はじめに

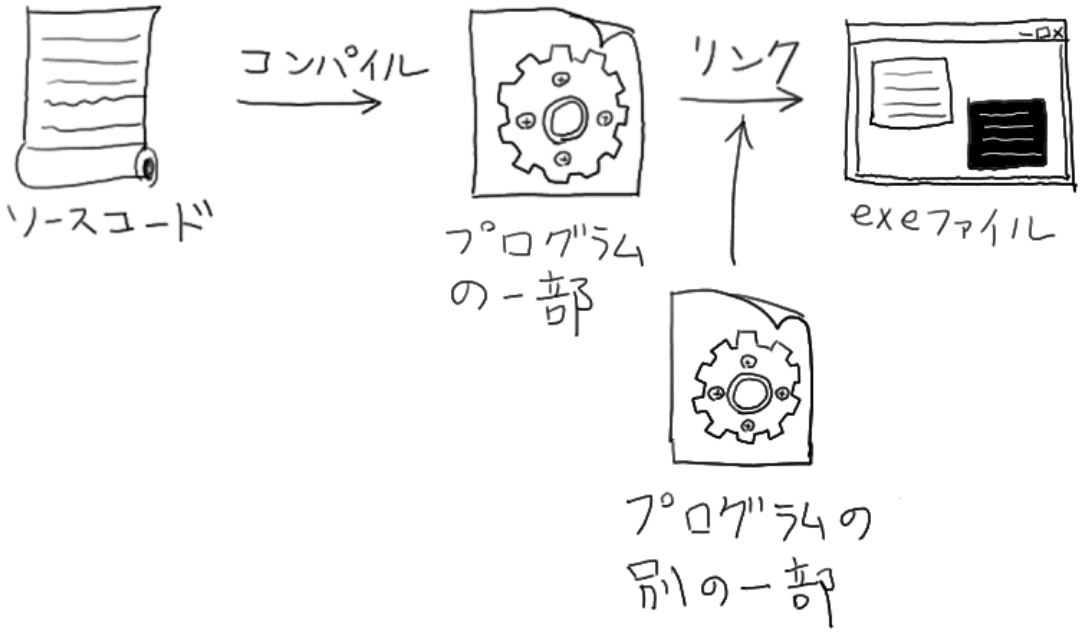
この文章はプログラミング研究会による、新入生向け講座資料です。プログラミングを学ぶ際、苦しむ原因の一つは「覚えることが多くて習得まで時間がかかる」ということです。そこで、この講座資料では「他人に見せられるプログラムを作れるようになるのに必要最低限の知識を最短で身につける」ことを意識して C 言語を解説していきます。

プログラムができあがるまで

言語によって異なりますが、C 言語の場合は、

ソースコード→コンパイル→リンク→プログラム完成

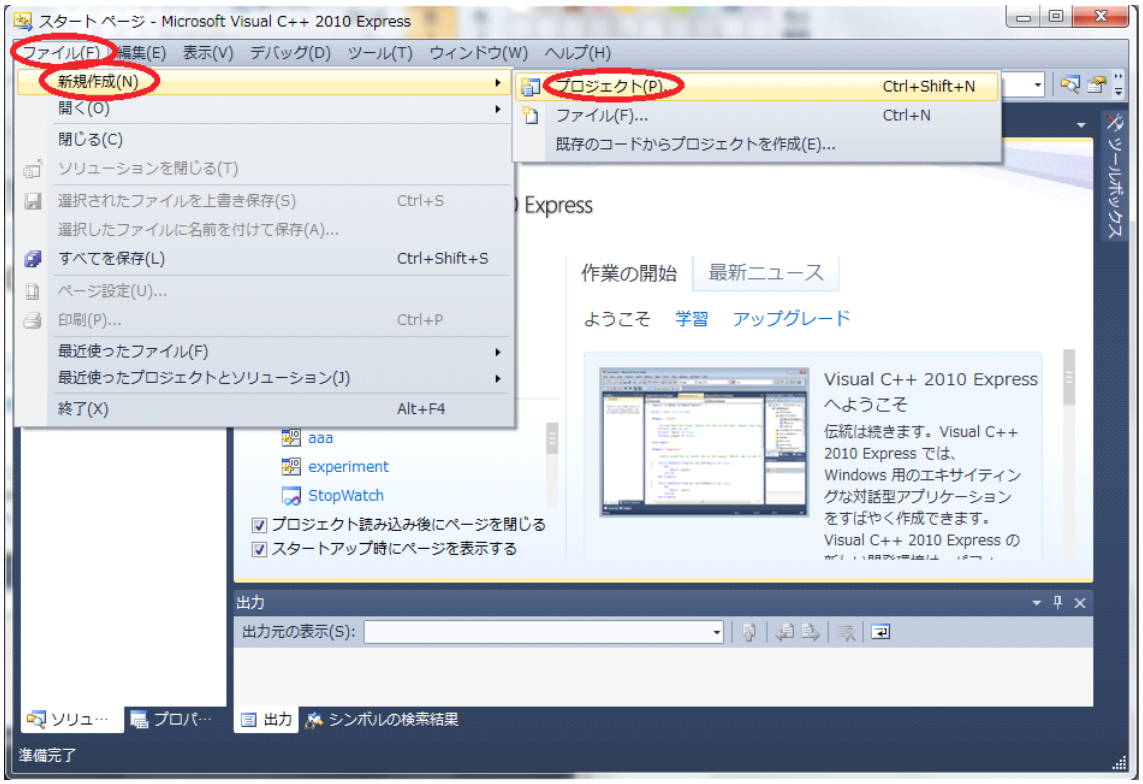
という手順でできあがります。ソースコードは私たちがこれから書いていくプログラムの設計図です。ソースコードができあがったらコンパイラというものがソースコードを解読して機械語に変換します（これをコンパイルという）。ソースコードをコンパイルする際、他のファイルを参照していたら、リンカと呼ばれるものが、そのファイルをプログラムに組み込みます（これをリンクという）。コンパイル、リンクが終わったらプログラムが完成して exe ファイルができあがるわけです。コンパイラ、リンカは Visual C++ などの開発環境をインストールすると大抵自動的に付いてきます。



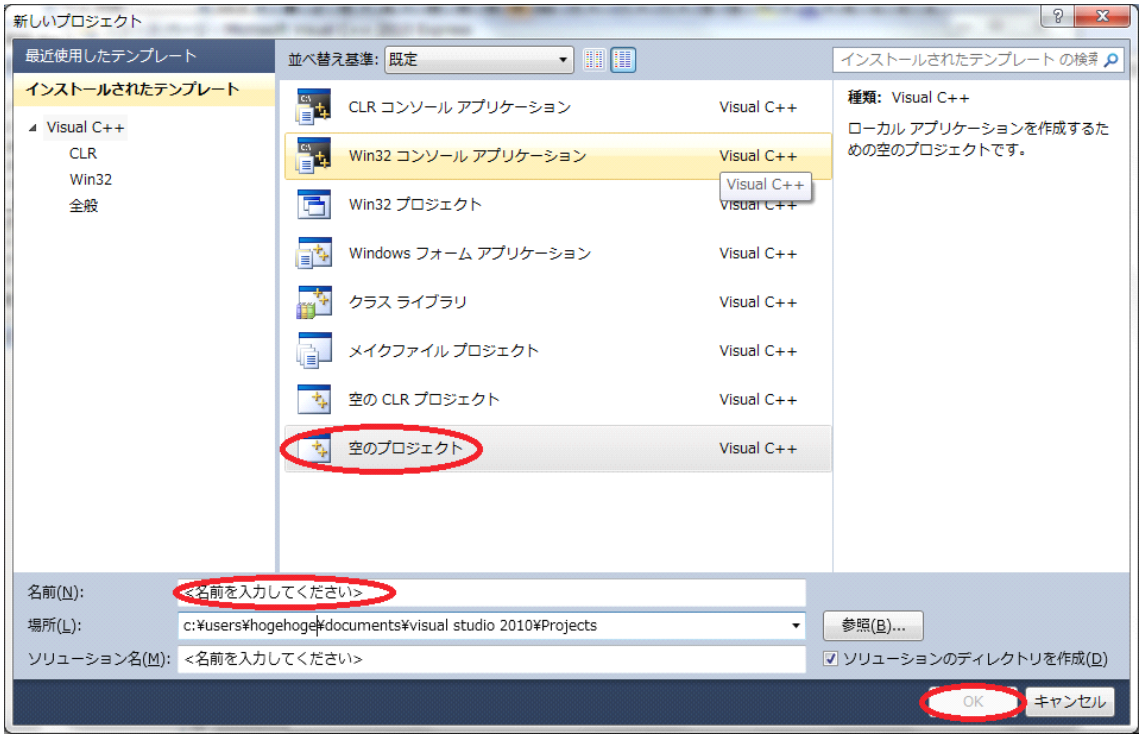
インストールと新規作成

プログラミングをするには、開発環境が必要となります。
http://www.microsoft.com/japan/msdn/vstudio/express/
より VisualC++2010 ExpressEdition をダウンロードした後、画面に従ってインストールしてください。

インストール後起動させるとスタートページが立ち上がります。

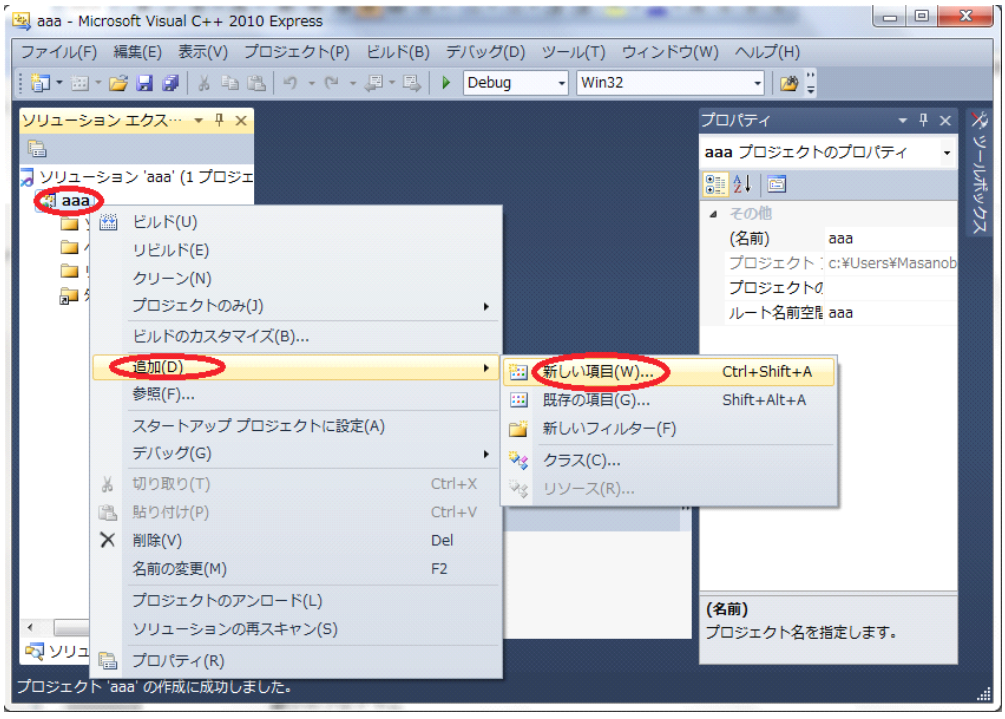


まずはプログラムを作るためにプロジェクトを立ち上げる必要があります。プロジェクトを立ち上げるには、「ファイル」→「新規作成」→「プロジェクト」を選びます。

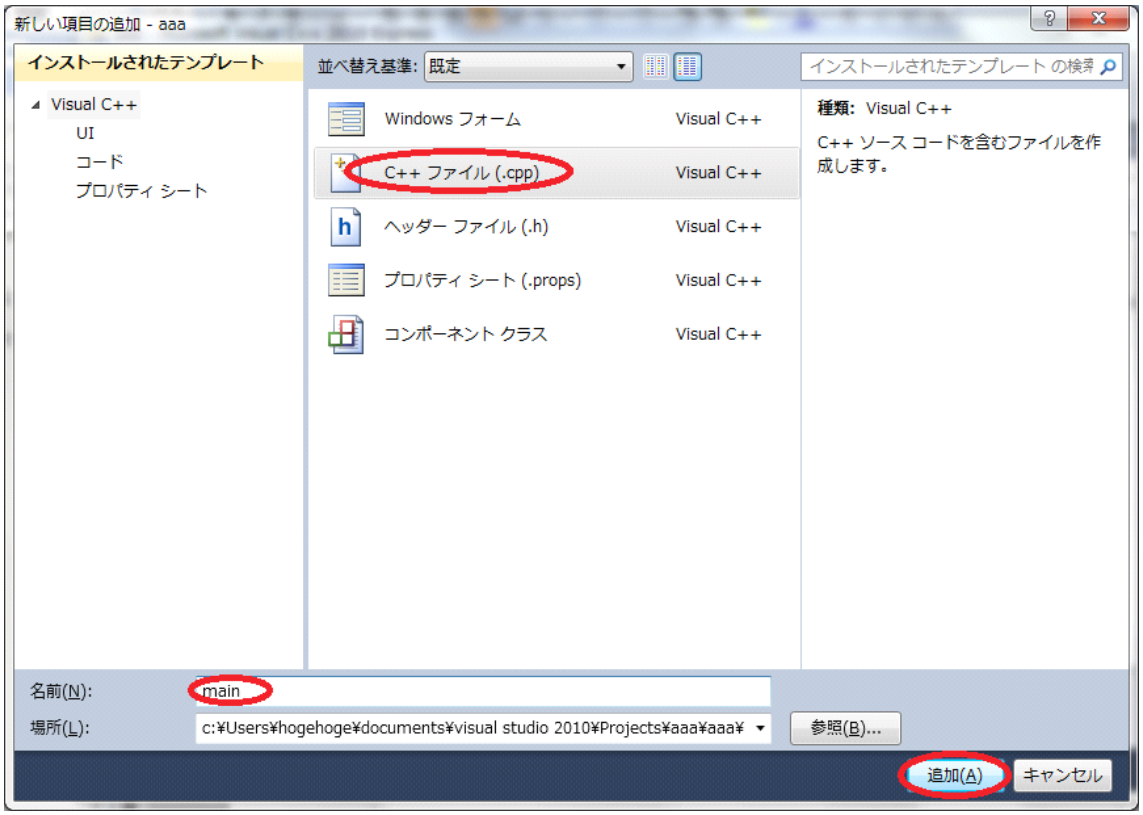


次に、左の「インストールされたテンプレート」から「Visual C++」または「全般」を選んで、真ん中から「空のプロジェクト」を選択。「名前」にこのプロジェクト名を自由に入力した後「OK」を押してください。この例ではプロジェクト名は適当に「aaa」と名付けました。

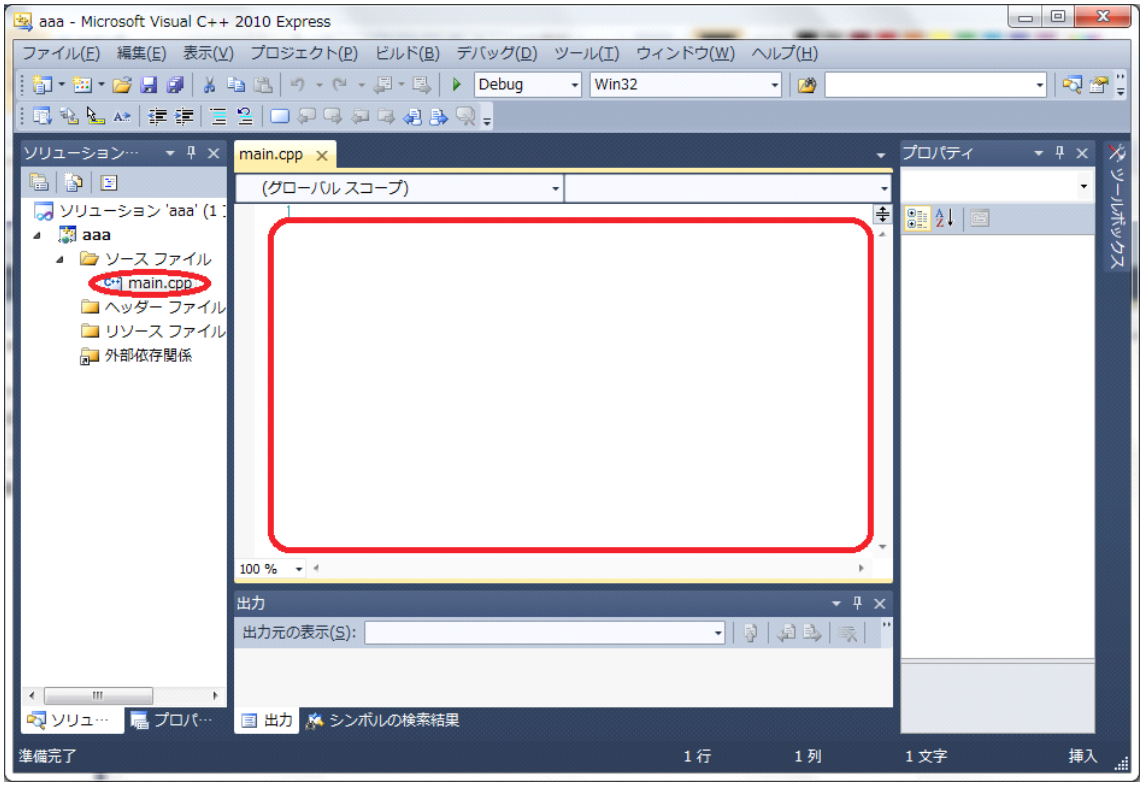
ここまできたら下の画面が出てくると思います。



次に、実際にコードを書き込んでいくソースファイルをこのプロジェクトに追加する必要があります。
ソリューションエクスプローラ内の、先ほど付けたプロジェクト名が書いてあるところの上で右クリックをし、「追加」→「新しい項目」を選びます。ソリューションエクスプローラが表示されていない人は、メニューバーから「表示」→「ソリューションエクスプローラ」を選べば左に表示されます。
そうすると、追加するファイルの種類を選ぶ画面が現れます。



左の「インストールされたテンプレート」から「Visual C++」または「コード」を選び、真ん中から「C++ファイル(.cpp)」を選択し、名前に好きなソースファイル名を入力してください。今回の例では名前を「main」としておきました。そして最後に「追加」を押します。そうするとプロジェクトに「main.cpp」が追加されたはずです。



ソリューションエクスプローラ内の「main.cpp」をダブルクリックしたら「main.cpp」と書かれたタブが現れます。そのタブの下の広いエディタ部分にソースコードを書いていきます。これでコードを書くための準備は終わりました。

最小のプログラム

まずは最も小さいC言語のプログラムを見てみましょう。

最小のプログラム：

```
int main()   //ここからプログラムの開始
{
    return 0;    //返し値と関数の終了
}
```

実行

上のコードをエディタに書き込んだ後はデバッグをする必要があります。デバッグをするには、メニューバーから「デバッグ」→「デバッグ開始」を選び、「ビルドしますか？」というダイアログが出るので、「はい」を選ぶとビルドしてデバッグが開始されます。または、F5 キーを押すことによってもデバッグを開始することができます。ビルドに失敗した場合は、書いたコードのどこかに間違いがあるということなので、修正して再びビルドする必要があります。以下、プログラムを実行する時は以上の手順で行ってください。

実行した結果、一瞬だけコンソール画面が出てすぐ消えると思います。以下、ソースコードの解説です。
int main() とは、ここからプログラムが開始されることを意味します。プログラムはこの中の一番上から下に向かって一つずつ実行されていきます。
そして、この int main() の中身は { } によって囲まれた部分です。この中身を上から順番に実行していき、最後まで到達したらこのプログラムは終了となります。
しかし、最後まで到達しなくても終わらせる方法があります。それが return 0 です。これを書き込むと、最後まで到達しなくてもプログラムを終了させることができます。
また、プログラムの中の命令ひとつにつき、；で区切る必要があります。

ちなみに、プログラムを書いていると自分の書いたコードがどうなっているのか分からなくなることがあります。そういうときのために // を用いてコメントを書き、自分の書いたコードに説明を加えることができます。 // 以降の一行だけは実行されることはありません。
複数行にわたってコメントを残したい場合は、/* */ の間にコメントを書けばその間は実行されることはありません。

```
/* ここの部分はコメント部分です。実行されることはありません */
```

また、改行とスペース、タブも同様に実行されることはありません。これらを使って自分の見やすいようにコードを書いていってください。

今後ソースコードを書いてデバッグする時、コンソール画面が一瞬だけ現れてすぐ消えるようでは正しく動作しているのか確認できません。すぐにコンソール画面が消えるのを防ぎたい時は、 return 0 の直前に、

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

の 2 行を付け足してください。コンソール画面を消す時は右上の ×ボタンを押すか、エンターキーを押してください。

課題：最小のプログラムを入力し、実行せよ。

HelloWorld

それでは、実際にコンソール画面に文字を表示させてみましょう。

HelloWorld プログラム：

```
#include<stdio.h>
int main()
{
    printf("HelloWorld");
    return 0;
}
```

実行結果：

```
HelloWorld
```

最小のプログラムの時と同じように、「デバッグ」→「デバッグ開始」または F5 キーで実行してください。

#include<stdio.h> は今のうちはおまじないだと思ってください。これをプログラムの最初に書くことで printf 関数という、文字を出力する命令が使えるようになります。しばらくの間このおまじないを使っていきます。

そしてプログラムのなかの printf("HelloWorld"); は、「HelloWorld という文字列を画面に表示しなさい」という命令です。出力したい文字列はダブルクォーテーション（""）またはシングルクォーテーション（''）で囲ってあげる必要があります。

また、文字列を改行して複数行にわたる出力をすることもできます。

複数行出力プログラム：

```
#include<stdio.h>
int main()
{
    printf("1line¥n");
    printf("2line¥n");
    printf("3line¥n4line¥n");
    printf("5line_1");
    printf("5line_2¥n");
}
```

出力結果：

```
1line
2line
3line
4line
5line_15line_2
```

¥n は改行コードを意味し、出力したい文字列に含ませると、それが改行する箇所になります。逆に、 ¥n を入力しない限り、出力結果が改行されることはありません。

出力関数は printf 関数の他にも、puts 関数があります。puts 関数の中には文字列を入れます。

puts 関数プログラム：

```
#include<stdio.h>
int main()
{
     puts("abcde");
     puts("fghij");
     return 0;
}
```

実行結果：

```
abcde
fghij
```

puts 関数の場合は printf 関数と違って、自動的に改行が最後に付加されます。

課題："I started C language!"と画面に出力せよ。

変数

文字列や文字、数値計算などを繰り返してプログラムはできあがっていきますが、この文字列や文字、数値をとっておいて後で使いたい場面があります。そんなときは変数という、データを入れる箱のような物を利用します。

変数プログラム：

```
#include<stdio.h>
int main()
{
    //変数に数値や文字を代入
    int num = 10;    //整数値を代入
    float num2 = 2.1;    //7 桁までの小数値を代入
    double num3 = 3.1111111111;    //15 桁までの小数値を代入
    char chr = 'a';    //文字を代入

    printf("int type: %d¥n",num);
    printf("float type: %f¥n",num2);
    printf("double type: %f¥n",num3);
    printf("char type: %c¥n",chr);
    return 0;
}
```

出力結果：

```
int type: 10
float type: 2.1
double type: 3.1111111111
char type: a
```

変数を使う時はまず変数を用意する必要があります。変数を用意するには、

[変数の型] [変数名];

と記述することでできます。この変数を用意することを「宣言」と言います。例えば、"num"という名前の整数型変数を宣言するには"int num"と書きます。変数の型については後述します。

変数を宣言したら次に、その変数に具体的なデータを代入する必要があります。この処理を「定義」と言います。変数の定義をするには

[宣言した変数名] = [データ]

というようにイコールを使って行います。例えば、先ほど宣言した整数型変数 num に 10 を代入するには、 num = 10 のようにします。また、宣言と定義を同時にやって、 int num = 10 のようにすることもできます。

上記のプログラムを見てみましょう。上記のプログラムでは num, num2, num3, chr という名前の変数を宣言、定義しています。

変数名の前に付く int, float, double, char は何が違うのかというと、入れられるデータの種

類が違います。これを変数の型と言います。以下に主要な変数の型一覧を表示します。

名前	入れられるデータ	入れられるデータの例
int	整数	10, -2, 0
float	7 桁までの小数	1.0, -2.2
double	15 桁までの小数	0.00000000001,
char	1 文字	'a', '¥n'
char 配列	文字列	"aaaaa¥n"

float と double は、精度の高い計算の時は double が必要ですが、普段プログラミングする場合は float で十分の場合が多いです。char 配列については後で説明します。

つづいて、printf 関数の使い方についてです。以前使った時は括弧の中に出力したい文字列を入れるだけでしたが、今回は少し違っています。 %d や %f などを printf の中の文字列に含ませると、そこを別の数値や文字などで置き換えることができます。今回の例を見てみると、コード中の printf("int type: %d", num); というのは、「%d と書かれているところに、変数 num に代入されている整数値を入れて画面に出力しろ」ということを意味しています。同様に、printf("float type: %f¥n",num2) とは、num2 に代入されている小数値を %f に入れて出力しろということを意味しています。 %d, %f, %c は何が違うかというと、入れられるデータの種類が異なります。この %d といったものを「フォーマット指定子」と呼びます。printf で使われるフォーマット指定子を以下に示します。

指定子	入れられるデータ	相性の良い型
%d	整数値	int, char
%f	小数値	float, double
%c	文字	char
%s	文字列	char 配列
%u	unsigned 整数型	unsigned int

unsigned による正の数

int 型と char 型のみ、変数宣言の際に先頭に unsigned を付け加えることができます。unsigned を付け加えることで、負の数を代入することはできなくなりますが、その分正の数の上限が 2 倍に増えます。正の数しか使わないと分かっているときに便利です。

名前	代入可能なデータ範囲
int	-2147483648～2147483647
unsigned int	0～4294967295

char	-128～127
unsigned char	0～255

unsigned のプログラム：

```
#include<stdio.h>
int main()
{
    int num = 4294967295;
    unsigned num2 = 4294967295;
    printf("%d¥n", num);
    printf("%u¥n", num2);
    return 0;
}
```

出力結果：

```
-1
4294967295
```

num を代入した方は-1 になり、num2 を代入した方は正しい表示になっています。

課題：変数を作り、10 と 3.2 を代入してから画面に出力せよ。

配列

変数は 1 個ずつだけではなく、何個も連なるようにして宣言することができます。これを配列と言います。配列を宣言する時は変数名の後に [] をつけ、その中に用意する配列の数を入れます。この用意する配列の数を要素数と言います。

配列の型 配列名[要素数];

配列プログラム：

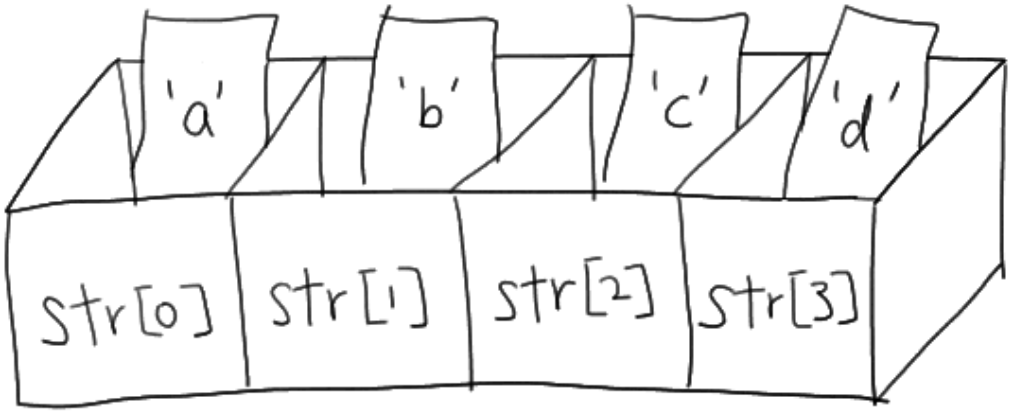
```
#include<stdio.h>
int main()
{
	int num[5] = {1,2,3,4,5};  //要素数5の整数型配列の宣言時定義
	int num2[3];       //整数型変数を3つ宣言
	char str[80] = "abcde";    //文字型配列80個。代入は79文字までOK。

	num2[0] = 1;
	num2[1] = 2;
	num2[2] = 3;
	//num2[3] = 4;    範囲外アクセス


	printf("%d %c %s",num2[1],str[3],str);
	return 0;
}
```

実行結果：

```
2 d abcde
```



配列を宣言した時に同時に定義をするには、{ } で囲って中にカンマ区切りでデータを入れてあげます。また、文字型配列に限り、"abcde" のようにダブルクォーテーションを付けることで文字列をそのまま代入することができます。ただし、文字列をそのまま代入できるのは宣言と同時に定義する時だけであり、宣言した後に間を置いて文字列を代入することはできません。代わりに strcpy 関数を使いましょう。

```
#include<string.h>
int main()
{
     char str[80];
     //str = "abcde";    このようにすることはできない！
     strcpy(str, "abcde");
     return 0;
}
```

なお、文字配列に代入できる文字数は、[宣言した要素数]-1 です。これはなぜかというと、文字列の一番最後には、"¥0" という文字列の終わりを表す文字が自動的に入るためです。

配列の個々の要素にアクセスするためには、変数の後ろに [] をつけて中にアクセスしたい番号の要素の数を入れます。また、配列にデータを代入、もしくは参照する時、その要素は 0 から始まります（例: int num[0] など）。そのため、代入、参照する時の要素の上限は、[要素数]-1 となります。

また、2 次元配列という物を作ることもできます。

配列の型 配列名[要素数 1][要素数 2]

2 次元配列プログラム：

```
#include<stdio.h>
int main()
{
        int num[2][2];
        num[0][0] = 0;
        num[0][1] = 1;
        num[1][0] = 2;
        num[1][1] = 3;

        printf("%d %d %d %d",num[0][0],num[0][1],num[1][0],num[1][1]);

        return 0;
}
```

さきほどの配列は [ ] がひとつだったのにたいし、今回は [ ] が変数のあとに 2 つくっついています。イメージ的には、1 次元配列が線で、2 次元配列が平面のように考えると良いです。また、あまり使われませんが 3 次元配列も作ることができます。

課題：文字列"HelloWorld"を文字型配列に代入した後、6 文字目の"W"だけ取り出して画面に出力せよ。

計算の仕方

プログラムを作るときは数値の計算が必須です。計算の仕方を早速見てみましょう。

計算プログラム：

```
#include<stdio.h>
int main()
{
    int num = 5;
    int num2 = 3;

    printf("足し算: %d¥n", num+num2);
    printf("引き算: %d¥n", num-num2);
    printf("かけ算: %d¥n", num*num2);
    printf("割り算: %f¥n", (float)num/num2);
    printf("剰余: %d¥n", num%num2);
}
```

出力結果：

```
足し算: 8
引き算: 2
かけ算: 15
割り算: 1.666667
剰余: 2
```

計算する際は以下の演算子を使います。

+	足し算	-	引き算
*	かけ算	/	割り算
%	余り		

また、今回割り算した際、割り切れないため、printf 関数で出力するときに float 型に変換し

ました。変換についてはまた後で説明します。

課題：11*22*33*44*55 を計算して画面に出力せよ。また、11223344 を 567 で割った余りを計算して出力せよ。

キーボード入力

C 言語ではキーボードから入力を受け取る機能がはじめから付いています。キーボードからの入力を標準入力といいます。早速プログラムを見てみましょう。

標準入力プログラム：

```
#include<stdio.h>
int main()
{
        char str[80];
        puts("文字を入力してください。");
        fgets(str,80,stdin);
        puts(str);
}
```

これを実行すると入力した文字が出力されるはずです。入力には fgets 関数を使用します。第 1 引数には入力した文字列を格納する変数を、第 2 引数には最大で何文字入れるのかを、第 3 引数には、標準入力の場合は stdin を入れます。標準入力ではない場合については、しばらく後の「ファイルの利用」でやります。とりあえず、キーボードから入力を受ける時は stdin を入れるんだなと覚えておいてください。第 2 引数は普通、配列の数と一致させます。

しかし上の例は文字しか受け取ることができません。例えば、10 という数値を代入したつもりでも内部では"10"という文字として受け取られます。
そこで、数値は数値として、文字は文字として入力を受け取るための関数が scanf 関数です。

scanf 関数プログラム：

```
#include<stdio.h>
int main()
{
        int num;
        char chr;
        char str[80];

        scanf("%d %c %s",&num,&chr,str);
        printf("入力した数値は%d¥n",num);
        printf("入力した文字は%c¥n",chr);
        printf("入力した文字列は%s¥n",str);

        return 0;
}
```

入力内容：

```
10 a abcde
```

出力結果：

```
入力した数値は 10
入力した文字は a
入力した文字列は abcde
```

scanf 関数は printf 関数とほぼ同じです。第 1 引数にフォーマット指定子を含んだ文字列を記述し、第 2 引数以降読み取ったデータを格納する変数を入れます。ただし注意なのは、第 2 引数以降、文字配列以外は変数名の前に ＆ をつけていることです。これをつけなければ予期せぬ動作をするので必ず付けるようにしてください。float 型や double 型も同様です。文字配列だけは ＆ をつけないで、配列名を記述するだけで格納できます。
また、入力する時は scanf 関数の第 1 引数に書いたとおりに入力してください。今回の例では scanf 関数の第 1 引数にて、フォーマット指定子の間に半角スペースが入れてあります。そのため、入力するときも入力データ間に半角スペースをはさんであります。もし scanf("%d,%c,%s",&num,&chr,str); と記述したなら、入力データの間にはカンマ(,)を入れな

ければなりません。

課題：2 つの数字を入力して足し合わせ、計算結果を表示するプログラムを作成せよ。

関数

プログラムを書いていくと、同じ処理をすることが多くなります。こういった場合、何回も同じ処理を書くのは面倒になってきます。そこで、関数という物を使えば処理をまとめることができます。関数を定義する時は、

```
[戻り値の型] [関数名](引数 1, 引数 2, ...)
{
        //処理内容を記述
         return [戻り値];        //戻り値がない場合は書く必要なし
};
```

といった感じに書きます。
そして利用する時は、

```
[戻り値と同じ型の変数] = [関数名](引数 1, 引数 2, ...);
```

といった感じに記述します。戻り値がない場合は、

```
[関数名](引数 1, 引数 2, ...);
```

とだけ記述して使用します。

関数プログラム：

```
#include<stdio.h>
//関数定義
int funcsum(int n, int n2) //戻り値ありの関数定義
{
        return n+n2;
}
void display(int n)        //戻り値なしの関数定義
{
        printf("引数に入れられた数字は%dです。¥n", n);
}

int main()
{
        int num;
        num = funcsum(2,3);     //関数の使用と、整数型の戻り値。戻り値は代入で格納。
        display(num);
        return 0;
}
```

出力結果：

```
引数に入れられた数字は 5 です。
```

int funcsum(int n, int n2); を見てみましょう。先頭に付いている int は戻り値の型です。この関数の中を見てみると、一番最後に "return n+n2" と書いてありますよね。関数は、return を使ってその内部で処理した結果を関数の呼び出し元に返すことができます。そのときに返すデータを戻り値といい、その型を戻り値の型といいます。今回 funcsum 関数の戻り値の型は int 型なので、関数定義の際、一番最初に int をつけてあげます。戻り値がない場合は関数定義の際に void とつけてあげます。

funcsum というのは関数の名前です。これはプログラマーが自由に名付けることができます。ただし、数字や一部の記号から始めることはできません。

次に、funcsum 関数の引数が入っている括弧の中を見てみましょう。 (int n, int n2) というのは、「int 型のデータを 2 つくれ。そうすれば関数内で処理してデータを返すから。」ということを意味します。

ちなみに、今までプログラムの開始地点としてずっと使ってきた int main() というのも、関数なのです。「引数はいらないけど、int 型の返り値を返すよ」ということを意味する関数です。そのため最後に必ず return 0; を付けていたのでした。この関数は少々特殊な扱いをされますが、ここでは説明は省略します。

課題：2 つの整数型データを受け取り、足し算して返す関数を作れ。

制御式

今まで見てきたのは、上から下まで順番にひとつずつ処理していくプログラムの書き方でした。しかし、時には流れを分岐させたり、繰り返したりしたくなるときがあります。そこで、条件式の出番です。

制御式（if 文）

if 文は流れを分岐させる制御式です。「こんな条件の時はこういう処理をしたい」という時に使います。if 文は以下の文法のように記述します。

```
if( 条件式 ){ 条件を満たす時に実行する処理内容 }
```

早速プログラムを見てみましょう。

制御式 if のプログラム：

```c
#include<stdio.h>
int main()
{
	int num=3;
	if(num==3)
	{
		puts("if文のなかだよぅ3");
	}
	if(num==4)
	{
		puts("if文のなかだよぅ4");
	}
	return 0;
}
```

出力結果：

```
if 文のなかだよぅ 3
```

一つ目の if 文を見てみると、 if(num==3) の部分で分岐し、処理する内容は { } 内の部分となります。今回変数 num に入れた数は 3 なので、この分岐の中身を実行したということになります。一方、 if(num==4) は条件にあてはまらないので括弧内の処理はされずに流されます。
この == を比較演算子といいます。比較演算子には以下のものがあります。

==	同じ値かどうか	=>	左辺は右辺以上か
!=	異なる値かどうか	=<	左辺は右辺以下か
>	左辺は右辺より大きいか	<	右辺は左辺より小さいか

比較した結果、真だったら true を、偽だったら false を返します。
また、if の括弧の中の条件は論理演算子というものを用いて複数設定することができます。

&&	かつ	||	または

if 文プログラム 2：

```
#include<stdio.h>
int main()
{
	int num = 3;
	int num2 = 4;
	//numが3で、かつnum2が4ならば
	if(num==3 && num2==4)
	{
		puts("num==3 && num2==4");
	}

	return 0;
}
```

出力結果：

```
num==3 && num2==4
```

課題：整数型変数 num, num2 を作り、num が 5 かつ num2 が 6 のとき「条件に一致しています」と出力するプログラムを作れ。

制御式（if-else 文）

if 文を使えば条件を分岐させることができました。しかし、これだけでは分岐は 1 つしか作れません。そこで、複数の分岐を作るために if-else 文を使います。if-else 文の文法は以下のようになっています。

```
if( 条件式 1 )
{
        条件式 1 が真の時実行する処理内容
}
else if( 条件式 2 )
{
        条件式 1 が偽で条件式 2 が真の時実行する処理内容
}
else
{
        条件式 1 も 2 も偽の時実行する処理内容
}
```

複数分岐プログラム：

```
#include<stdio.h>
int main()
{
        int num=4;

        if(num==3)
        {
                puts("num==3");
        }
        else if(num==4)
        {
                puts("num==4");
        }
        else
        {
                puts("どれでもなし");
        }

        return 0;
}
```

実行結果：

```
num==4
```

課題：整数型変数 num, num2 を作り、「num==4」のとき、「num2==4」のとき、「num==4 && num2==4」のときで条件分岐するようなプログラムを作れ。

制御式（for 文）

続いての制御式は for です。これは繰り返しをしたい時に使用します。

```
for( 初期設定式; 条件式; 繰り返し時設定式 )
{
        繰り返し処理をしたい内容
}
```

for 文による繰り返しプログラム：

```
#include<stdio.h>
int main()
{
        for(int i=0; i<5; i++)
        {
                printf("%d¥n",i);
        }
        return 0;
}
```

出力結果：

```
0
1
2
3
4
```

for の丸括弧 ( ) の中身を見てみましょう。はじめの int i=0 というのは、「変数 i を用意し、この変数を 0 で初期化せよ」ということを意味します。以下、条件式、繰り返し時設定式はこの初期設定式で使用した変数 i を使います。セミコロンで区切った二つ目の i<10 というのは、「変数 i が 5 未満の間は for 文の中身を繰り返せ」ということを意味しています。そして、セミコロンで区切った三つ目の i++ というのは、「ループを一回繰り返すごとに変数 i をインクリメントせよ」ということを意味しています。つまり、この for 文は、まず始めに整数型変数 i を 0 で初期化し、ループを繰り返すごとに変数 i の値をインクリメントしながら、i の値が 5 未満の間はずっと処理を繰り返していることになります。

インクリメントとは、整数型変数にのみ使えるもので、値を 1 増やす働きをします。 i++ は i+=1 または i=i+1 と全く同じ働きをします。他にもデクリメントという、 i-- （ i-=1 または i=i-1 と同じ ）というものもあります。デクリメントは値を 1 減らす役割をします。

課題：0～20 までの偶数のみを出力するプログラムを for 文を使って作成せよ。

for 文の利用

for 文は配列との相性が非常に良いです。1 次元配列、2 次元配列との組み合わせの例を見てみましょう。

for 文と配列プログラム：

```c
#include<stdio.h>
int main()
{
	int num[3];
	int num2[3][3];

	puts("1次元配列とfor文の組み合わせ");
	for(int i=0; i<3; i++)
	{
		num[i] = i;
		printf("%d,",num[i]);
	}
	printf("¥n");

	puts("2次元配列とfor文の組み合わせ");
	for(int i=0; i<3; i++)
	{
		for(int j=0; j<3; j++)
		{
			num2[i][j] = i+j;
			printf("%d,",num2[i][j]);
		}
	}

	return 0;
}
```

出力結果：

```
1 次元配列と for 文の組み合わせ
0,1,2,
2 次元配列と for 文の組み合わせ
0,1,2,1,2,3,2,3,4,
```

2 次元配列と for 文の組み合わせを利用する時は、for 文の中にさらに for 文を入れる入れ子

構造にしてあげます。

課題：for 文を使って要素数 10 の 1 次元配列に 0～9 をそれぞれ代入した後、各配列に格納されている値を出力するプログラムを作成せよ。

制御式（while 文）

while 文も for 文と同様に基本的には繰り返しの文ですが、for との違いは、for 文は整数型変数を用いて繰り返し回数を設定するのに対し、while 文は丸括弧内の条件が満たされている間はずっとループを繰り返すことにあります。while 文の文法は以下の通りとなります。

```
while( 条件式 )
{
        条件式が真の間繰り返し処理する内容
}
```

while 文プログラム：

```
#include<stdio.h>
int main()
{
        int num = 0;
        int num2 = 20;
        while(num<6 && num2>15)
        {
                printf("%d,%d¥n",num,num2);
                num++;
                num2--;
        }
        return 0;
}
```

出力結果：

```
0,20
1,19
2,18
3,17
4,16
```

この例を見てみると、while の丸括弧 ( ) の中では、if 文の時に使用した条件が書かれています。今回の例だと、「変数 num が 6 より小さくて、かつ変数 num2 が 15 より大きい間は while 文の中を繰り返し実行しろ」ということを意味しています。この条件が満たされなくなったら while 文から抜け出すことができます。

制御式（switch 文）

分岐処理をする場合は if 文でできますが、分岐の数が多くなってくると if 文では煩雑になることがあります。そんなときは switch 文の出番です。switch 文の文法は以下の通りになります。

```
switch( 分岐の材料となる変数 )
{
case 分岐条件となる値 1:
        処理内容 1
        break;
case 分岐条件となる値 2:
        処理内容 2
        break;
default:
        どのケースにも当てはまらない場合行う処理
}
```

上の文法では case 分岐条件となる値 2 まで書きましたが、case はいくつでも増やすことができます。

switch 文のプログラム：

```c
#include<stdio.h>
int main()
{
	char chr;

	puts("aかbを入力してください。");
	scanf("%c",&chr);

	switch(chr)
	{
	case 'a':
		puts("aが入力されました。");
		break;
	case 'b':
		puts("bが入力されました。");
		break;
	default:
		puts("予想外の文字が入力されました。");
		break;
	}
	return 0;
}
```

出力結果：

```
a か b を入力してください。
b
b が入力されました。
```

switch 文の後の丸括弧 ( ) に記述されている変数にどんな値が格納されているかによって、その処理内容が変わってきます。例えば 'a' が入っていれば case 'a' 以降からの処理になり、 'b' が入っていれば case 'b' 以降からの処理になります。また、 default とは、その上で設定してある全てのケースに当てはまらない場合の処理内容を記述するためにあります。

大事なのは、すべての case の中の最後に break; をつけてあることです。 break; がないと、

その処理は case の垣根を越えて上から下に順番に処理を実行していってしまうので、意図的な場合を除き必ず付けるようにしましょう。

switch 文 break プログラム：

```
#include<stdio.h>
int main()
{
        int num=3;
        switch(num)
        {
        case 3:
                puts("3だよぅ");
        case 4:
                puts("4だよぅ");
        default:
                puts("defaultだよぅ");
        }
        return 0;
}
```

出力結果：

```
3 だよぅ
4 だよぅ
default だよぅ
```

break; をしっかり書いておかないと、case 4 以降もそのまま実行してしまうことになってしまいます。

課題：文字型変数 chr を作成し、'a', 'b', 'c', それ以外の文字のどれが格納されているかによって処理を分岐させるプログラムを作成せよ。

制御式（break,continue,return）

今までの制御式にもいくつか出てきましたが、break は「ループから脱出する」役割があり、

continue は「一回分だけループを飛ばす」役割があり、return は「関数から脱出する」役割があります。

break, continue, return プログラム：

```
#include<stdio.h>
void useReturn()
{
        int num = 0;
        while(1)
        {
                if(num==3) return;
                printf("%d¥n",num);
                num++;
        }
}
int main()
{
        int num = 0;
        puts("break 文利用の開始");
        while(1)
        {
                if(num==3)break;
                printf("%d¥n",num);
                num++;
        }
        puts("continue 文の開始");
        for(int i=0; i<3; i++)
        {
                if(i==1)continue;
                printf("%d¥n",i);
        }
        puts("return 文の開始");
        useReturn();
        return 0;
}
```

出力結果：

```
break 文利用の開始
0
1
2
continue 文の開始
0
2
return 文の開始
0
1
2
```

break によって while 文から脱出し、continue 文によって 2 回目のループのみスキップして再び for 文による繰り返しが発生し、return 文によって useReturn 関数から脱出しています。なお、今まで main 関数内の最後に書いてきた return 0 は戻り値を表すだけでなく、main 関数からの脱出（＝プログラムの終了）も意味していたのでした。

スコープ

変数には適用範囲があります。まずはプログラムを見てみましょう。

スコーププログラム：

```
#include<stdio.h>

void func()
{
        int num=5;
        printf("%d¥n",num);
}

int num = 10;

int main()
{
        puts("関数でのスコープ");
        int num = 3;
        func();
        printf("%d¥n",num);

        puts("制御式でのスコープ");
        for(int num=0; num<2; num++)
        {
                printf("%d¥n",num);
        }
        printf("%d¥n",num);

        return 0;
}
```

出力結果：

```
関数でのスコープ
5
3
制御式でのスコープ
0
1
3
```

num という名前の変数はあちこちで宣言、定義されていますが、それぞれ異なる値が代入されています。そして、main 関数内、func 関数内、制御式内でそれぞれ宣言した変数を、どこで出力するかによってその値が変わってきます。変数は、基本的に関数内や制御式内など、ブロック内で宣言された物が優先されます。このブロック内の変数をローカル変数と言います。それに対し、どのブロックの中でない所で宣言された変数をグローバル変数と言います。今回の場合だと、プログラムの最初の方で宣言、定義した int num=10 がグローバル変数で、その他の変数 num がそれぞれ main 関数、func 関数、制御式のローカル変数となります。基本的に、グローバル変数はどこからでも参照することができますが、ローカル変数はそれが属するスコープの中からしか参照できません。参照の際グローバル変数とローカル変数の名前がかぶる時はローカル変数が優先されます。

複数ファイルによるプログラム

プログラムが大きくなると一つの cpp ファイルだけでコードを書ききるのが困難になってきます。そんなときは異なる cpp ファイルにソースコードを書き、複数ファイルに分けてプログラムを組んでいく必要があります。
複数ファイルに分ける時は、関数の宣言を書いた「ヘッダファイル」、ヘッダファイルで宣言した関数の定義を書く「ソースファイル」の 2 つが必要になります。そして、「どこに書かれている関数の定義をしているのか」を表すために、ソースファイルの最初の方に

```
include "ヘッダファイル名"
```

を記述する必要があります。これをインクルードすると言います。
ここまで書いたら、あとは main 関数のあるファイルでそのヘッダファイルをインクルードしてあげれば自由にその関数を利用することができます。

複数ファイルによるプログラム：

test.h

```
//ヘッダファイル
void func();
```

test.cpp

```
//ソースファイル
#include"test.h"
#include<stdio.h>

void func()
{
	puts("test.cpp");
}
```

main.cpp

```
//インクルードするのはヘッダファイルだけでOK
#include"test.h"
#include<stdio.h>
int main()
{
	func();
	return 0;
}
```

出力結果：

```
test.cpp
```

func 関数のみ分離してプログラムの中で使用してみました。なお、自分で作ったヘッダフ

ファイルをインクルードするときは、ヘッダファイルの名前をダブルクォーテーション（" "）で囲います。C 言語に付属しているライブラリをインクルードするときは <> で囲います。

なお、関数だけでなく変数も外部に分けることができますが、変数の使い方は少し独特です。実体の宣言と定義はソースファイルのほうに書き、外部変数としての宣言をヘッダファイルに記述します。

外部変数プログラム：

test.h

```
extern int hoge;
```

test.cpp

```
#include"test.h"
int hoge=10;
```

main.cpp

```
#include"test.h"
#include<stdio.h>
int main()
{
        printf("%d¥n",hoge);
        return 0;
}
```

出力結果：

```
10
```

extern は、「外部のどこかにすでに宣言、定義されてる変数である」ということを意味します。実体は test.cpp にすでにあるので、test.h では extern を使います。

マクロ

文字列や値を保存する時は変数を使いますが、変数は途中でその値を変更できるのに対し、時には「変更することはない」と分かっている固定の値を用いることがあります。そんなときはマクロを使うことが多いです。マクロは

#define [マクロ名] [値]

と記述しておくと、そのマクロ名を変数のように使用することができます。変数のようにマクロの値を後で変更するということはできません。

マクロプログラム：

```
#include<stdio.h>
#define EXAMPLE 60
#define STRING "HelloWorld!"
int main()
{
        printf("%d¥n",EXAMPLE);
        puts(STRING);
        return 0;
}
```

出力結果：

```
60
HelloWorld!
```

マクロ名を書く時は、すべて大文字のアルファベットを使うことが多いです。

その他プリプロセッサ

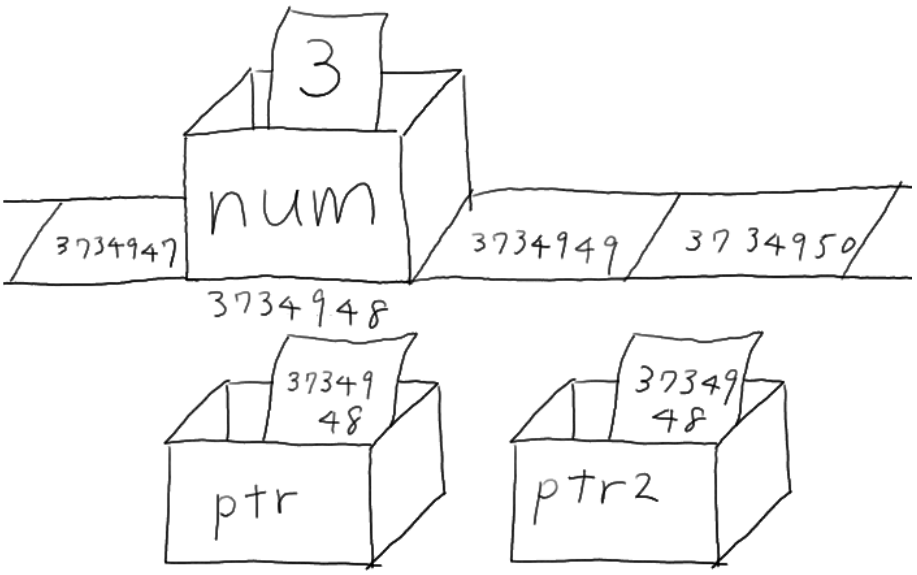
マクロはプリプロセッサと呼ばれる、前処理の一つです。コードを実行ファイルにするためにはコンパイルする必要がありますが、そのコンパイルをする前に行う処理を記述した物がプリプロセッサです。例えば、マクロもプリプロセッサの一つなのですが、マクロ名とデータを全て置き換えるという前処理をする働きがあります。
よく使うプリプロセッサは他にも

#pragma once

というものがあります。これは、重複インクルードを防ぐためのものです。複数ファイルに分けてプログラムを作成する時はヘッダファイルにこれを記述しておくと無難です。ただしこれは VisualC++依存のプリプロセッサなので、他のコンパイラでは使用できません。移植性がなくなるので、Windows でのみ使用するプログラムを書く時だけ使用しましょう。

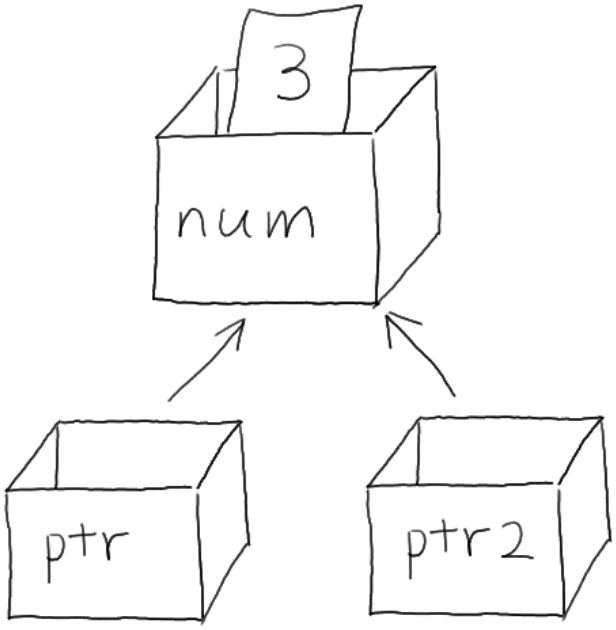
ポインタ

今まで C 言語で変数を扱う時はその変数に数値や文字列などのデータを代入していました。変数はデータを格納するただの箱でした。実はこの変数、C 言語では固有の住所のような番号が一つ一つ付いています。それがアドレスです。
変数が宣言されるとき、その変数に代入される値を保持しておくためのスペースがメモリ上に確保されます。ポインタとは、このメモリ上に確保された変数のアドレスを格納するものです。



↑実際にはポインタにはメモリ上のアドレスが格納されている

しかし実際にはメモリ上のアドレスそのものを意識させられることはあまりありません。
むしろ、ポインタは「ある変数を指している」といった感じに考える方が実用的です。



↑このように考える方が実用的

```
//ポインタの宣言
[指し示す対象の型]* [ポインタ名];

//変数のアドレスを代入
[ポインタ名] = &[変数];

//ポインタを代入
[ポインタ名] = [ポインタ];
```

ポインタプログラム：

```c
#include<stdio.h>
int main()
{
	int num = 3;
	int *ptr = &num;  //変数のアドレス代入
	int ptr2 = ptr;   //ポインタ代入

	printf("%d¥n",num);     //変数に代入されている値を表示
	printf("%d¥n",ptr);     //numのメモリアドレスを表示
	printf("%d¥n",*ptr);    //ポインタptrが指している先の変数に代入されている値を表示
	printf("%d¥n",*ptr2);       //ポインタptrが指している変数と同じ変数の値を表示
}
```

出力結果：

```
3
3734948
3
3
```

出力結果の 2 行目は、人によって異なると思います。
ポインタを宣言する時は、ポインタ名の前に * を付ける以外、変数の宣言と同じです。また、ポインタはメモリアドレスを入れるものなので、代入するのはデータではなくメモリアドレスです。出力結果の 2 行目を見てみると、今回のサンプルプログラムでは、変数 num は 3734948 というメモリ上の場所に確保されていたようです。
すでに宣言された変数名の前に & を付けることで、その変数のメモリ上の場所を表すことができます。今回の例だと、ポインタ ptr には変数 num のアドレスが格納されている、つまりポインタ ptr は変数 num を指し示していることになります。
指し示している先の変数に代入されている値を表現したい時は、すでに宣言、定義されたポインタ名の前に * をつけます。

ポインタによって変数を指し示すことができることで何が変わるのでしょうか。ポインタの利用例を見てみましょう。

ポインタ利用例：

```c
#include<stdio.h>
int main()
{
	int num = 3;
	int *ptr = &num;

	printf("%d¥n",num);
	printf("%d¥n",*ptr);
	*ptr = 5;
	printf("%d¥n",*ptr);
	printf("%d¥n",num);
}
```

出力結果：

```
3
3
5
5
```

ポインタ ptr の指し示している先の変数データを途中で 5 に変更しました。そうすると、変数 num のデータも変更されていることが分かります。このように、ポインタを通して指し示している変数のデータを変更することができます。

他にも、関数を用いてポインタを利用する方法もあります。

ポインタ利用例関数編：

```c
#include<stdio.h>

void func(int *ptr)
{
        *ptr = 5;
}

int main()
{
        int num = 3;
        printf("%d¥n",num);
        func(&num);
        printf("%d¥n",num);
        return 0;
}
```

出力結果：

```
3
5
```

関数の引数にポインタを入れることで、その指し示している先のデータを変更して返すことができます。つまり、本来はスコープの関係上関数の中では扱えない外部の変数を、ポインタを引数に入れることで関数の中で編集することができます。

課題：1. ポインタ ptr を作り、これを通して整数型変数 num の値を編集せよ。
2. 変数の値を 2 倍にして返す関数を作成せよ。ポインタを使用して直接変数の値を書き換えるようにせよ。

ファイルの利用

C 言語ではテキストファイルやバイナリファイルを扱うことができます。
ファイルを扱う場合は、ポインタを通じてまずはそのファイルへのストリームを作成する

必要があります。ファイルストリームを作成して、文字列をテキストファイルとして出力してみましょう。

ファイルストリームプログラム：

```
#include<stdio.h>
int main()
{
        FILE* file;
        file = fopen("Hello.txt","w");
        if(file == NULL)
        {
                puts("エラーが発生しました。");
                exit(EXIT_FAILURE);
        }
        fputs("HelloFileStream!¥n",file);
        fclose(file);
        return 0;
}
```

出力結果：

```

```

Hello.txt 出力結果：

```
HelloFileStream!
```

ファイルにつながる窓口は FILE* file です。このポインタを通じてテキストファイルにアクセスし、読み書きを行います。
fopen 関数はファイルストリームを作成するのに使用します。第 1 引数に対象のテキストファイル名を記述します。第 2 引数にオープンモードを記述します。オープンモードは以下のものがあります。

オープンモード	意味
r	テキストファイルを読み込み用に開く
w	テキストファイルを書き込み用に開く・作る
a	テキストファイルを追記用に開く・作る
rb	バイナリファイルを読み込み用に開く
wb	バイナリファイルを書き込み用に開く・作る
ab	バイナリファイルを追記用に開く・作る
r+	テキストファイルを読み書き両用に開く
w+	テキストファイルを読み書き両用に開く・作る
a+	テキストファイルを読み追記両用に開く・作る
rb+	バイナリファイルを読み書き両用に開く
wb+	バイナリファイルを読み書き両用に開く・作る
ab+	バイナリファイルを読み追記両用に開く・作る

「作る」と書いてあるものは、そのファイルが存在しない時自動で作成してくれます。
一度 fopen 関数でファイルストリームを開いたら、使い終わった時にそれを閉じる作業をする必要があります。それをしてくれるのが fclose 関数です。引数には、ファイルポインタを入れます。
今回の例では、ファイルストリームが開けなかった時に備えてエラー処理を記述しておきました。 if(file == NULL) と書くとエラー処理を行うことができます。
fputs 関数は、ファイルへの書き込みを行う関数です。第 1 引数に出力したい文字列を入れます。改行は手動で ¥n 打ち込まないとしてくれないので注意です。第 2 引数にはファイルポインタを入れます。

テキストファイルへの書き込み

読み書きを行う関数は様々あります。先ほどの例で使った fputs 関数以外にも、fputc 関数、fprintf 関数があります。

fputs 関数……文字列を書き出す関数。途中で ¥n を入れれば複数行出力することもできる。
fputc 関数……1 文字を書き出す関数。
fprintf 関数……printf 関数のような、書式付き出力関数。

書き込み関数の例を見てみましょう。

ファイル書き込みプログラム：

```
#include<stdio.h>
int main()
{
	FILE* file;
	file = fopen("Hello2.txt","w");
	fputs("Hello,fputs¥n",file);
	fputc('a',file);
	fputc('¥n',file);
	fprintf(file,"fprintf:%d, %s, %f¥n",32,"Hello",0.22);
	fclose(file);
	return 0;
}
```

出力結果：

```

```

Hello2.txt：

```
Hello,fputs
a
fprintf:32, Hello, 0.220000
```

fputs 関数は文字列をファイルに出力する関数です。第 1 引数に出力する文字列を入れます。第 2 引数にファイルポインタを入れます。

fputc 関数は、1 文字をファイルに出力する関数です。第 1 引数に出力する文字を入れます。第 2 引数にファイルポインタを入れます。

fprintf 関数は、書式付き出力関数です。第 1 引数にファイルポインタを入れること以外は printf 関数と使い方はほとんど同じです。第 2 引数に出力したい文字列とフォーマット指定子を入れます。第 3 引数以降は、第 2 引数に記述したフォーマット指定子に代入するデータを入れていきます。

テキストファイルの読み込み

続いてはファイルを読み込む関数の解説です。読み込み関数にも様々あります。

fgets 関数……1 行を読み込む関数。
fgetc 関数……1 文字を読み込む関数。
fscanf 関数……書式に従って読み込む関数。

早速例を見てみましょう。

Hello.txt：

```
abcde
fghij
```

Hello2.txt：

```
abcde
fghij
```

Hello3.txt：

```
abc 20 31.54
```

ファイル読み込みプログラム：

```c
#include<stdio.h>
int main()
{
	char str[80];
	char chr;
	int num;
	float num2;
	FILE* file;

	//fgets関数
	file = fopen("Hello.txt","r");
	fgets(str,sizeof(str),file);
	puts(str);
	fclose(file);

	//fgetc関数
	file = fopen("Hello2.txt","r");
	chr = fgetc(file);
	printf("%c¥n",chr);
	fclose(file);

	//fscanf関数
	file = fopen("Hello3.txt","r");
	if(file == NULL)
	{
		puts("error");
		exit(EXIT_FAILURE);
	}
	fscanf(file,"%s%d%f",str,&num,&num2);
	printf("%s %d %f",str,num,num2);
	fclose(file);

	return 0;
}
```

実行結果：

```
abcde


a
abc 20 31.54
```

fgets 関数は、ファイルポインタの指しているファイルから 1 行だけ読み込む関数です。第 1 引数には読み込んだ文字列を格納する配列へのアドレスを入れます。第 2 引数には第 1 引数に入れたポインタが指す配列の要素数を入れます。第 3 引数にはファイルポインタを入れます。ここで注意すべきなのは、Hello.txt の 1 行目の最後に改行コードがあり、さらにその後 puts 関数で出力することにより自動的に改行されるため、2 回改行することになるということです。実行結果の 2 行目を見てみると、puts 関数により自動的に付加された改行が表示されています。

fgetc 関数は、ファイルから 1 文字だけ読み込む関数です。引数にはファイルポインタのみ入れます。

fscanf 関数は、書式付き入力関数です。printf 関数のように、書式を指定してファイルを読み込むことができます。第 1 引数にはファイルポインタを入れます。%s は文字列、%d は整数、%f は小数を読み込むフォーマット指定子であり、printf 関数で使ったものと全く同じものがここでも使われます。フォーマット指定子は他にも「%c」「%u」などがあります。今回の例では、Hello3.txt の中の半角空白を含む文字列の内、「abc」を文字列として読み込み、「20」を整数として読み込み、「31.54」を小数として読み込み、それぞれを第 3 引数以降の str, num, num2 に格納しています。

ファイルポジションとシーク

ファイルを用いた関数である fgets 関数や fgetc 関数などを用いる時は、その処理の対象がファイルのどの位置からなのかということが問題になることがあります。この処理の対象となる位置をファイルポジションといいます。そして、目当ての位置から処理を開始するためにファイルポジションを移動させることをシークと言います。このファイルポジションとシークのプログラムを示します。

Hello.txt：

```
abcde
fghij
klmno
12345
```

ファイルポジション・シークプログラム：

```c
#include<stdio.h>
int main()
{
	char str[80];
	FILE* file;
	file = fopen("Hello.txt","r");
	fgets(str,sizeof(str),file);
	printf(str);
	fgets(str,sizeof(str),file);
	printf(str);
	//ファイルポジションをファイルの先頭に移動
	rewind(file);
	fgets(str,sizeof(str),file);
	printf(str);
	//ファイル先頭から8文字移動した位置にファイルポジションを移動
	fseek(file,8,SEEK_SET);
	fgets(str,sizeof(str),file);
	printf(str);
	fclose(file);
	return 0;
}
```

実行結果：

```
abcde
fghij
abcde
ghij
```

シークするのに使用する関数は rewind 関数と fseek 関数です。
rewind 関数はファイルポジションをファイルの先頭に移動させる関数です。引数にはファイルポインタを入れます。
fseek 関数は、ある場所から何文字分先（もしくは前）の箇所にファイルポジションを移動させるかを決める関数です。第 1 引数にはファイルポインタを入れます。第 2 引数の前に第 3 引数について先に説明すると、第 3 引数には、起点となる箇所を設定するマクロを入れます。第 3 引数には次のマクロのいずれかを入れることができます。

第 3 引数に入るマクロ	起点位置
SEEK_SET	ファイルの先頭
SEEK_CUR	現在のファイルポジション
SEEK_END	ファイルの最後

そして、第 2 引数は、第 3 引数に記述した起点位置から何文字分先（もしくは前）にファイルポジションを移動させるかを表す整数を入れます。上の例だと、ファイルの先頭から 8 文字分先にファイルポジションを移動させたため、その直後の fgets 関数で得られる文字列が「ghij」となっています。
このようにしてシークを駆使することで目当ての箇所のみテキストファイルを読み込んだりすることができます。

課題 : インターネットで Web サイトを見る時にダウンロードされる HTML ファイルはテキストファイルであり、拡張子を変えただけのものである。あらかじめ適当な Web サイトから入手した HTML ファイルをコピーするプログラムを作れ。

構造体

C 言語では、複数の変数をまとめて新しい変数のようなものを作り出すことができます。それが構造体です。例えば、シューティングゲームの場合自機の X 座標位置、Y 座標位置、あたり判定の広さなど、自機に関する情報をまとめて管理したりするのに便利です。
構造体は以下のようにして定義します。

```
struct [構造体名]
{
        //メンバ変数の記述
};
```

また、定義した構造体を使って変数を宣言するには以下のようにします。

```
[構造体名] [変数名];
```

また、構造体の定義と変数の宣言を一緒にして

```
struct [構造体名]
{
        //メンバ変数の記述
} [変数名];
```

とすることもできます。

構造体プログラム：

```
#include<stdio.h>
struct info
{
        int x;
        int y;
        int size;
};

int main()
{
        info data;
        data.x=10;
        data.y=20;
        data.size=30;

        printf("x=%d, y=%d, size=%d",data.x,data.y,data.size);
        return 0;
}
```

出力結果：

```
x=10, y=20, size=30
```

info 構造体を新たに作成し、info 型の変数 data を作成しました。構造体に含まれる変数（これをメンバ変数と呼ぶ）にアクセスするにはドット(.)を用いて、

[構造体の変数名].[メンバ変数名]

とします。
また、作成した構造体は変数と同じように扱うことができるので、関数の戻り値にすることもできます。

構造体と関数プログラム：

```
#include<stdio.h>
struct info
{
	int x;
	int y;
	int size;
};
info setInfo(int t_x, int t_y, int t_size)
{
	info hoge;
	hoge.x = t_x;
	hoge.y = t_y;
	hoge.size = t_size;
	return hoge;
}
int main()
{
	info data;
	data = setInfo(10,20,30);
	printf("x=%d, y=%d, size=%d",data.x,data.y,data.size);
	return 0;
}
```

出力結果：

```
x=10, y=20, size=30
```

列挙型

列挙型を覚えなくてもゲームは作れますが、この列挙型を学んでおくと非常に便利なので今回講座資料にとりあげました。列挙型は、マクロのような文字列を識別コードのように

用いることができるものであり、状態（オブジェクト指向ではステートといわれる）を識別して表すのに非常に便利です。実際には列挙型の実態は、単に整数をマクロのように置き換えただけなのですが、これによりコードの可読性が向上します。
新たな列挙型を定義するときは、

enum [列挙型名]{ 列挙定数 1, 列挙定数 2, … };

とします。列挙定数には、マクロのように大文字だけのアルファベットの単語を入れるのが慣例です。そして作った列挙型を宣言するときは

[列挙型名] [変数名]

とします。
宣言以降、その変数には定義の際に記述した列挙定数を代入することができます。
また、定義と宣言を一緒にして、

enum [列挙型名] {列挙定数 1, 列挙定数 2, …} [変数名];

とすることもできます。

列挙型プログラム：

```
#include<stdio.h>
enum Hoge{START, WAIT, STOP, FINISH};
int main(){
	int i=0;
	Hoge state = START;
	puts("初期状態はSTARTです");
	while(true){
		i++;
		if(i%5==0){
			switch(state){
			case START:
				state = WAIT;
				puts("状態がWAITになりました");
				break;
			case WAIT:
				state = STOP;
				puts("状態がSTOPになりました");
				break;
			case STOP:
				state = FINISH;
				puts("状態がFINISHになりました");
				break;
			case FINISH:
				puts("FINISHです");
				return 0;
			}
		}
		else{
			printf("%d¥n",i);
		}
	}
	return 0;
}
```

実行結果：

```
初期状態は START です
1
2
3
4
状態が WAIT になりました
6
7
8
9
状態が STOP になりました
11
12
13
14
状態が FINISH になりました
16
17
18
19
FINISH です
```

列挙型は制御文 switch との組み合わせが強力です。これにより、状態遷移をわかりやすく書くことができます。

C 言語ライブラリ

C 言語にはすでに備わっている便利なライブラリがあります。例えば、今まで使用してきた printf 関数や、puts 関数、fgets 関数なども C 言語に付属しているライブラリです。このほかにも、数値の絶対値を取得する関数、外部の exe ファイルを実行する関数、時間を取得する関数、乱数を発生させる関数など、便利な関数がたくさんあります。以下のリンクから、そのライブラリを一通り見て、使い方を自分で学んでみましょう。

C/C++リファレンス　　http://www.cppll.jp/cppreference/
C++リファレンス（日本語版）　　http://www.cppreference.com/wiki/jp/start

参考文献

この講座資料は、なるべく不必要なものは除いて、最短でプログラムを組めるだけの知識を得られるように書きました。そのため、一部説明を省いた部分、説明していない仕様もあります。完全版としての C 言語を学びたい方は、以下のリンク先より学ぶのが良いでしょう。

Programming Place Plus　　http://www.geocities.jp/ky_webid/