Cプログラムの基本

講義の目的

- システムプログラミングとは？
- なぜ、Cを使うか。
- Javaとの違い

システムプログラミング

- システムソフトウェア
  - オペレーティングシステム（OS）
  - デバイスドライバ

  などの、ハードウェアを直接操作するソフトウェア

  および、OSカーネルを直接呼び出すソフトウェア
  - libcなどのライブラリ

- これらシステムソフトウェアのプログラミングを一般的にシステムプログラミングと呼ぶ。
  - C言語や機械語が使われることが多い。

なぜCを使うか

- システムプログラミング
  - 実行環境が単純であり、OSなしで実行可能にすることが容易。
  - ハードウェアの直接操作に必要なメモリアクセスを制御可能。
  - C言語と機械語を組み合わせることが容易。
- 汎用的なプログラミング
  - 実行時のオーバヘッドが小さく、効率が良い。

Javaとの違い

- Java
  - オブジェクト指向
  - 強い型付け
  - ポインタは無い
  - GC（ゴミ集め）がある（メモリ管理はJava任せ）
  - Java VM上で実行（基本的には）
- C
  - 型付けは弱いので、何でもあり
  - ポインタで何でもできる
  - GCがない（メモリ管理は自分で気をつける）
  - 機械語にコンパイルしてCPUが直接実行

講義資料

- 前半C言語の配付資料

http://www.hpcs.cs.tsukuba.ac.jp/~msato/lecture-node/sys-prog2014

評価

- 成績は、 出席、演習の提出、中間試験、期末試験にて、評価します。
  - 中間試験　11月17日
  - 期末試験　2月9日

hello world!

```
#include <stdio.h>

main()
{
    printf("hello world!¥n");
        /* print hello world */
    return 0;
 }
```

- プログラムは"main(){" の直後から始まります。これをmain関数と呼びます。（関数とは何かについては後で解説します）
- プログラムは、書いた順番に実行されます。
- ";"で終わっている単位が文です。文を単位に実行されます。
- はじめに実行される文は、printf(“…”); です。printf関数は、画面に" "に囲まれた文字列を出力する関数です。（おなじく、関数とはなにかはあとで）
- “…”で、囲まれた部分は文字列定数といいます。（日本語をいれてもいいのですが、コンパイラ、システムによってつかえないことがあるので、英文にします）

- 文字列定数の中に¥nとありますが、これは改行を意味します。つまり、 Hello worldと出力したあと、改行を出力します。¥から始まる文字列はエスケープシークエンスといいます。たとえば、¥tはタブなどがあります。¥を出力したい場合に¥¥と書きます。

- 空白や改行は文の間にいれてかまいません。なくてもかまいません（ただ、みにくくなるので適当に改行しましょう）printfと（の間、(と“の間にも空白をいれてかまいません。しかし、これもみにくくなるのでやめましょう。ただし、printfという名前の間には空白をいれてはいけません。たとえば、 pri ntf とすると別の名前になってしまいます。

- /* */の間の部分をコメントといいます。これは空白と同じく扱われて、無視されます。

- printfの次は、return 0が実行されます。これは、main関数を終了するという文です。プログラムはCのプログラムではmainから始まりますが、main関数の実行が終了するとプログラムの実行が終わります。

- #include <stdio.h>は、printf関数を使う場合には書いておいてください。これはstdio.hを読み込むというマクロという機能ですが、いまはおまじないとおもってかいておいてください。

printf関数

```
printf("Hello!¥nthe number is %d¥n",10);
```

- 端末に文字を出力する
- ¥nは改行
- %dは文字列の,（カンマ）の後の数字を１０進数で表示しなさい、という意味

計算をさせてみる

```
#include <stdio.h>

main()
{
      int x,y;
      x = 1;
      y = x + 13;
      printf("y = %d¥n",y);
       return 0;
}
```

変数

- 変数とは、値をいれて（記憶させて）おく箱のようなもの
- 変数は（main関数の最初で）「宣言」しておかなくてはなりません
- 数学の「変数」とは違うので注意

変数への代入

- 「代入」とは、変数の箱に値をいれること

  変数の名前　＝　式　；

- ＝は、数学の＝とはちがう

- 変数が式の中に現れた場合には、変数の値を取り出すということ　　これを「参照」という

算術式

- ＋　は　足し算
- －　は引き算
- ＊　は掛け算
- ／　は割り算

変数の宣言の仕方

- 関数（main関数）の最初に宣言しておく

  変数の型　　変数名、変数名、…;

- 変数の型には、
  - int　整数
  - float　実数　（浮動小数点数）

- 名前（識別子）は、英数字から始まる名前

ここまでの内容

- プログラムの基本
- 出力　printf
- 変数、代入、式

- 教科書のLesson 3まで

入出力と分岐

Scanf関数

- 出力は　printf
- 入力は、scanf

```
scanf("%d",&整数型の変数);

scanf(“%f”,&実数型の変数);
```

- ＆をわすれないで！

```
#include <stdio.h>
main()
{
   int x;
     printf("please input ?");
     scanf("%d",&x);
     printf("input is %d¥n",x);
     return 0;
 }
```

If文

- 基本　＝　上から順に実行する
- 「判断」して、実行する文を変える

if(条件式)　条件が真の時に実行される文

もう少し複雑なif文

```
if(条件文)
    条件が真の時に実行される文
else
    条件が真でないときに実行される文
```

```
#include <stdio.h>

main()
{
   int x;
     printf("please input ?");
     scanf("%d",&x);
     if(x > 0)
          printf("input is positive¥n");
    else
          printf("input is negative or zero¥n");
     return 0;
 }
```

条件文

式1 > 式2	式1が式2よりも大きい場合に真
式2 < 式2	式1が式2よりも小さい場合に真
式1 >= 式2	式1が式2よりも大きい、または等しい場合に真
式2 <= 式2	式1が式2よりも小さい、または等しい場合に真
式1 == 式2	式1が式2と等しい場合に真
式2 != 式2	式1が式2と等しくない場合に真

複文：複数の文を実行する

- ある条件が成立した時に、実行したい文が１つの文であるとは限りません。
- 複数の文を実行したい場合があります。その場合は、複数の文を{ …} で囲みます。
- これを複文（compound statement）といいます。

絶対値を求める

```
#include <stdio.h>
main()
{
   int x;
     printf("please input ?");
     scanf("%d",&x);
     if(x < 0){
         printf(“input is negative¥n”);
         x = -x;
     }
     printf(“absolute number is %d¥n”,x);
     return 0;
 }
```

もっと複雑なif文

```
if(条件式１)
        条件式１の時、実行する文
else if(条件２)
    条件式２の時に実行する文
 else
    それ以外に実行する文
```

もっと複雑な条件式

- 論理演算子

条件式1 && 条件式2	条件式1が真かつ条件式2が真、ならば真
条件式1 || 条件式2	条件式1が真または条件式2が真、ならば真
!条件式	条件式が真ならば、偽

複雑な算術式

- 演算子には優先度がある
- *、／は、＋、－よりも先に計算される
- 算術式は、論理演算子よりも先に計算される

いろいろなデータ型

- 整数と実数があり、区別されている
- 整数　　int
- 実数　float
  - doubleというのもある
- 混ぜてつかう場合に注意。
- scanfで入力するときにちがうので注意

ここまでの内容

- 入力　scanf
- if文
- 条件式
- データ型、int 型とfloat型

- 教科書では、Lesson 5まで

繰り返しと配列

コンピュータはどのくらい速いか？！

- クロック速度がコンピュータの速さの目安
  - PCでよく使われているプロセッサ Intel IA32プロセッサファミリー
  - 最新のPentium 4のクロックは、3.6GHz
  - 数クロックに１命令づつ実行できる。
  - かりに、３クロックに１命令だと、１秒間に1G命令、つまり、1,000,000,000 = 10億命令
    - 1,000　＝　１K　　千
    - 1,000,000 = 1M　　百万

- 百万回繰り返しても、あっという間！！！
  - コンピュータのパワーの源

goto文：実行の順番を変える

- “万能”の制御文
- 何も、しないとプログラムは上から下に順に実行される。
- goto文は、ラベル文が書かれているところに制御を移す。
- あまり、推奨されない。
  - goto文を使いすぎているプログラムは、悪いプログラム

goto文

- もとにもどって繰り返す。　永遠に！

```
#include <stdio.h>
main()
{
   int x;
     x = 0;
 again:
     printf("x=%d¥n",x);
      x = x + 1;
     goto again;
     return 0;
 }
```

goto文

- 99でとめてみる。

```
#include <stdio.h>
main()
{
   int x;
     x = 0;
 again:
    if(x == 99) goto stop;
     printf("x=%d¥n",x);
      x = x + 1;
     goto again;
 stop:
     return 0;
 }
```

Goto文はなぜ悪いか？

- “万能”の制御文
- あまり、推奨されない
  - いろいろなところに飛んで、探すのが大変
  - いきあたりばったりに制御してしまうプログラムを書きがちになる。
  - プログラムの構造が見えない
- 構造が見える制御文を使おう！！！
  - while
  - for
  - do

while文：条件が成立するまで繰り返す

- *while*文は、ある条件が成立している間、文を実行するという繰り返し文

```
#include <stdio.h>
main()
{
   int x;
     x = 0;
     while(x < 100){
        printf("x=%d¥n",x);
        x = x + 1;
     }
     return 0;
}
```

while文

- 書き方

```
while(条件)
        条件が成立している間実行する文
```

- 複数の文があるときには、{ … }で囲んで、複文にする。

while文

- gotoで書き直してみれば、....

```
main()
{
   int x;
     x = 0;
 again:
     if(x < 100){
          printf("x=%d¥n",x);
       x = x + 1;
          goto again;
     }
     return 0;
 }
```

for文：0から１づつ増やして順番に

- 変数の値をある値からある値まで変化させるのはよく現れるパターン
- この時、便利なのかfor文

```
for(i = 0; i < 100; i++) { …. }
```

```
for(初期化するための式 ; 条件式 ; 繰り返し式)
        繰り返し実行する文
```

- 繰り返しする文が複数あるときは{ … }を使う。

for文

- for文で書き換えてみると、…

```
main()
{
   int x;
     for(x = 0;x < 100; x = x + 1)
        printf("x=%d¥n",x);
     return 0;
 }
```

for文

- ある変数を順に増やしていく時には、while文よりもずっと、簡単に書ける
  - 簡単　⇒　わかりやすい
  - コンパクト

- １づつ増やすという場合だけでなく、初期化、繰り返し、停止条件というバターンにあう場合には使える
  - よく現れるパターン

記述を簡単にする演算子

- 「1づつ足す」`x = x + 1` というのは、よく現れるパターン
- `x = x + 1` は、 `x++` と書くことができる。

```
for(x = 0; x < 100; x++)
        …
```

インクリメント・デクリメント演算子

変数++	ポストインクリメント。変数に1加える。但し、式としての値は1加える前の値となる。
++変数	プリインクリメント。変数に1加える。式としての値は1加えた後の値となる。
変数--	ポストデクリメント。変数を1減らす。但し、式としての値は1減らす前の値となる。
--変数	プリデクリメント。変数を1減らす。式としての値は1減らした後の値となる。

ポストインクリメント、ポストデクリメントは、元の値を使った後、インクリメントを行う

代入演算子

- 変数　演算子= 式　は、　変数 = 変数 演算子　式　とおなじ

- 例えば、

  x += 10　は、x = x + 10　とおなじ

break文とcontinue文：ループの停止と中断

- break文が実行されるとそのループの実行を直ちに実行を中断します。

- continue文は、今実行している回のループの実行する文の実行をやめて、次の回のループに移ります
  - for文の場合には、次の回が実行される前に、繰り返し式が実行されてから、つぎの回の実行されることに注意してください

do文

- while文の変形

```
do
    条件が成立している間繰り返す文
while(条件文)
```

配列とは

- 変数は、値を入れておく箱
  - でも、名前で指定しなくてはならない
- 配列とは、データを入れる箱を並べて、何番目か（インデックス）で参照するためのデータ型
  - 計算した値で、何番目かというのを指定したい
- 表を作る場合にはなくてはならない。

配列の宣言

- 宣言

基本データ型　配列の名前[要素の数]　;

- 例: 100個の整数の配列

```
int A[100];
```

- この要素の数（つまり、何個のデータが入るか）を配列のサイズという

配列の参照

- 参照

  配列名 [何番目かを指定する式]

- 注意！！！！　要素の指定は、0から数える
  - はじめの要素は、0番目

- 参照（取り出し）

```
A[i*2+1]+10
```

- 左に書くと代入

```
A[i+3] = 100;
```

配列を使ってみる

- １０個の数を入力して、合計を計算するプログラム

```
#include <stdio.h>
int a[10];
main()
{
    int i,k,s;
    printf("Please input 10 numbers?");
    for(i=0; i<10; i++){
       scanf("%d",&k);
       a[i]=k;
    }
    s = 0;
    for(i=0; i<10; i++) s += a[i];
    printf("sum is %d¥n",s);
    return 0;
 }
```

変数と配列の初期化

- あらかじめ、変数に最初に値をセットしておく（初期化）と便利なことがある
- 変数aの宣言と同時にaには10をいれておく

```
int a=10;
```

- 配列の宣言と同時に、サイズ４の配列に順番に1,3,4,1という値を入れる

```
int test[4] = {1,3,4,1};
```

今日の内容

- 繰り返しの書き方
  - while
  - for
  - break ,continue, do, … (ちょっと勉強してください)
  - (goto)　あまりつかっちゃいけない
- 配列
  - 宣言の仕方、参照、代入、初期化

関数とは

数学の関数

- 数学の関数は、パラメータに対して、ある値をマッピングするもの

$$f(x) = 2x + 1$$

- これと似ているがちょっとちがう

プログラミング言語の関数

- プログラミング言語のもっとも重要な要素の１つが関数

- 「数学の関数」と同じ、働きをさせることができる。

```
int foo(int x)
{
    return 2*x+1;
}
```

- xに２をかけて、それを返す（return 文）

関数の呼び出し

- 関数の使い方　－＞　関数呼び出し
- 呼び出しは次のように書く

```
int main()
{
        int z;
        z = foo(10);
        printf("z=%d¥n",z);
}
```

プログラミング言語での関数の役割

- 数学の関数のような働き
  - 値を入力して、値を計算する。
  - 関数　“function”
- ある特定のことをする部分をまとめておく
  - 手続き　“procedure”
  - 呼び出すことによって、その機能を使うことができる
  - printfやscanfの関数
  - 大きなプログラムを書く時には必須！！！
    - プログラムは、関数のあつまりでできている

関数の定義

- 関数定義の仕方

```
関数の返す値のデータ型　関数名（パラメータのデータ型　パラメータ名
{
        /*  あれば、変数の宣言  */
    …関数のプログラム…
    return  関数から返す値の式;
}
```

- 呼び出されると、上から順に実行される
- return文で、呼び出された先に戻る

関数の定義

- 関数のパラメータは、呼び出されたときに指定された値が代入される（セットされる）変数
  - 変数のデータ型を指定しておくこと。
- 関数のデータ型は、return文で返される値のデータ型を指定する
  - これは省略できるが、そのときにはintが仮定される。
- 関数の終わりまでいくと、呼び出された先に戻る

関数の呼び出し

- 呼び出しの形式

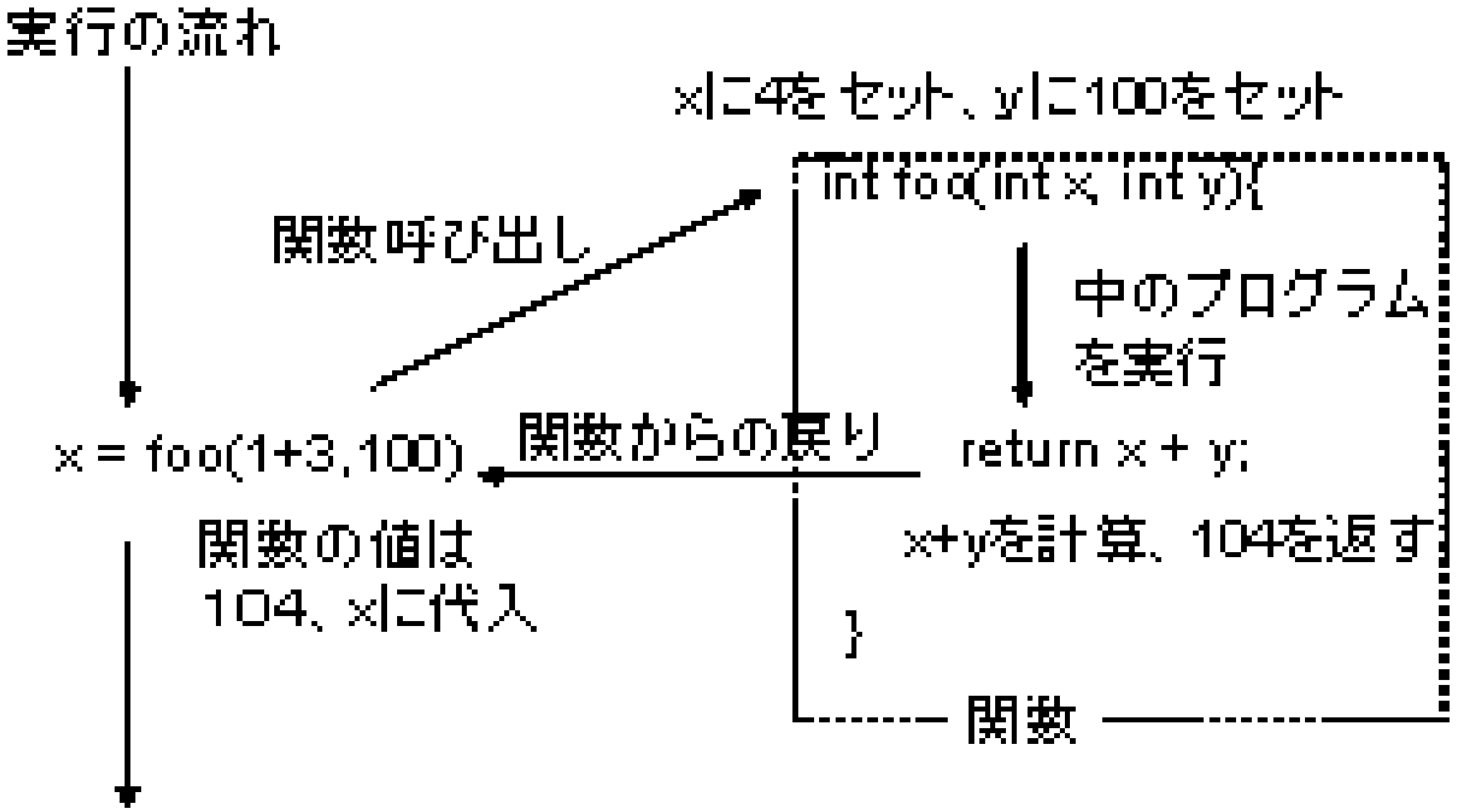
  関数名（引数の式、....）

- 式の値が計算されて、その値が呼び出しに使われる　これを引数（ひきすう）とよぶ。

- 式の一つとして使う

```
x = foo(1,y)+1;
```

関数の実行の流れ


実行の流れ
xに4をセット、yに100をセット
int foo(int x, int y){
関数呼び出し
中のプログラム
を実行
x = foo(1+3,100);
関数からの戻り
return x + y;
関数の値は
104、xに代入
x+yを計算、104を返す
}
関数

関数の宣言

- 関数の呼び出しをする場合は、その前に関数の型の宣言が必要

  関数の返す値のデータ型　関数名(パラメータのデータ型　パラメータ名、…);

- 関数定義の本体の部分を除いたもの
- 関数のプロトタイプ宣言という

まとめ

- まず、先頭にプロトタイプ宣言をしておく。
- 関数の呼び出しはその後。
- プロトタイプ宣言をしておけば、関数の本体を定義はどこでもかまいません。

例

```
#include <stdio.h>
int imax(int a,int b);
main()
{
    int x,y,z;
    scanf("%d",&x);
    scanf("%d",&y);
    z = imax(x,y);
    printf("max is %d¥n",z);
}
int imax(int a,int b)
{
    if(a > b) return a;
}
```

- 関数はある機能を一まとめにしたもの
- ４つの数の最大値を求めたい場合

```
imax(imax(1,3),imax(5,4))
```

のほうが簡単！

手続きとしての関数

- 定義の仕方

```
void 関数名(パラメータのデータ型 パラメータ名、…)
{
        /* あれば、変数の宣言 */
    …関数のプログラム…
    return;
}
```

- 関数の返す値はないので、voidにしておく
- returnには値がなくてもいい。

手続きとしての関数

- 呼び出し方は

  関数（引数、...）；

- 文として書く

いろいろな変数

- 局所変数
  - 関数の中に宣言されているもの
  - 関数の定義に書かれたパラメータ変数もその１つ
- 大域変数
  - 関数の外に宣言されているもの
- 静的変数

局所変数

- 局所変数
  - 関数の中に宣言されているもの
  - 関数の定義に書かれたパラメータ変数もその１つ
- 関数には、いろいろなプログラムが書ける
  - 局所変数は、作業領域としてつかう。
- 局所変数の値は、関数の実行が終わると無効になる。
- 違う関数の中の局所変数は、使えない

大域変数

- 大域変数
  - 関数の外に宣言されているもの
  - 関数の外に書くことができる
- 関数の間で共有するデータをいれておくもの。

大域変数の例

- 関数はいろいろな機能に名前をつけて、手続きにしておくのに便利

```
int counter;

void increment() { counter += 1; }

void decrement() { counter -= 1; }
```

- 名前をつけておくと、わかりやすくなることが多い

大域変数の悪い例

```
int t;

int foo(int x,int y)
{
     t = x + y;
     return t;
}
```

```
int goo(int x, int y)
{
        t = x-y;
        z = foo(x,y);
         … ;
        return t;
}
```

副作用(side effect)

局所的しか使わない一時的なデータにはできるだけ局所変数を使い、
必要なときだけ大域変数を使うようにしましょう。

ここまでの内容

- 関数とは
- 関数は数学の関数とはちがう
- 関数の２つの役割
  - 数学の関数と同じように、何かを計算して返す
  - 手続き、いろいろな動作をひとまとまりにしておく
- 関数の定義、呼び出し、宣言

- いろいろな変数
  - 局所変数
  - 大域変数
    - side-effect
  - （静的変数）

文字コードと文字列

文字型と文字コード

- コンピュータは「計算機」
- でも、計算だけじゃない！

- 情報を扱うためには文字も扱うことが必要

- データベース
- ワープロ
- …

文字型と文字コード

- 文字列とは、文字の列
- 文字のデータ型が文字型
- 文字型の変数の宣言

```
char c;
```

- 文字定数は、’で囲む

```
c = 'A';
```

文字の入出力

- printfの中では、%c　をつかう

```
printf("character is %c¥n",c);
```

- scanfでも同じ

```
scanf("%c",&c);
```

- 関数もある

```
c = getchar();
putchar(c);
```

文字コード

- 実際、計算機の中では文字は数字で表現されています。これを*文字コード*といいます。

- 約束ごとなので、いろいろなコード系がある
  - アルファベットと数字では、８ビット（１バイト）
    - ASCII（American Standard Code for Information Interchange ）
    - EBCDIC（Extended Binary Coded Decimal Interchange Code)
  - 日本語では、２バイト
    - JIS
    - shift JIS
    - EUC
    - UNICODE utf8

文字の比較

- 文字コードは数字なので、比較できる。
- ‘A’から’Z’、’a’から’z’，‘0’から’9’は順番に並んでいる

- 文字型変数cが大文字かどうかをみるには

```
c >= 'A' && c <= 'Z'
```

- 数字かどうかみるには？
- 小文字かどうかみるには？

文字列の変換

- ‘A’から’Z’、’a’から’z’, ‘0’から’1’は順番に並んでいる

- 小文字から大文字に変換するには

```
c = c - ‘a’ + ‘A’;
```

- 数字を数値に直すには

```
i = c - ‘0’;
```

文字列とは

- 文字列は文字の列なので、実は文字の１次元配列

```
char s[10];
```

- 文字列定数は、”で囲んだ文字列
- 入出力は%s

```
printf(“string is %s¥n”,s);
scanf(“%s”,s);
```

ここで、&がつかないことに注意

文字列とは

- 文字列の初期化

```
char s[10] = “ABC”;
```

- だだし、代入はできない

```
s = “ABC”;
```

はだめ！

文字列とは

- もう一つの条件：
  文字列とは「コード0で終わる文字列」
- コード0 (‘¥0’と書く)は文字列の最後を示すコード
- 10文字の文字列sに文字列”ABC”を入力すると、配列の最初の4要素に ‘A’,‘B’,‘C’,‘¥0’,が入る
- printfの%sでは、’¥0’までの文字を出力します。
- 文字列のための文字配列は文字列の長さ+1の要素が必要であることに注意

文字列を使った例

- 小文字を大文字に直して出力するプログラム

```
main(){
        int i;
        char s[10];
        printf("please input string?");
        scanf("%s",s);
        for(i = 0; s[i] != '¥0'; i++)
          if(s[i] >= 'a' && s[i] <= 'z')
              s[i] = s[i] - 'a'+'A';
        printf("result = %s¥n",s);
        return 0;
}
```

文字列のコピー

- *文字列は最後の文字が’¥0’の文字の配列であることをおぼえておくこと*

```
for(i = 0; s1[i] != '¥0'; i++)
        s2[i] = s1[i];
s2[i] = '¥0';
```

配列の引数

- 関数宣言

```
関数の返す値のデータ型
        関数名 (配列パラメータのデータ型　配列のパラメータ名 [ ] 、…)
{
    …後は同じ…
}
```

- 呼び出し

```
関数名(引数の配列名、…)
```

- 文字列も文字の配列

配列を引数に持つ関数

- 配列の加算をする関数

```
int sum(int a[],int n)
{
   int i,s;
   s = 0;
   for(i = 0; i < n; i++) s+= a[i];
   return s;
}
```

文字列をコピーする関数

- 文字列も文字の配列

```
void string_copy(char s2[],char s1[])
{
   int i;
   for(i = 0; s1[i]!='¥0'; i++)
          s2[i] = s1[i];
   s2[i] = '¥0';
}
```

文字列のライブラリ関数

- 便利な関数として、文字列の*コピー関数strcpy*、*連結strcat*、*辞書順比較 strcmp*があります
- strcpy(文字列1,文字列２)は、文字列１に文字列２をコピー
- strcmp(文字列１、文字列２)は、文字列１と文字列２を辞書順に比較して、前であれば、-1、同じであれば、0、後ろであれば、1を返します。たとえば、同じ文字列かどうかを判定するには、0と比較します。

```
strcmp(s1,s2) == 0
```

ここまでの内容

- 文字型と文字コード
- 文字列
  - 文字の配列
  - ０で終わる
- 配列を引数にもつ関数
- 教科書
  - ２．３　文字と数値
  - ７．６　文字列と配列

２次元配列

配列の使い方

- 配列とは、データを並べて、何番目か（インデックス）で参照するためのデータ型
- 例：　数列

```
main(){
  int a[11];
  int i;
  a[0] = 1;
  a[1] = 1;
  for(i=2; i <=10; i++)
     a[i] = a[i-2]+a[a-1];
  printf("a10 is %d¥n",a[10]);
}
```

配列の使い方

- 集合

```
#include <stdio.h>
main(){
 int A[10];
 int i,flag;
 for(i=0; i<10; i++)
   scanf("%d",&A[I]);
 flag = 0;
 for(i=0; i<10; i++)
   if(a[i] == 3){
     flag = 1;
     break;
   }
 if(flag) printf("3 is found¥n");
 else printf("3 is not found¥n");
}
```

配列の使い方

- 表
- 升目の表を考える
  - １０行２０列

```
int T[10*20];
```

  - i行j列を取り出す

```
x =T[i*20+j]
```

２次元配列

- 表をわかりやすく書くための仕掛け
- 宣言

基本データ型　配列の名前[行のサイズ][列のサイズ];

```
int T[10][20];
```

- 参照

配列名[行の位置][列の位置]

```
x=T[i][j];
```

要素を参照するときにはいちいち列や行のサイズを気にする必要がなくなります

２次元配列

- ２次元以上もあり

基本データ型　配列の名前[１次元目のサイズ][２次元目のサイズ][３次元目のサイズ]…;

```
int T3[10][20][3];
```

文字列の配列

- 文字列が１次元配列なので、文字列の配列は２次元配列
- 宣言

```
char 文字列の配列の名前[配列のサイズ][文字列の長さ];
```

```
char S[10][20];
```

- 文字列の長さは、正確にはこの配列に格納する文字列の最大の長さ
- S[i][j]と書くとiは何番目の文字列か、jは、その文字列の文字の位置をすることになり、その文字が参照されます

文字列の代入（コピー）

- 文字列には代入はできない！
- その代わり、コピーする
- コピーにはstrcpy関数をつかう

```
char S[10][20];
char w[20];
      ….
strcpy(w,s[i]);
```

ここまでの内容

- 配列のいろいろな使い方
- ２次元配列
  - 表
- 文字列の配列
  - 文字列の配列は２次元配列
- 教科書
  - ７．５