第4回 特別講義

本日のゴール

C言語基礎：関数を作成することができるようになる、変数のスコープが理解できる

関数とは

実はC言語は関数の塊

関数と言っても大きく２つの書き方がある。

1. 関数を作るための書き方
2. 関数を使うための書き方

関数を作る書き方

main関数も他の関数も基本の型は同じ

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

```
関数の型 関数名(引数)
{
    return 戻り値;
}
```

sumという名前の関数を作ってみた場合

- sum関数の仕様は以下とする
- ２つの整数値の引数をとり、その引数を足した合計値を戻す

上記の条件を満たす関数の作り方をSTEPに分けて解説します。

STEP1 まずは型をコピー

```
関数の型　関数名(引数)
{
    return 戻り値;
}
```

STEP2 sum関数なので名前を変えてみる

次のコードを編集して関数名をsumにしてください。

```
関数の型  関数名(引数)
{
    return 戻り値;
}
```

STEP2 sum関数なので名前を変えてみる

正解はこちら、sum関数なら関数名の部分をsumにもし、triangle関数を作るなら
関数名の部分をtriangleに変更します。関数名はその関数が行う処理をイメージできる名前にするとわかりやすくなるので、拘って名前をつけましょう。

```
関数の型 sum(引数)
{
    return 戻り値;
}
```

名前をつける際の常套句

変数名や関数名など名前をわかりやすくつける際に、単語同士を組み合わせる事があります。
例えば、String型をInt型に変更する関数を考えたとします。この時、名前の付け方として単語の組み合わせ方法に大きく２パターンあります。スネークケースとキャメルケースと呼ばれる方法です。

スネークケースとキャメルケース

```
string_to_integer(); //  単語同士を_（アンダーバー）で繋げる方法をスネークケースと呼ぶ
stringToInteger(); //単語同士の繋がりを大文字にして繋げる方法をキャメルケースと呼ぶ
```

これらは、名前づけの際に利用する常套句のようなものです。覚えておきましょう。

STEP3 ２つの整数を取るための変数を用意して引数とする

変数は整数なのでint型 a や b は変数名なので任意の文字で作れる（別にaやb以外でも良い。本当は意味がわかる名前をつけると良い）

```
関数の型 sum(int a, int b)
{
    return 戻り値;
}
```

double型の引数が１つの場合はどのように書けばよいか？

double型の引数aを１つ持つ sum関数を書いてコンパイルしてください。

```
double sum(/* ここに答えを書いてね */){
    return a + a;
}

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

STEP4 戻り値を準備する

戻り値は、その関数が実行した結果を返す時に必要なものです。ただし、関数によっては戻り値がなくても構わないものもあります。今回は戻り値がある場合について解説します。

```
関数の型 sum(int a, int b)
{
    int result;
    result = a + b;
    return result;
}
```

上記は引数のaと引数のbを結果を代入する為の変数 result に入れて結果を返してます。result は int型の変数でint型を戻り値として結果を返してます。

STEP5 戻り値の型が決まれば関数の型が決まる

resultはint型だったので、関数の型もintになります。

```
int sum(int a, int b)
{
    int result;
    result = a + b;
    return result;
}
```

char型の戻り値を持ち、 文字'a' を返す return_a関数を完成させなさい。

```c
#include <stdio.h>

//ここにreturn_a関数を作成してください。

int main(void) {

    printf("%c", return_a());
    return 0;
}
```

戻り値がない場合

それでは、次に戻り値が無い場合を考えてみましょう。
sum関数は引数aとbを受け取って結果を数値として返す関数でした。
それを結果を出力するだけの関数にしてみます。
どのように変更すれば良いかみてみましょう。

```
void sum(int a, int b)
{
    int result;
    result = a + b;
    printf("%d + %d = %d", a, b, result);
}
```

この関数は、return をしてないので戻り値がありません。その場合の関数の型はvoidを使います。

リファクタリング

実はこんな書き方もできるというので上記の説明で使ったプログラムの別の書き方をご紹介します。
以下のファイルは画面出力と同時に結果も返す型に変えてますが、結果を返す変数を省略して直接計算結果を戻してる例です。

```
int sum(int a, int b)
{
    printf("%d + %d = %d", a, b, a + b);
    return a + b;
}
```

関数の定義の方法がわかったので次は関数の使い方

先ほどのsum関数を使う時はどのように書けば良いでしょう？

呼び出し方法は簡単です。関数名を使って呼び出します。

```
関数名(引数);
```

先ほどのsum関数の場合の例、引数aに1を引数bに2を渡してる場合

```
sum(1, 2);
```

main関数からの呼び出し

続いて、main関数からsum関数を呼び出せるようにしましょう。

```c
#include <stdio.h>

int sum(int a, int b){
    return a+b;
}

int main(void) {
    sum(1, 2);
}
```

このように,`関数名(引数)`で呼び出しできます。

main関数とsum関数の定義場所の関係

今度は、sum関数の定義をmain関数の後に書いてみます。

```
#include <stdio.h>

int main(void) {
    sum(1, 2);
}

int sum(int a, int b){
    return a+b;
}
```

コンパイルエラーが出ましたよね？

これは、プログラムは上から順に実行される為、sum関数の定義の前に存在する「sum関数の呼び出し文」が存在しないと判断されてしまう為です。

プロトタイプ宣言

main関数より前に定義しないと自作関数が使えないとmain関数の前に必ず書かないといけなくなります。
関数がたくさんある場合、常に順序関係を気にかけて、
必要な関数を先頭に持ってくるのは面倒です。

どんな関数が作られているのか、前もって一覧にしておくようにすれば、
いちいち関数を前に書いたり後に書いたりする心配はなくなります。
関数の一覧を作るには、関数の形を、プログラムの初めの方に並べます。

プロトタイプ宣言の書き方

```c
#include <stdio.h>

/* プロトタイプ宣言 */
int sum(int,int);

int main(void) {
    printf("%d",sum(1, 2));
    return 0;
}

int sum(int a, int b){
    return a+b;
}
```

問い1

次の関数をプロトタイプ宣言を使わずに実行できるように修正してください。

```
#include <stdio.h>

int main(void) {
    printf("%d",multiply(2));
    return 0;
}

int multiply(int a){
    return a*a;
}
```

問い2

次の関数をプロトタイプ宣言を使って実行できるように修正してください。

```
#include <stdio.h>

int main(void) {
    printf("%d",multiply(2));
    return 0;
}

int multiply(int a){
    return a*a;
}
```

問い3

次の関数をプロトタイプ宣言を使って実行できるように修正し、
double型の実数を引数に取り、戻り値もdouble型で返すように修正してください。

例) `multiply(2.0)` を実行したら `4.000000` を返す

```
#include <stdio.h>

int main(void) {
    printf("%d",multiply(2));
    return 0;
}

int multiply(int a){
    return a*a;
}
```

問い４

コンソールから入力値を取得する scanf("%d %d", &input, &input2); 関数がある。使い方を以下に示すので

応用して、multiply(int a)のaの値をscanfで取得して実行するプログラムを書きなさい。

※ コンソールから入力値を取得することを今後は標準入力と表現します。

例)

```c
#include <stdio.h>

/* プロトタイプ宣言 */
int sum(int,int);

int main(void) {
    int input, input2;
    printf("こちらに1 2のようにスペースを入れて数値２つを入力してください->");
    scanf("%d %d", &input, &input2);
    printf("%d",sum(input, input2));
    return 0;
}

int sum(int a, int b){
    return a+b;
}
```

問い5

次の関数を作成してください。
関数名 repeat_calls で引数に整数値を入れるとその数値の回数分 "hello!"を改行付きでコンソール上に出力します。なお戻り値は不要です。
※ コンソール上への出力を標準出力といい、今後は標準出力と表現します。

問い6

次の関数を作成してください。
関数名 odd_number_judgment で奇数判定するプログラムを作成してください。
標準入力した数値が奇数であれば odd 偶数であれば even を標準出力する。
※HINT 奇数か偶数かは２で割り切れるかで判断できる 余りを求める演算子%を利用すると簡単にできる

変数の範囲

自作関数を作成する際に変数を定義する事は多いです。ではこの変数はどこまでの範囲有効なのでしょうか？
変数には有効な範囲に基づいてローカル変数、グローバル変数という概念があります。

実際に動きで確認してみましょう。

例）
カウントをする自作関数を作成して、関数を複数回呼び出します。また、今回は説明用にプレビュー関数を作りプレビュー関数の中でカウント関数を呼び出しています。実行して実行結果を確認してみましょう。

```c
#include <stdio.h>
void counter(void);
void preview(void);

int main(void) {
    counter();
    preview();
    return 0;
}

void counter(void){
    int count = 0;
    count++;
    printf("%d\n", count);
}

void preview(void) {
    counter();
}
```

ローカル変数

先に見た例では、変数の値は1のままでcount++;（インクリメント）を利用しても数値が 1 加算されずに毎回 1 が標準出力されていました。これは、変数の有効範囲（移行 スコープと呼ぶ）が関数の中のみだからです。その為、関数を呼び出す旅に初期化されます。仮に、初期化をしない場合には、値が不定格になり動作を保証しないため、どのような数値になるかは未知数になります。試しに、`int count = 0;`を`int count;`としてみましょう。（実開発の際には動作保証しないので初期化しないのは危ないのでやらないように）

```c
#include <stdio.h>
void counter(void);
void preview(void);

int main(void) {
    counter();
    preview();
    return 0;
}

void counter(void){
    int count;
    count++;
    printf("%d\n", count);
}

void preview(void) {
    counter();
}
```

グローバル変数

さて、先ほどの例はどうでしたか？環境や人によっても実行結果が変わってきます。試しにコンパイルをし直して再度実行してみると、先ほどとはまた違った値になる可能性も大いにあります。（たまたま同じになる事もあります）
それでは、次にグローバル変数の有効範囲を見ていきたいと思います。グローバル変数とは、関数の外で変数を用意して宣言した場合です。プログラム全体が終了するまで変数は生き残り続け、宣言されたソースファイル内のすべての関数から使用できるようになります。

例を見てみましょう。

```c
#include <stdio.h>
void counter(void);
void preview(void);

/* グローバル変数 */
int count;

int main(void) {

    printf("%d\n", count);
    counter();
    preview();
    return 0;
}

void counter(void){
    count++;
    printf("%d\n", count);
}

void preview(void) {
    counter();
}
```

グローバル変数の初期化

グローバル変数はローカル変数の場合と違って、明示的に初期化はしてませんでした。しかし、実行は1,2と表示され、
変数を0で初期化した振る舞いと同じ結果となってました。これは、グローバル変数は、プログラムの開始時に自動的に0に初期化される仕様となっています。ローカル変数との違いは、ローカル変数は関数の呼び出しのたびに作られるので、
そのたびに初期化しているとムダになるので自動的には初期化しないのですが、グローバル変数は最初に1回だけ初期化すれば良いためそのような仕様になっています。

しかし、グローバル変数は多用するとソースが読みづらくなります。どこで使われているか追うのが大変です。そこで、
グローバル変数は、プログラム全体で共有する特別なデータだけに使い、基本的にはローカル変数を使用することで、プログラムをわかりやすくできます。

ローカル変数のスコープ

グローバル変数、ローカル変数の大まかな違いを知った所で、ローカル変数について
さらに詳しく見ていきます。ローカル変数の有効範囲は実は`{}`のブロック内の中だけという仕様になっています。
`{}`はifや関数、for, whileなどで利用される事が多いですが、`{}`だけを利用する事も可能です。
有効範囲について実験してみましょう。

```c
#include <stdio.h>

int main(void)
{ //①の{}
    int value1 = 10;
    int value2 = 20;

    printf("1:value1 %d\n", value1);
    printf("1:value2 %d\n", value2);

    {//②の{}
        int value1;
        value1 = 30;
        value2 = 40;
        printf("2:value1 %d\n", value1);
        printf("2:value2 %d\n", value2);
    }

    printf("3:value1 %d\n", value1);
    printf("3:value2 %d\n", value2);

    return 0;
}
```

どうでしたでしょうか？ コメントで記載した`①の{}`で準備した変数は`②の{}`のかっこの中にも反映されている事が確認できました。また、`②の{}`の中で再度、value1という名前の変数を準備した場合には、`①の{}`で準備した変数とは別に`②の{}`の中だけに変数が効いてない事が確認できたと思います。

静的なローカル変数

さて、ここまでグローバル変数とローカル変数の特徴とスコープについて見てきました。
実は、この2つの中間的な存在である、変わった特徴を持つ変数が存在します。
関数内で変数を宣言する時に、型名の前にstatic(スタティック)とつけることで、
静的なローカル変数を宣言できます。

```c
#include <stdio.h>
void counter(void);
void preview(void);

int main(void) {
    counter();
    preview();
    return 0;
}

void counter(void){
    static int count; /* 静的なローカル変数 */
    count++;
    printf("%d\n", count);
}

void preview(void) {
    counter();
}
```

静的なローカル変数の特徴

実行するとグローバル変数と同じようにカウントされる事がわかります。
しかし、変数countは関数内で宣言されているため、本質的にはローカル変数です。
実際、main関数内で変数countを使用すると、エラーとなります。

これが、静的なローカル変数の特徴です。
関数内で宣言されているので、使用できるのは宣言された関数内のみですが、
その値はプログラムが終了するまで残るのです。
また、とくに初期化しなくても、自動的に0に初期化されます。

なお、初期化は初めに1回だけしか行われませんので、
たとえば、次のように初期化を行った場合もカウントできます。

```
static int count = 0; /* 静的なローカル変数 */
```

この変数は、関数が以前に呼び出された時の値を覚えておきたい場合に使用されます。
使い道は限られますが、関数の呼び出し回数を数える場合や、
検索を行う関数で、以前に見つかった文字位置を記憶する場合などが考えられます。

問い1

下記のコメントの部分をコメントにて指示されている変数名に置き換えてプログラムを完成させ、コンパイルして実行してください。

```c
#include <stdio.h>

int var;

int main(void)
{
    int count = 1;
    static int arc = 2;

    printf("グローバル変数の値%d", /* ここにグローバル変数を入れる */)
    printf("ローカル変数の値%d", /* ここにローカル変数を入れる */)
    printf("静的なローカル変数の値%d", /* ここに静的なローカル変数を入れる */)

    return 0;
}
```

総合課題：

下記を1～100までの値を標準出力するプログラムを書き、下記の条件に当てはまる時のみ以下に従うプログラムを書きなさい。

– 3で割り切れるときは「Fizz」を標準出力
– 5で割り切れるときは「Buzz」を標準出力
– 両方で割り切れるときは「FizzBuzz」を標準出力