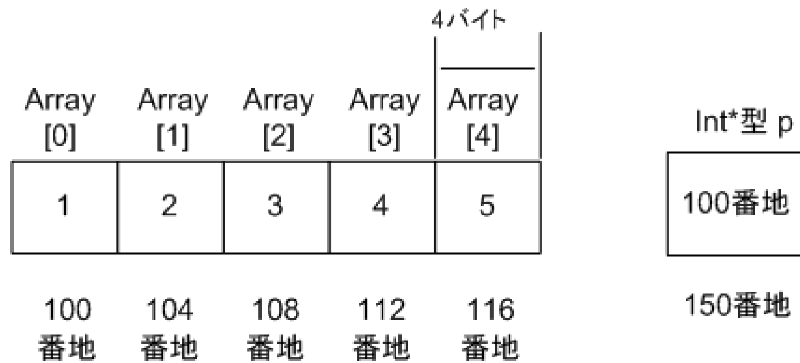

図 6.13: メモリ内部の様子

というのはダメでしたよね．array[0] は int 型です．そして p は int*型です．int*型には整数 (int) は代入できないです．しかし，int*型は int へのアドレス値を保持することができます．よって，array[0] のアドレスを代入するためには,

```
p = &array[0];
```

と&をつけなければなりませんでした．忘れていたらしっかり復習しておいてください．

また，例えば array[1] へのアドレスを p へ代入したい場合は,

```
p = &array[1];
```

とすればよいわけです．

では次に

```
printf("*(p+0) : %d\n", *(p+0) );
printf("*(p+1) : %d\n", *(p+1) );
printf("*(p+3) : %d\n", *(p+3) );
```

の部分の説明に入ります．

`*p` とはなんでしょうか．p は現在 100 番地という値を持っています．ここで，`*p` とは 100 番地にあるデータという意味です．つまり，array[0] と同じです．この部分もしっかり思い出しておいてください．

さて，`*(p + 0)` は `*(p)` と同じ，つまり `*p` と全く同じです．この p は配列の中の 1 番目の要素を指しているという意味です．図 6.13 を書き直すと図 6.14 のようになります．

よって,

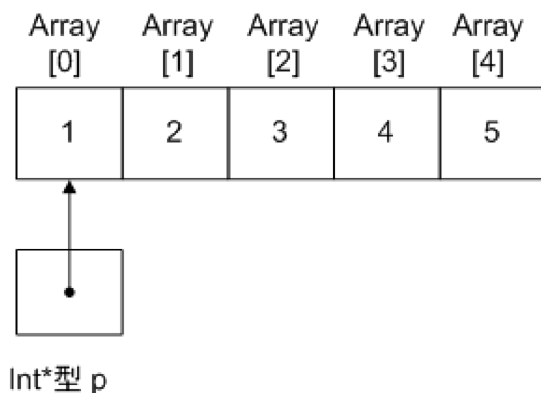

図 6.14: メモリ内部の様子

```
printf("*(p+0) : %d\n", *(p+0) );
```

の出力は array[0] の値である 1 が表示されるはずです．
次の

```
printf("*(p+0) : %d\n", *(p+1) );
```

の部分はどうなるでしょうか．p の中身は 100 番地です．ここに 1 を足すのだから 101 番地となるでしょうか．いいえ，そうはなりません．
int 型の size は 4 バイト（違う場合もある）だと説明しました．ここで p+1 とすると，int 型の大きさ（ここでは 4 バイト）だけ p の番地に足します．つまり，p+1 をすると，p の中身の 100 番地に int 型の大きさの 4 を足して，104 番地となります．
整理しますと，`*(p+0)` の部分は，`*(100 番地)` です．`*(p+1)` は `*(100 番地+4 番地)` つまり `*(104 番地)` となり，配列の 2 番目の値 `array[1]` を指し示すことを意味します．
最後の

```
printf("*(p+0) : %d\n", *(p+3) );
```

の部分の出力はもう分かりますね．

実行結果

*(p+0) : 1
*(p+1) : 2
*(p+3) : 4

読者の中には次のような疑問を持った方もいるかもしれません．

```
*( p + 1 );
```

となぜ p + 1 の部分にカッコをつける必要があるのかと．

ここで次のようなコードを書いてみます．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int array[5] = { 1, 5, 10, 15, 20 };
6      int* p;
7
8      p = &array[0];
9      printf("array[0] : %d\n", *(p + 0) );
10     printf("array[1] : %d\n", *(p + 1) );
11
12     printf("*p+1 : %d\n", *p + 1 );
13
14     return 0;
15 }
```

このコードの実行結果は以下の通りです．

実行結果

```
array[0] : 1
array[1] : 5
*p+1 : 2
```

`*(p + 1)` と `*p + 1` では違う結果が出てしまいましたね．ここで小学校の算数を思い出してください．$3+5*9$ の計算結果はなんでしょうか．答えは 48 です．式の演算には優先順位というものがありました．この場合，+より*の方が優先順位が高いため $5*9$ の計算が先に行われ，最後に 3 を足すんでしたね．もし，$3+5$ の計算をしてから 9 をかけたい場合は，$(3+5)*9$ とすればよいわけです．

C 言語の場合も同じです． `*p + 1` の場合は * と + では * の方が優先順位が高いため，先に `*p` の計算が行われます．そして，*p の結果に 1 を足す，つまり $1+1$ となり答えは 2 となってしまいます．よって，

```
printf("*p+1 : %d\n", *p + 1 );
```

の結果は 2 です．

よって p が保持しているアドレスに 1 を足したい場合は，

```
printf("*p+1 : %d\n", *(p + 1) );
```

とカッコをつけなければなりません．
さて，話を戻します．最初の例を思い出してください．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int array[5] = { 1, 2, 3, 4, 5 };
6       int* p;
7
8       p = &array[0];
9
10      printf("*(p+0) : %d\n", *(p+0) );
11      printf("*(p+1) : %d\n", *(p+1) );
12      printf("*(p+3) : %d\n", *(p+3) );
13      return 0;
14  }
```

ここで，厄介なことに

```
p = &array[0];
```

の部分は実は，

```
p = array;
```

と書いてもよい事になっています．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int array[5] = { 1, 2, 3, 4, 5 };
6       int* p;
7
8       p = array; /* p に array の先頭のアドレスを代入する */
9
10      printf("*(p+0) : %d\n", *(p+0) );
11      printf("*(p+1) : %d\n", *(p+1) );
```

```
12        printf("*(p+3) : %d\n", *(p+3) );
13        return 0;
14    }
```

この先頭のアドレスを代入するというコードはよく使うため，このような省略形が生まれました．array 全体の要素を p へ代入する，という意味**ではない**ので注意してください．

6.7.2 配列とポインタ

ここからが配列とポインタがごちゃごちゃになるかもしれない部分です．次のようなコードを書いたとします．

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        int array[5] = { 1, 2, 3, 4, 5 };
6        int* p;
7
8        p = array; /* p に array の先頭のアドレスを代入する */
9
10       printf("p[0]   : %d\n", p[0] );
11       printf("p[1]   : %d\n", p[1] );
12       printf("*(p+1) : %d\n", *(p + 1) );
13       return 0;
14   }
```

実行結果は以下のようになります．

実行結果

```
p[0] : 1
p[1] : 2
*(p+1) : 2
```

`p[0]` とは一見配列のように見えますが，**これは配列ではありません**．そもそも p は配列ではなく array へのポインタでした．この `p[0]` という部分は実は `*(p + 0)` と全く同じ意味です．よって， `p[1]` という部分は `*(p + 1)` と同じです．この `*(p + 1)` といったような操作はよく行われるのですが，いちいちこのように書くのは面倒だということで， `p[1]` のような省略形が誕生しました．

では，次のようなコードを書いてみます.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int array[5] = { 1, 2, 3, 4, 5 };
6      int* p;
7
8      p = &array[1]; /* array[1] のアドレスを代入する */
9
10     printf("p[0] : %d\n", p[0] );
11     printf("p[1] : %d\n", p[1] );
12     printf("p[-1]: %d\n", p[-1] );
13     return 0;
14 }
```

このプログラムの実行結果は次のようになります.

実行結果

```
p[0] : 2
p[1] : 3
p[-1]: 1
```

最後の p[-1] の部分に注目してください．このような操作は配列では**絶対にできないこと**です．つまり， `array[-1]` なんて操作は絶対にできないのです． `p[-1]` の部分は `*(p - 1)` と全く同じなので，このようなコードを書くことができます.

また，余談になりますが， `*(p + 1)` は `*(1 + p)` と同じです．算数の計算の時に $(3+5)$ と $(5+3)$ の計算結果が一緒なのと同じですね.

さて， `*(1 + p)` ということは `1[p]` という風に書くこともできます．しかし，**このような書き方はしないようにしてください**.

6.7.3 引数に配列へのポインタを渡す

いままで引数には `int i;` などの変数を渡してきました．しかし，次のようなことも出来ます.

```
1  #include <stdio.h>
2
3  void myPuts(char str[20])
```

```
4   {
5       int i;
6       for( i = 0; str[i] != '\0'; i++ ) {
7           putchar( str[i] );
8       }
9       printf("\n");
10  }
11
12  int main(void)
13  {
14      char str[20] = "Hello world";
15      myPuts( str );
16      return 0;
17  }
```

実行結果

```
Hello world
```

このコードは一見配列を引数として渡しているように見えます．しかし，残念ながら **myPuts の引数である str は配列ではありません**．これは実はポインタなのです．

紛らわしいことに，**仮引数である**char str[20] は `char* str` と**全く同じもの**です．もちろん，main 関数内ではこれら 2 つは同じ意味ではありません．例えば次のように書いたとしましょう．

```
int main(void)
{
  char str[20];
  char* str2;
}
```

この str と str2 は全くの別物です．あくまで char str[20] と `char* str` が同じ意味になるのは**仮引数の時だけ**です．

つまり，始めに書いた myPut 関数は

```
void myPuts(char* str)
{
  int i;
  for( i = 0; str[i] != '\0'; i++ ) {
    putchar(str[i]);
```

```
    }
    printf("\n");
}
```

と全く同じ意味となります．また，`void myPuts(char str[])` としても実は同じ意味となります．ここら辺がややこしいところです．

ポイントを整理しましょう．

要点

void myPuts(char* str);
void myPuts(char str[20]);
void myPuts(char str[]);
上記はすべて同じものである．

メイン関数側の `myPuts( str );` の部分は `myPuts( &str[0] )` と同じでしたね．つまり，先頭要素へのアドレスを渡しているわけです．

なぜ配列をそのまま引数として渡せないのでしょうか．
たとえばもし main 関数内で

```
char str[10000];
```

という配列をつくり myPuts にそのまま引数として渡せたとしましょう．
関数の部分で仮引数は実引数のコピーを作ると説明しました．
もし配列のコピーを作ってしまうと，いちいち配列を渡すごとにメモリを大量に消費してしまいます．そのため，配列そのままを渡すのではなく，配列へのポインタを渡すようになっています．

6.7.4 ポインタ配列

`int array[5]` とすると，int 型の配列（要素数は 5 個）を作ることができました．同様に，`int* p[5]` とすると，int*型の配列（要素数は 5 個）を作ることができます．

ここで，次のようなコードを書いたとします．

```
1  #include <stdio.h>
2
```

```
3   int main(void)
4   {
5       int array[5] = {1, 2, 3, 4, 5};
6       int* p[5];
7       int i;
8
9       for(i = 0; i < 5 ; i++) {
10          p[i] = &array[i];
11          printf("p[%d] : %p\n", i, p[i]);
12      }
13
14      *p[0] += 10;
15
16      for(i = 0; i < 5 ; i++) {
17          printf("*p[%d] : %d\n", i, *p[i]);
18      }
19  }
```

初めの for 文は，`array[i]` のアドレスを `p[i]` に代入しています．そして，`p[i]` の中身を表示しています．ここまでの様子を図 6.15 に示します．

ここまでの実行結果は以下のようになるでしょう．

始めの for 文までの実行結果

p[0]：100 番地
p[1]：104 番地
p[2]：108 番地
p[3]：112 番地
p[4]：116 番地

次の `*p[0] += 10` では，p[0] の中身のアドレスが指している場所の値に 10 を足しています．この場合では，100 番地にあるデータに 10 を足していることになります．よって，`array[0]` に 10 を足したということです．そして，その後の for 文内での `*p[i]` の部分は `p[i]` が指している場所のデータを表示しています．

よって，プログラム全体の実行結果は次のようになります．

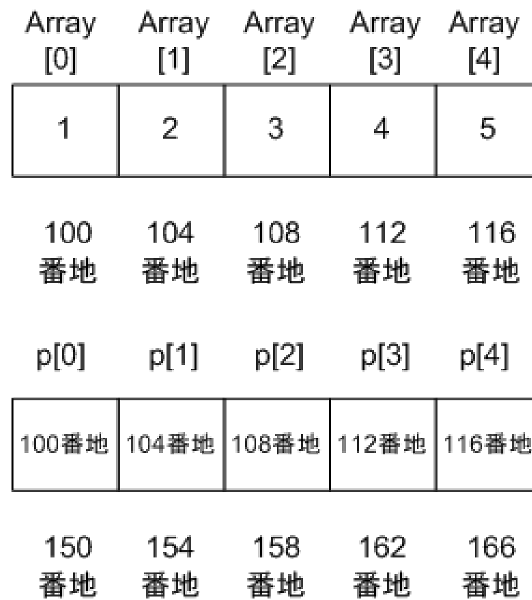

図 6.15: メモリ内部の様子

実行結果

```
p[0] : 100 番地
p[1] : 104 番地
p[2] : 108 番地
p[3] : 112 番地
p[4] : 116 番地
*p[0] : 11
*p[1] : 2
*p[2] : 3
*p[3] : 4
*p[4] : 5
```

6.8 文字列定数とポインタ

文字列定数へのポインタを作ってみましょう．次のコードを見てください．

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

```
2
3   int main(void)
4   {
5       char* str = "Hello";
6       printf("%s\n", str );
7
8       return 0;
9   }
```

このコードの実行結果は以下のようになります.

実行結果

```
Hello
```

このプログラムの様子を表したのが図 6.16 です．図をみて分かるように str は文字列定数の配列の先頭要素へのアドレスとなっています．文字列定数には変数名が付いていません．つまり，ポインタを使わないと絶対にアクセスできない変数です.

図 6.16: メモリ内部の様子

さて， `char* str = "Hello"` と `char str2[10] = "Hello"` の違いはなんでしょうか．図 6.17 を見てください.

まず str は char へのポインタであるのに対し，str2 は文字列配列です．それだけではなく，この二つには重要な違いがあります．次のプログラムコードを見てください.

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

図 6.17: メモリ内部の様子

```
2
3  int main(void)
4  {
5      char* str = "Hello";
6      char str2[10] = "Hello";
7
8      str[0] = 'B';    /* ここはエラー */
9      str2[0] = 'B';   /* ここは OK */
10
11     return 0;
12 }
```

一見同じようなプログラムコードに見えますが，両者は全く違います．`str[0]` というのは `*(str + 0)` と同じでしたね．また，str は文字列定数を指しています．文字列定数とは名前の通り定数です．定数の値を書き換えることはできません．なので， `str[0] = 'B'` というようなコードを書いてはいけません．

str2 は変数なので，自由に値を書き換えることができます．

6.9 ポインタのポインタ

今までは char 型や int 型へのポインタを作ってきましたが，char*型や int*型のポインタも作ることができます．つまり，ポインタを使って別のポインタを指すことができます．

次のコードを見てください．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int hoge = 10;
6      int* p_hoge;
7      int** p_p_hoge;
8
9      p_hoge = &hoge;
10     p_p_hoge = &p_hoge;
11
12     printf("hoge = %d\n", hoge);
13     printf("*p_hoge = %d\n", *p_hoge);
14     printf("p_hoge address = %p\n", &p_hoge);
15     printf("p_p_hoge = %p\n", p_p_hoge);
16     printf("**p_p_hoge = %d\n", **p_p_hoge);
17
18     return 0;
19 }
```

このプログラムの 10 行目までの様子を表したのが図 6.18 及び図 6.19 です．

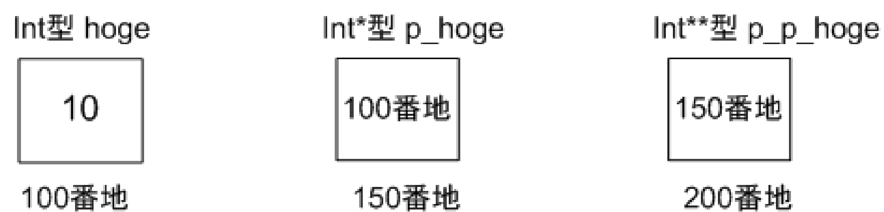

図 6.18: メモリ内部の様子

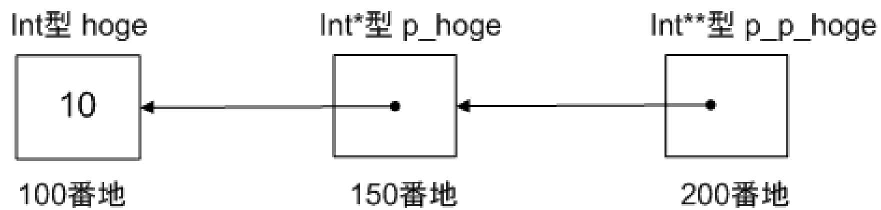

図 6.19: メモリ内部の様子

p_hoge には hoge のアドレスだけが代入できます．

p_p_hoge は int*型の変数のアドレスだけが格納できるものです．よって p_p_hoge には p_hoge のアドレスだけが代入できます．hoge のアドレスなどは代入できません（図 6.20）．

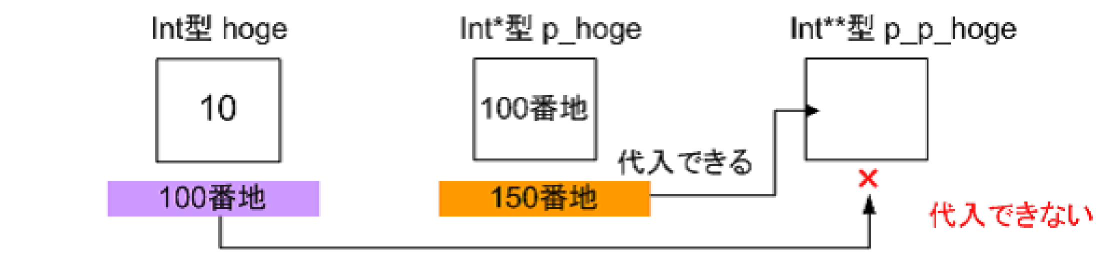

図 6.20: メモリ内部の様子

また，**p_p_hoge は*(*p_p_hoge) と同じです．つまり *(p_hoge) と同じです．

このプログラムの実行結果は以下のようになります．

実行結果

```
hoge = 10
*p_hoge = 10
p_hoge address = 150 番地
p_p_hoge = 150 番地
**p_p_hoge = 10
```

6.10 例題

6.10.1 自作 strlen 関数の作成再び

仕様

標準関数 strlen と同等の機能を持つ関数 my_strlen を作成しなさい.

文字列の長さを計算する my_strlen を次のようにして作ることも出来ます.

```
1  #include <stdio.h>
2
3  int my_strlen(const char* str)
4  {
5      int i = 0;
6      while( *str++ ) {
7          i++;
8      }
9      return i;
10 }
11
12 int main(void)
13 {
14     char str[20] = "Hello world";
15     printf("%s の長さは %d です\n", str, my_strlen(str));
16     return 0;
17 }
```

const char*とは str が指しているデータの中身を書き換えないようにという命令です．つまり，my_strlen の中では

```
str[i] = ’E’;
```

つまり

```
*(str + i) = ’E’;
```

などの命令はエラーとなります.

`*str++;` という部分は `*(str++);` だということに注意してください.

また，文字列の最後には NULL 文字が含まれています．NULL 文字（`\0`）は 0 と同じ値です. つまり，while(0) になったらループを終了するのですね．このような書き方は良く出てくるので覚えて置いてください.

6.10.2 自作 strcat 関数の作成再び

仕様

標準関数 strcpy と同等の機能を持つ関数 my_strcpy を作成しなさい.

```
1  #include <stdio.h>
2
3  void my_strcpy(char* dest, const char* src)
4  {
5      while( *dest++ = *src++ )
6          ;
7  }
8
9  int main(void)
10 {
11     char str[20] = "Hello world";
12     char copy[20];
13
14     my_strcpy(copy, str);
15     printf("コピーした文字列は %s です\n",copy);
16     return 0;
17 }
```

6.11 演習問題

問題 1

次のコードの実行結果を予測しなさい.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int hoge = 30;
6      int* piyo = &hoge;
7
8      printf("out1 :%d\n",hoge);
9      printf("out2 :%d\n",*piyo);
```

```
10
11      return 0;
12  }
```

問題 2

次のコードの実行結果を予測しなさい.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int x = 10;
6       int y = 20;
7       int* hoge = &x;
8       int* piyo = &y;
9       int* tmp;
10
11      tmp = hoge;
12      hoge = piyo;
13      piyo = tmp;
14
15      printf("out1 :%d\n",x);
16      printf("out2 :%d\n",*hoge);
17
18      return 0;
19  }
```

問題 3

次のコードの実行結果を予測しなさい.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int array[5] = {1,2,3,4,5};
6       int* p = &array[0];
7
```

```
8        p += 2;
9        printf("out1 :%d\n",*p);
10
11       return 0;
12   }
```

問題 4

次のプログラムの実行結果を予測しなさい.

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        int array[5] = { 1, 5, 10, 15, 20 };
6        int* p;
7
8        p = &array[0];
9
10       printf("%d %d %d %d %d\n",*(p+0), *(p+1), p[2], 3[p], *p );
11       p = array;
12       printf("%d %d %d %d %d\n",*(p+0), *(p+1), p[2], 3[p], *p );
13
14       *p += 10;
15       printf("p[0] : %d\n",p[0]);
16
17       (*p)++;
18       printf("*p : %d\n",*p);
19
20       *(p++);
21       printf("*p : %d\n",*p);
22   }
```

問題 5

次のプログラムの問題点を指摘しなさい.

```
1    #include <stdio.h>
2
```

```
3   int main(void)
4   {
5       char* p = "Hello world";
6
7       printf("%s\n",p);
8       p[1] = 'a';
9       printf("%s\n",p);
10
11      return 0;
12  }
```

問題 6

次のプログラムの実行結果を予測せよ．ただし，hoge のアドレスは 100 番地，i のアドレスは 200 番地，j のアドレスは 300 番地，k のアドレスは 400 番地とせよ．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int hoge = 10;
6       int* i = &hoge;
7       int** j = &i;
8       int*** k = &j;
9
10      printf("hoge = %d\n", hoge);
11      printf("&hoge= %p\n", &hoge);
12      printf("*i   = %d\n", *i);
13      printf("&i   = %p\n", &i);
14      printf("&j   = %p\n", &j);
15      printf("*j   = %p\n", *j);
16      printf("**j  = %d\n", **j);
17      printf("k    = %p\n", k);
18      printf("*k   = %p\n", *k);
19      printf("**k  = %p\n", **k);
20      printf("***k = %d\n", ***k);
21
22      return 0;
23  }
```

第7章　構造体

7.1　構造体

構造体とは，複数の変数をまとめて使いたい時に使います．使い方は次のようになります．

書式

```
struct 構造体の名前 {
    構造体の中身
};
```

では，構造体を使ったプログラムの例を見てみましょう．

```
1  #include <stdio.h>
2
3  #define NUMBER_OF_STUDENT 3
4
5  struct Student{
6      char student_name[100];
7      int student_age;
8  };
9
10 int main(void)
11 {
12     struct Student st[NUMBER_OF_STUDENT];
13     int i;
14     int num;
15     printf("%d 人の生徒の情報を入力してください\n",NUMBER_OF_STUDENT);
16
17     for( i=0; i < NUMBER_OF_STUDENT ; i++ ) {
18         printf("%d 人目の名前 :",i+1);
19         scanf("%s",st[i].student_name);
20         printf("%d 人目の年齢 :",i+1);
```

```
21            scanf("%d",&st[i].student_age);
22        }
23
24        for(;;) {
25            printf("何人目の生徒の情報が見たいですか :");
26            scanf("%d",&num);
27
28            if( num > NUMBER_OF_STUDENT || num <= 0) {
29                    printf("終了します.\n");
30                    break; /* 無限ループから抜ける */
31            } else {
32                    printf("名前 : %s, 年齢 : %d\n",st[num-1].student_name,
33                            st[num-1].student_age);
34            }
35        }
36        return 0;
37    }
```

まず，struct Student st[NUMBER_OF_STUDENT]; という部分で，Student という構造体の変数を NUMBER_OF_STUDENT 個だけ作っています．そして，構造体の中にある変数にアクセスするときには，st[0].student_name とします．

このプログラムの実行結果は以下のようになります．

実行結果

```
3人の生徒の情報を入力してください
1人目の名前 :nishio
1人目の年齢 :20
2人目の名前 :itoh
2人目の年齢 :20
3人目の名前 :yamada
3人目の年齢 :33
何人目の生徒の情報が見たいですか :2
名前 : itoh, 年齢 : 20
何人目の生徒の情報が見たいですか :1
名前 : nishio, 年齢 : 20
何人目の生徒の情報が見たいですか :3
名前 : yamada, 年齢 : 33
何人目の生徒の情報が見たいですか :0
終了します.
```

7.2 構造体へのポインタ

int 型のポインタなどと同様に，構造体のポインタも作成することができます．int 型のポインタを作成したい場合， `int* hoge;` などと宣言しました．ここで，仮に

```
struct Student{
    char student_name[100];
    int student_age;
};
```

という構造体を作ったとします．Student 構造体のポインタを作成するには，

```
struct Student* hoge;
```

と宣言します．

ここで注意しなければならないのは，hoge の中にある要素にアクセスしたい場合です．hoge 内部の要素にアクセスする場合は，

```
hoge->student_age
```

とします． `->` はアロー演算子と呼ばれ， `-` (マイナス）と `>` （大なり）を合わせて作ったものです．なぜこのようなものが使われるのでしょうか．

例えば次のようにして構造体内部のデータにアクセスしてみましょう．

```
(*hoge).student_age
```

これは問題ありません．hogeが指している先のデータ（構造体）の中のstudent_ageというデータを取り出せ，という意味です．

では，次のようにしたらどうなるでしょうか．

```
*hoge.student_age
```

多くの人はこれを，

```
(*hoge).student_age
```

と勘違いしてしまいますが，優先順位 (表 7.2) の関係でこれは，

```
*(hoge.student_age)
```

となってしまい，意味がぜんぜん違うものになってしまいます．この混乱を解消するために，アロー演算子というものが開発されました．

表 7.1: 演算子の優先順位

<table>
<tr><th>優先度順位</th><th>演算子</th></tr>
<tr><td>優先度が高い</td><td>( ) [ ] -> .</td></tr>
<tr><td></td><td>! ˜ ++ == = * & sizeof</td></tr>
<tr><td></td><td>* / %</td></tr>
<tr><td></td><td>+ -</td></tr>
<tr><td></td><td><< >></td></tr>
<tr><td></td><td>< <= > >=</td></tr>
<tr><td></td><td>== !=</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td></td><td>^</td></tr>
<tr><td></td><td>|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td></td><td>||</td></tr>
<tr><td></td><td>?</td></tr>
<tr><td></td><td>= += -= *= /=</td></tr>
<tr><td>優先度が低い</td><td>,</td></tr>
</table>

7.3 typedef指定子

typedef 指定子は，既存の変数の型に別の名前を付けることができるものです．

書式
```
typedef 既存の型名 別名;
```

例えば unsigned int 型のように長い名前の型を typedef を使うことで短い名前を付けることが可能です．

```
tyepdef unsigned int uint;
```

こうすることで，unsigned int は uint としても使うことができます．

このtypedef を構造体に適用してみましょう．次のプログラムを見てください．

```
1   #include <stdio.h>
2
3   #define NUMBER_OF_STUDENT 3
4
5   typedef struct Student_tag{
6       char student_name[100];
7       int student_age;
8   } Student;
9
10  int main(void)
11  {
12      Student st[NUMBER_OF_STUDENT];
13      int i;
14      int num;
15      printf("%d 人の生徒の情報を入力してください\n",NUMBER_OF_STUDENT);
16
17      for( i=0; i < NUMBER_OF_STUDENT ; i++ ) {
18          printf("%d 人目の名前 :",i+1);
19          scanf("%s",st[i].student_name);
20          printf("%d 人目の年齢 :",i+1);
21          scanf("%d",&st[i].student_age);
22      }
23
24      for(;;) {
25          printf("何人目の生徒の情報が見たいですか :");
```

```
26            scanf("%d",&num);
27
28            if( num > NUMBER_OF_STUDENT || num <= 0) {
29                    printf("終了します.\n");
30                    break; /* 無限ループから抜ける */
31            } else {
32                    printf("名前 : %s, 年齢 : %d\n",st[num-1].student_name,
33                            st[num-1].student_age);
34            }
35        }
36        return 0;
37    }
```

ソースコードの12行目を見てください．`struct Student_tag st[NUMBER_OF_STUDENT];` としなくても構造体を作成できていることが分かります．typedefを使い，

```
typedef struct
typedef struct Student_tag{
    char student_name[100];
    int student_age;
} Student;
```

のように宣言すると, この構造体の名前はStudentになります.

7.4 例題

7.4.1 2点間の距離を求めるプログラム

仕様

2つの$p1(x_1, y_1), p2(x_2, y_2)$座標を受け取り，その2点間の距離を求めるプログラムを作成しなさい．なお，2点間の距離dは

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

で求めることができる.

```
1   #include <stdio.h>
2   #include <math.h>
3
```

```
4   typedef struct Point_tag{
5           double x;
6           double y;
7   } Point;
8
9   int main(void)
10  {
11          Point p1, p2;
12          double distance;
13
14          printf("p1 の (x,y) 座標を入力 :");
15          scanf("%lf %lf", &p1.x, &p1.y);
16
17          printf("p2 の (x,y) 座標を入力 :");
18          scanf("%lf %lf", &p2.x, &p2.y);
19
20          distance = sqrt( (p2.x - p1.x) * (p2.x - p1.x) +
21                                  (p2.y - p1.y) * (p2.y - p1.y) );
22          printf("2 点間の距離 : %f\n", distance );
23          return 0;
24  }
```

第8章　メモリ管理とファイル入出力

8.1　動的メモリ割り当て

C 言語には動的なメモリ割り当てを行うことができる malloc という関数があります．この関数を使うことで可変長の配列を作成することができます．

次のコードを見てください．

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5           int buf[10];
6           int number, i, sum;
7
8           printf("入力データ数 : ");
9           scanf("%d", &number);
10
11          for(i = 0; i < number; i++) {
12                  printf("%d 個目のデータ :", i+1);
13                  scanf("%d", &buf[i] );
14          }
15
16          for(i = 0, sum = 0; i < number; i++ ) {
17                  sum += buf[i];
18          }
19
20          printf("合計 : %d\n", sum);
21          return 0;
22  }
```

このプログラムはユーザが指定した数だけのデータを受け取り，受け取った数値の合計を求めるプログラムです．

このプログラムには重大な問題点があります．buf は 10 個以上のデータを保存することはできません．しかし，例えば 1000 個のデータを入力したいとします．そうした場合はどうしたらよいでしょうか．素朴な実装では次のようにします．

```
int buf[1000];
```

しかし，この場合 10 個だけのデータを受け取りたい場合，残り 990 個の変数が無駄になってしまいます．

この問題を解決するためには，可変長の配列を作成します．可変長配列の作成には malloc 関数を使用します．この関数は stdlib.h をインクルードしなければなりません．

書式

```
ポインタ変数 = malloc( メモリバイト数 );
```

malloc 関数はメモリバイト数で割り当てた領域の先頭を指すポインタを返します．メモリ割り当てに失敗した場合は NULL ポインタが返ってきます．[1]

また，割り当てたメモリはプログラムが終了するまで確保されますが，メモリが不要になった場合には自分で開放してやる必要があります．こうしないと，無駄なメモリが残り続けてしまいます．メモリ開放には free 関数を使用します．

書式

```
free( ポインタ変数 );
```

例えば int 型の配列（要素数 5）を作成したい場合は次のようにします．

```
int* p;

p = (int *)malloc( 4 * 5 );

if( !p ) {
    printf("メモリ割り当て失敗\n");
    exit(1); /* プログラムを終了させる関数 */
}
```

int 型のサイズを 4 バイトとした場合，int 型の変数を 5 つ持った配列を作るので割り当てるメモリサイズは 20bytes です (図 8.1).

しかし，int は 4 バイトとは限りません．よって int 型のバイト数を取得する必要があります．そういった場合には sizeof を使用します．

[1] メモリの容量が足りない時などに malloc は失敗します．

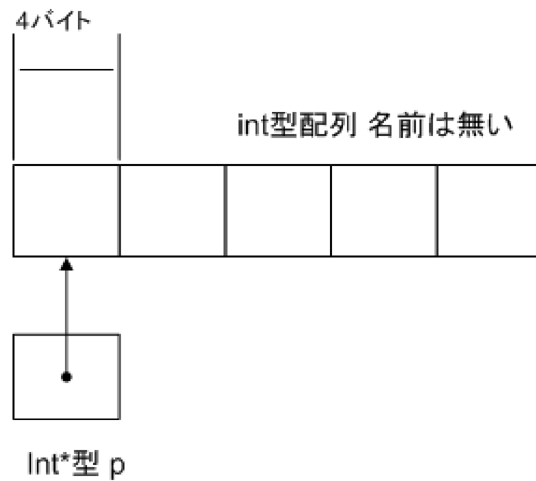

図 8.1: 動的メモリの割り当て

```
p = (int *)malloc( sizeof(int) * 5 );
```

では malloc を使って先ほどのプログラムを作成してみましょう.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6           int* buf;
7           int number, i, sum;
8
9           printf("入力データ数 : ");
10          scanf("%d", &number);
11
12          buf = (int *)malloc( number * sizeof(int) );
13
14          if( !buf ) {
15                  printf("メモリ割り当て失敗\n");
16                  exit(1);
17          }
18
19          for(i = 0; i < number; i++) {
```

```
20                  printf("%d 個目のデータ :", i+1);
21                  scanf("%d", &buf[i] );
22          }
23
24          for(i = 0, sum = 0; i < number; i++ ) {
25                  sum += buf[i];
26          }
27
28          printf("合計 : %d\n", sum);
29          free( buf );
30          return 0;
31  }
```

exit 関数を実行した場合，プログラムは終了します．引数には 0 または 0 以外の数字を渡します．0 を渡した場合には正常終了，0 以外の数値を渡した場合は異常終了となります．

このプログラムの実行結果は次の通りです．

実行結果

```
入力データ数 : 5
1 個目のデータ :1
2 個目のデータ :2
3 個目のデータ :3
4 個目のデータ :4
5 個目のデータ :5
合計 : 15
```

8.2 ファイル入出力

8.2.1 ストリーム

ファイル入出力においてストリームという重要な概念が存在します．C 言語では，どのような入出力デバイス（キーボードやディスプレイ，ディスクファイルなど）においても同じ手法で操作することができます．これは，ストリームと呼ばれる抽象的なインターフェースを操作するから実現可能なことなのです．ファイル入出力を行いたい場合はストリームを通してファイルを操作することになります．

8.2.2 ファイル開閉

では，ファイルを開く方法について学びましょう．まず，ファイルを開きストリームと結びつける必要があります．そのためには，FILE 構造体と fopen 関数を使います．

書式

```
FILE* fp = fopen(char* ファイル名, char* モード);
```

これらの関数，構造体は stdio.h をインクルードすることで使用できます．fopen 関数はファイル名とファイルモードを渡すと FILE 構造体のポインタを返します．この FILE 構造体がストリームであり，これらを使ってファイルの入出力を行っていきます．

ここでモードという引数が出てきました．このモードは表 8.2.2 にある文字列を渡すことでファイルのモードを選択することができます．

表 8.1: モード指定可能な値

モード	説明
r	読み込み用にテキストファイルを開く
w	書き込み用にテキストファイルを開く
a	テキストファイルにデータを追加する
rb	読み込み用にバイナリファイルを開く
wb	書き込み用にバイナリファイルを開く
ab	バイナリファイルにデータを追加する
r+	読み込み/書き込み用にテキストファイルを開く
w+	読み込み/書き込み用にテキストファイルを作成する
a+	読み込み/書き込み用にテキストファイルを追加する

r モードでは，ファイルが見つからなかった場合 fopen 関数は NULL を返します．w モードでは，ファイルが既に存在していた場合にはその内容を破壊します．a モードでは，ファイルが存在しなかった場合にはファイルを新たに作成します．

また，テキストモードとバイナリモードというものが存在しています．テキストモードでは主に ASCII 文字列を扱い，何らかの文字変換が行われる場合があります．例えば，改行コードは勝手に文字変換される場合があります．Windows の場合，改行コードは実は`\n`ではなく`\r\n`なのです．しかし，UNIX 系の OS の場合，改行は`\n`であったりするわけです．C 言語では OS によって改行コードを自動的に判別して勝手に変換してくれます．

これに対して，バイナリモードでは文字変換は一切行われません．

ファイルは開いたら必ず閉じなければなりません．ストリームからファイルを切り離すには fclose 関数を使用します.

書式

```
fclose( FILE* ストリーム );
```

では，指定したファイルが存在するかどうかを確認するプログラムを作成してみましょう.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       char filename[100];
7       FILE *fp;
8
9       printf("ファイル名 :");
10      scanf("%s" , filename);
11
12      fp = fopen(filename , "r");
13      if (fp == NULL) {
14          printf("ファイルが開けませんでした\n");
15          exit (1);
16      }
17      printf("ファイルを正常に開くことができました\n");
18
19      fclose(fp);
20      return 0;
21  }
```

8.2.3 ファイル読み込み

開いたファイルの内容を表示させてみましょう．ファイルから文字を取得するには fgetc 関数を使用します.

書式

```
int fgetc(FILE* ストリーム);
```

fgetc 関数はストリームから unsigned char 型の値を 1 文字読み取り，int 型の値として返します．では，ソースファイルと同じ場所に testfile.txt というファイルを作成してください．ファイルの中身は

hello

world

363

としてみましょう．そして，このファイルを開くプログラムを作成してみましょう．

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       char filename[100] = "testfile.txt";
7       FILE *fp;
8       int ch;
9
10      fp = fopen(filename , "r");
11      if (fp == NULL) {
12          printf("ファイルが開けませんでした\n");
13          exit (1);
14      }
15
16      for(;;) {
17          ch = fgetc( fp );
18          if( ch == EOF ) {
19                  break;
20          }
21          printf("%c", ch);
22      }
23      printf("\n");
24
25      fclose(fp);
26      return 0;
27  }
```

このプログラムの実行結果は以下の通りです．

実行結果

hello
world
363

fgetc で取得した文字を表示していくプログラムです．ここで ch が EOF[2]となった時に無限ループから抜けています．この EOF はファイル読み込み中にエラーが発生したか，またはファイルの最後に達した場合に fgetc が返します．

ファイルから 1 文字ずつではなく，文字列として読み取る関数も存在しています．それが fgets 関数です．

書式

```
char* fgets(char* 文字列, int サイズ, FILE* ストリーム);
```

fgets 関数では第 1 引数の文字列に読み込んだ文字列を格納していきます．また，fgets 関数はサイズ-1 の文字を読み取るか，改行またはファイルの終わりまで読み取ります．ストリームには読み出すストリームを指定します．戻り値は読み込んだ文字列の先頭へのポインタ，または読み取りに失敗した場合は NULL を返します．

では，先ほどのプログラムを fgets を使って書き直してみましょう．

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define BUFSIZE 100
5
6   int main()
7   {
8           char filename[100] = "testfile.txt";
9           char buf[BUFSIZE];
10          FILE *fp;
11
12          fp = fopen(filename , "r");
13          if (fp == NULL) {
14                  printf("ファイルが開けませんでした\n");
15                  exit (1);
16          }
17
18          while( fgets( buf, BUFSIZE, fp ) != NULL ) {
```

[2]End of File の略です．

```
19                  printf("%s", buf);
20          }
21          printf("\n");
22          fclose(fp);
23          return 0;
24  }
```

fgets 関数以外に fscanf 関数というものも存在しています．この関数は scanf 関数によく似ています．

書式

```
int fscanf(FILE* ストリーム, char* フォーマット, ...);
```

エラーが発生した，またはファイルの終端まで読み取った場合 EOF を返します．先ほどのプログラムを fscanf 関数を使って書き換えてみましょう．

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define BUFSIZE 100
5
6   int main()
7   {
8           char filename[100] = "testfile.txt";
9           char buf[BUFSIZE];
10          FILE *fp;
11
12          fp = fopen(filename , "r");
13          if (fp == NULL) {
14                  printf("ファイルが開けませんでした\n");
15                  exit (1);
16          }
17
18          while( fscanf( fp, "%s", buf ) != EOF ) {
19                  printf("%s\n", buf);
20          }
21          fclose(fp);
22          return 0;
23  }
```

8.2.4 ファイル書き込み

ファイルにデータを書き込んでみましょう．1 文字ずつファイルに書き込んでいく時は，fputc 関数を使用します．

書式

```
int fputc(int 文字, FILE* ストリーム);
```

ファイルに hello という文字を書き込んでみましょう．ファイル名は out.txt としましょう．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6          char filename[100] = "out.txt";
7          char buf[100] = "hello";
8          FILE* fp;
9          int i;
10
11         fp = fopen(filename , "w");
12         if (fp == NULL) {
13                 printf("ファイルが開けませんでした\n");
14                 exit (1);
15         }
16
17         for( i = 0; buf[i] != '\0'; i++ ) {
18                 fputc( buf[i], fp );
19         }
20         printf("ファイル書き込み完了\n");
21         fclose(fp);
22         return 0;
23 }
```

ファイルモードを w にするのを忘れないようにしましょう．書き込んだファイル out.txt をエディタで開いてみてください．ちゃんと書き込まれているでしょうか．

実行結果

```
ファイル書き込み完了
```

文字列を書き込むには fputs 関数を使います.

書式

```
int fputs(char* 文字列, FILE* ストリーム);
```

fputs 関数は書き込みに失敗すると EOF を返します.
では，先ほどのプログラムを fputs 関数を使って書き換えてみましょう.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6           char filename[100] = "out.txt";
7           char buf[100] = "hello";
8           FILE* fp;
9
10          fp = fopen(filename , "w");
11          if (fp == NULL) {
12                  printf("ファイルが開けませんでした\n");
13                  exit (1);
14          }
15
16          if( fputs( buf, fp ) != EOF ) {
17                  printf("ファイル書き込み完了\n");
18          }else {
19                  printf("書き込み失敗\n");
20          }
21          fclose(fp);
22          return 0;
23  }
```

また，fputs 関数以外に fprintf 関数というものも存在しています．これは printf 関数とよく似ています.

書式

```
int fprintf(FILE* ストリーム, char* フォーマット, ...);
```

先ほどのプログラムを fprintf 関数を使って書き換えてみましょう.

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

```
2   #include <stdlib.h>
3
4   int main()
5   {
6           char filename[100] = "out.txt";
7           char buf[100] = "hello";
8           FILE* fp;
9
10          fp = fopen(filename , "w");
11          if (fp == NULL) {
12                  printf("ファイルが開けませんでした\n");
13                  exit (1);
14          }
15
16          fprintf( fp, "%s", buf );
17          printf("ファイル書き込み完了\n");
18          fclose(fp);
19          return 0;
20  }
```

8.2.5 ファイル終端の判定とエラーチェック

fgetc や fgets 関数などは，ファイルの終端に達した，またはエラーが発生した場合には EOF を返却していました．しかし，EOF は次のように定義されています.[3]

```
#define EOF -1
```

この場合，もし読み取った値が-1 であった場合，EOF と判定されてしまいます. これは問題です.

この問題を解決するには feof 関数を使用します.

書式

```
int feof(FILE* ストリーム);
```

feof 関数はエラーが発生した場合，またはファイルの終端に達した場合は 0 以外の値を，そうでなければ 0 を返却します.

では，textfile.txt を読み込むプログラムを feof 関数を使って書き換えてみましょう．ファイルの中身は

[3] 環境によっては-1 で無い場合があります

hello

world

363

としてみましょう

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       char filename[100] = "testfile.txt";
7       FILE *fp;
8       int ch;
9
10      fp = fopen(filename , "r");
11      if (fp == NULL) {
12          printf("ファイルが開けませんでした\n");
13          exit (1);
14      }
15
16          for(;;) {
17                  ch = fgetc( fp );
18                  if( !feof( fp ) ) {
19                          printf("%c", ch);
20                  }else {
21                          break;
22                  }
23      }
24      printf("\n");
25
26      fclose(fp);
27      return 0;
28  }
```

実行結果

hello
world
363

エラーをチェックする ferror 関数も存在します．この関数はストリームでエラーが発生すると 0 以外の値を返します.

書式

```
int ferror(FILE* ストリーム);
```

先ほどのプログラムに ferror 関数を加えてみましょう.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       char filename[100] = "testfile.txt";
7       FILE *fp;
8       int ch;
9
10      fp = fopen(filename , "r");
11      if (fp == NULL) {
12          printf("ファイルが開けませんでした\n");
13          exit (1);
14      }
15
16          for(;;) {
17                  if( ferror( fp ) ) {
18                          printf("ファイルエラー発生\n");
19                          break;
20                  }
21
22                  ch = fgetc( fp );
23
24                  if( !feof( fp ) ) {
25                          printf("%c", ch);
26                  }else {
27                          break;
28                  }
29      }
30      printf("\n");
31
```

```
32          fclose(fp);
33          return 0;
34  }
```

このようなエラーチェックは実用的なプログラムを開発する時には使用するようにしましょう．

8.3 標準ストリーム

C 言語には標準ストリームというものが 3 つ存在しています．それぞれ標準入力 (stdin)，標準出力 (stdout)，標準エラー (stderr) です．デフォルトでは標準入力はキーボード，標準出力，標準エラーはディスプレイです．

では，画面に Hello world と表示するプログラムを fprintf 関数を使って作成してみましょう．

```
1   #include <stdio.h>
2
3   int main()
4   {
5           fprintf( stdout, "Hello world\n" );
6           return 0;
7   }
```

実行結果

```
Hello world
```

また，キーボードから文字列を受け取り，その文字列を表示するプログラムを作成してみましょう．

```
1   #include <stdio.h>
2
3   int main()
4   {
5           char buf[100];
6
7           printf("文字列を入力 :");
8
9           fscanf( stdin, "%s", buf );
10          fprintf( stdout, "入力した文字列 : %s\n", buf );
```

```
11          return 0;
12  }
```

実行結果

文字列を入力：Hello
入力した文字列：Hello

8.4 セキュアプログラミング

今まで文字列をキーボードから読み取る時は scanf 関数を使ってきました．しかし，この scanf 関数には重大な問題点があります．例えば次のようなプログラムを見てください．

```
1   #include <stdio.h>
2
3   int main()
4   {
5           char buf[10];
6
7           printf("文字列を入力 :");
8           scanf("%s", buf);
9           printf("入力された文字列 : %s", buf);
10          return 0;
11  }
```

このプログラムを Visual C++ 2003 で次のように実行させてみます．

実行結果

文字列を入力 :helloworld111111
入力された文字列：helloworld111111
ハンドルされていない例外：System.NullReferenceException: オブジェクト参照がオブジェクト インスタンスに設定されていません。

エラーが発生してしまいました．環境によってはエラーが出ないで正常終了する場合もあります．しかしよく考えてください．buf は 10 文字しか格納することができないのに，10 文字以上のファイルを格納しています．これは実はメモリ上の他の領域を破壊している事になります．

このようにメモリ領域を破壊することをバッファオーバーランといいます．クラッカーはこのバッファオーバーランを意図的に起こしてデータの改竄などを行

います．コンピュータウィルスなどもこのバッファオーバーランを利用したものがあります.

このようなバッファオーバーランを発生させないためには読み取る文字数を制限します．例えば次のようにすれば文字数を制限することができます.

```
1  #include <stdio.h>
2
3  int main()
4  {
5          char buf[10];
6
7          printf("文字列を入力 :");
8          fgets( buf, 10, stdin );
9          printf("入力された文字列 : %s\n", buf);
10         return 0;
11 }
```

実行結果

文字列を入力 :helloworld11111111
入力された文字列 : helloworl

8.5 例題

8.5.1 ファイルサイズを求めるプログラム

仕様

指定したファイルの容量を計算するプログラムを作成せよ.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  #define BUFMAX 100
6
7  int main()
8  {
```

```
 9          FILE* fp;
10          char filename[BUFMAX];
11          int size = 0, i;
12
13          printf("ファイル名 :");
14          fgets( filename, BUFMAX , stdin );
15
16          /* 改行を削除する */
17          i = strlen( filename ) - 1;
18          if( filename[i] == '\n' ) {
19                  filename[i] = '\0';
20          }
21
22          fp = fopen( filename, "rb" );
23
24          if( fp == NULL ) {
25                  printf("ファイルが開けませんでした\n");
26                  exit( 1 );
27          }
28
29          while( !feof( fp ) ) {
30                  fgetc( fp );
31                  size++;
32          }
33          size--; /* \0 の分を削除 */
34
35          printf("ファイルサイズ : %d\n", size);
36          fclose( fp );
37
38          return 0;
39  }
```

fgets 関数は改行も読み取ってしまうので，削除してやる必要があります.

8.5.2　平均点を求めるプログラム

仕様

次のような成績表ファイル seiseki.txt が与えられました.

nishio 90

yamada 68

shimizu 100

yoshioka 74

hosaka 28

成績表ファイルのフォーマットは 学生名, 得点となっています. この成績表から平均点を求めるプログラムを作成しなさい.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6          FILE* fp;
7          char filename[100] = "seiseki.txt";
8          char buf[100];
9          int tmp, sum = 0, number = 0;
10         double average;
11
12         fp = fopen( filename, "r" );
13
14         if( fp == NULL ) {
15                 printf("ファイルが開けませんでした\n");
16                 exit( 1 );
17         }
18
19         while( !feof( fp ) ) {
20                 fscanf( fp, "%s %d", buf, &tmp);
21                 number++;
22                 sum += tmp;
23         }
```

```
24
25            average = sum / number;
26            printf("平均点 : %f\n", average );
27            fclose( fp );
28
29            return 0;
30    }
```

8.6 演習問題

問題 1

指定したファイルの中身を表示するプログラムを作成せよ.

問題 2

指定したファイルの文字数及び行数を求めるプログラムを作成せよ.

問題 3

指定したファイルをコピーするプログラムを作成せよ.

問題 4

指定した 2 つのファイルの中身を交換するプログラムを作成せよ.

参考文献

[1] Brian W. Kernighan, Dennis M. Ritchie 著, The C Programming Language Second Edition

[2] Peter van der Linden 著, Expert C Programming Deep C Secrets

[3] ハーバートシルト 著 トップスタジオ訳, 独習 C 第三版

[4] Bjarne Stroustrup 著 長尾高弘 訳, プログラミング言語 C++ 第三版

[5] 前橋和弥 著, ポインタ完全制覇

[6] 浅井淳 著, ポインタが理解できない理由

[7] 田中敏幸 著, C 言語によるプログラミングの基礎

[8] 柴田望洋 著, 明解 C 言語

[9] 坂井 弘亮 著, C 言語 入門書の次に読む本

[10] 奥村晴彦 著, C 言語による最新アルゴリズム事典

[11] 松本健一　上田悦子　安室喜弘　著, 2006 年度プログラミング演習課題, http://chihara.naist.jp/people/STAFF/yasumuro/Pub/c-ensyu2006/

[12] Ryo Kawahara 著, C/C++ プログラミング初心者講座, http://www.stat.phys.kyushu-u.ac.jp/~ryokawa/cbegin2_3/pdf/cbegin.pdf

[13] TOMOJI 著, 初心者のためのポイント学習 C 言語, http://www9.plala.or.jp/sgwr-t/

[14] 小出俊夫 著, C 言語入門インターネット版, http://homepage1.nifty.com/toshio-k/prog/c/

[15] 赤坂玲音 著, C 言語入門, http://wisdom.sakura.ne.jp/programming/c/index.html

演習問題解答

第1章

問題1

```
#include <stdio.h>

int main(void)
{
  printf("Nishio\n");
  return 0;
}
```

問題2

```
#include <stdio.h>

int main(void)
{
  printf("\a");
  return 0;
}
```

問題3

Hello が表示される．なぜなら，\0 は文字列の終端を表す記号だからである．

問題4

```
#include <stdio.h>

int main(void)
{
  printf("The ICMP source quench message is "
    "the TCP/IP equivalent of telling"
    " another computer:"
    "\"I can’t keep up with"
    " all the traffic you’re "
    "sending me - slow down, please.\"");
  return 0;
}
```

第2章

問題1

```
#include <stdio.h>

int main(void)
{
  double hour, minute;

  printf("時間を入力してください（分単位）:");
  scanf("%lf", &minute);

  /* 分を 60 で割ると時間に変換できる */
  hour = minute / 60;
  printf("%f 分は%f 時間です\n", minute, hour);

  return 0;
}
```

問題2

```
#include <stdio.h>

int main(void)
{
  double c; /* 摂氏 */
  double f; /* 華氏 */

  printf("摂氏温度を入力 [C] :");
  scanf("%lf", &c );

  f = (9.0 / 5.0) * c + 32.0;
  printf("摂氏 %f[C] は華氏 %f[F] はです. \n", c, f);

  return 0;
}
```

問題3

0 が出力される．

問題4

str の要素数が少ない．

問題5

省略

索 引

10 進数, 24
16 進数, 24
2 進数, 24

C99, 12
char, 18
const, 26

double, 18, 19

float, 18

int, 18, 19

long, 18

main 関数, 8

NULL 文字, 11
n 進数, 24

printf 関数, 9

scanf 関数, 31
short, 18

unsigned char, 18
unsigned int, 18
unsigned long, 18
unsigned short, 18

暗黙の型変換, 30
インクリメント, 29
エスケープシークエンス, 10

型, 17
機械語, 1
キャスト, 23, 30
コメント, 11
コンパイラ, 1
コンパイル, 1

四則演算, 27
剰余, 28
初期化, 21
処理系依存, 18
スキャン集合, 41
セキュアプログラミング, 31
ソース, 8

代入, 21
デクリメント, 29

配列, 32
配列の初期化, 34
ブロック, 19
べき乗, 29
変換指定子, 22
変数, 17

文字コード, 38
文字コード表, 38
文字配列, 35
文字化け, 40

予約語, 20