目 次

第1章　第7回 3
- 1.1　第７回放送内容 3
- 1.2　ポインタ 3
- 1.3　ポインタ入門 3
- 1.4　値の代入 4
- 1.5　変数の中身 6
- 1.6　ポインタという名前 8
- 1.7　NULL ポインタ 9
- 1.8　ポインタの型 9
  - 1.8.1　変数の型の大きさ 9
  - 1.8.2　sizeof 10
  - 1.8.3　ポインタの型の大きさ 11
- 1.9　なぜポインタが必要なのか 14
- 1.10　演習問題 15
  - 1.10.1　演習１ 15
  - 1.10.2　演習２ 15
  - 1.10.3　演習３ 15
- 1.11　本日の講義のおさらい 16

第1章　第7回

1.1　第7回放送内容

1. ポインタ入門

1.2　ポインタ

ポインタはC言語では重要な項目であり、難しい項目の1つでもあります。C言語をはじめて勉強して途中で挫折した人の多くはポインタでつまずいたかと思います。実際にポインタを使いこなすことができるようになるには、相当時間がかかります。

1.3　ポインタ入門

ポインタとはなんでしょうか。よく、

> T*型の変数は、T型のオブジェクトのアドレスを保持できる [1, p.123]

なんていわれます。このように書くとまた挫折する人がいるかもしれません。今の段階では、コンピュータのメモリ上のアドレス（住所、または番地）を入れる変数[1] だと思ってもらえれば結構です。これでよく分からなくても大丈夫です。次の節で図を使って説明します。ここで、int型のポインタを宣言する場合は、

```
int* hoge;
```

または

```
int *hoge;
```

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、

[1]変数（変わりやすい数）とはなんでしょうか。たとえば int　hoge; と宣言すると、パソコンのメモリ上に、int (integer:整数）型の hoge という名前の箱ができると考えてもらえれば結構です。その箱の中には整数の値だけが代入（つまり箱の中に整数を入れる）できます。この箱の事を変数と呼びます。

```
int *hoge,*piyo
```

と宣言すると、hoge、piyo はどちらもポインタです。しかし、

```
int* hoge, piyo
```

とすると、hoge はポインタですが、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。さて、上の宣言では、メモリ上にint*型（整数のポインタ型）の箱を作ると考えてもらえればよいです。このことも、次の節で図を使って説明します。

1.4　値の代入

さて、`int hoge` というコードを書いたとします。これは、int 型の値を入れるため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に表してみます。

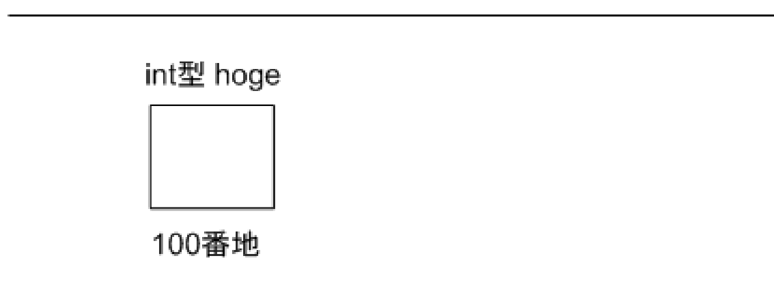

図 1.1: メモリ内部の様子

図 1.1 のようになります。これは、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。これは容易に理解できると思います。

では次に `int* p_piyo` という変数を作った様子を図 1.2 に表します。

さて、箱を作ったので、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。

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

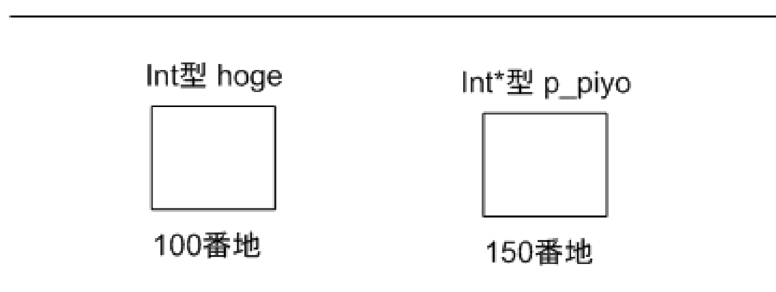

図 1.2: メモリ内部の様子

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;` のようにします。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。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。)

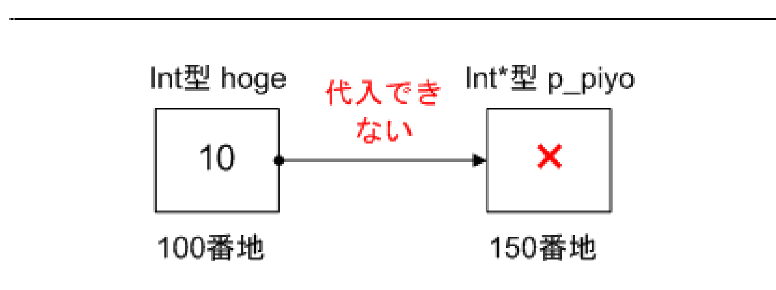

図 1.3: メモリ内部の様子

では、`p_piyo` に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 1.4 のようになります。

つまり、`&hoge` とは `hoge` の住所を取得せよ、ということです。

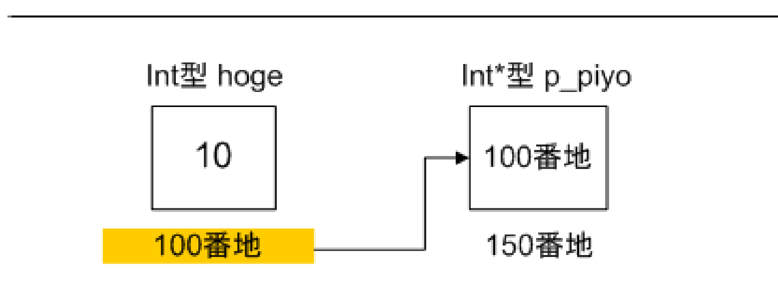

図 1.4: メモリ内部の様子

1.5　変数の中身

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。

```
int hoge;
int* p_piyo;

hoge = 10;
p_piyo = &hoge;

printf("hoge address :%p\n",&hoge);
printf("hoge :%d\n",hoge);
printf("p_piyo :%p\n",p_piyo);
printf("*p_piyo:%d\n",*p_piyo);
printf("p_piyo address :%p\n",&p_piyo);
```

ここで、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 1.5 のようになります。

`printf("hoge address :%p\n",&hoge);` のは前節で説明した&を使っています。よって、実行結果は `hoge address : 100 番地` となります。[2]

[2]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

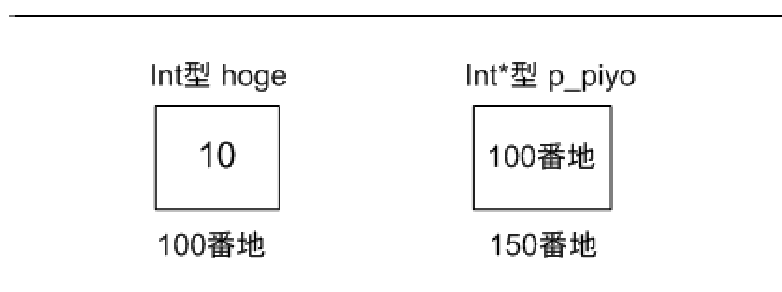

図 1.5: メモリ内部の様子

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。念のため実行結果を書きます。

```
hoge   : 10
p_piyo : 100 番地
```

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。*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。言い換えると、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。
最後の、 printf("p_piyo address :%p\n",&p_piyo); は、もう分かりますね。
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
```

1.6　ポインタという名前

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 §1.3 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 1.5 で示しました。ここで、図 1.5 は図 1.6 のように書き直すこともできます。

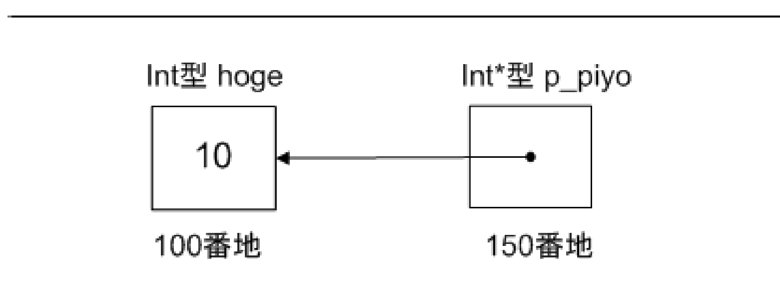

図 1.6: メモリ内部の様子

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。

1.7 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。そういった場合は、 p = NULL; とします。これは p はどこも指していないという事です。この NULL [3]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 )
```

や、

```
#define NULL -1
```

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。Java 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。これが某大型掲示板でよく使われている「ぬるぽ」です。

1.8 ポインタの型

1.8.1 変数の型の大きさ

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。以下の表を思い出してください。

short 型	-32768 ～ +32767
int 型	-2147483648 ～ +2147483647

[3]NULL はナル、またはヌルと読みます。本当はナルが正しいのですが、ヌルと呼ぶ人も多いです。

long型　-2147483648 ～　+2147483647
unsigned short型　0～+65535
unsigned int型 0 ～ +4294967295
unsigned long型 0 ～ +4294967295
signed char型　-128～+127
unsigned char型 0～+255
float型 3.4 * 10^-38 ～　3.4 * 10^+38
double型 1.7 * 10^-308 ～　1.7 * 10^+308

上の表をみてわかるように、int型は-2147483648 ～ +2147483647 の値を、char型は-128 ～ 127 [4]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。

さて、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ビットを二進法で表すと8桁の値を持つことになります。たとえば10進法で31の場合は、2進法で
00011111
となります。さて、8桁で表すことができる最大値は、
11111111
つまり10進法で255です。ここで
$127 - (-128) = 255$
つまり
$2^8 - 1$
です。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-128 ～ 127 であることがわかったと思います。unsigned char型の場合は　0 ～ 255 までの値を表すことができます。

同様にint型も4バイトなので、
$2^{8*4} - 1 = 2^{32} - 1$
となり、
-2147483648～　$+2147483647$ の範囲の値を表すことができます。

1.8.2　sizeof

さて、int型は4バイト、char型は1バイトという値は環境によって異なります。ここで、ある型が何バイトであるかを取得する方法があります。それがsizeofで

[4]char型の場合は必ずしもこの値とは限りません。0 ～ 255 の場合もあります。
[5]これらの値も環境によって異なる場合があります

す。sizeof は関数ではありません。int や return と同じ予約語です。では、次のようなコードを書いてみます。

```
#include <stdio.h>
int main(void)
{
        char hoge;
        int piyo;

        printf("char型のサイズは%uです", (unsigned int)sizeof(hoge));
        printf("int型のサイズは%uです", (unsigned int)sizeof(piyo));

        return 0;
}
```

%d は符号なし十進数を出力するのに使います。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。[6]

実行結果

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

1.8.3 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。

```
#include <stdio.h>
int main(void)
{
        char* hoge;
        int* piyo;
```

[6]環境によっては結果が違うかもしれません

```
        printf("char*型のサイズは%uです\n", (unsigned int)sizeof(hoge));
        printf("int*型のサイズは%uです\n", (unsigned int)sizeof(piyo));

        return 0;
}
```

さて、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。[7]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。これは次の章の配列とポインタというところで重要な考え方になってきますが、いまは簡単に説明しておきます。
例えば次のように配列を作ったとしましょう。

```
#include <stdio.h>
int main(void)
{
int array[5] = {5,10,15,20,25};
int* p;
p = &array[0];

        printf("p[0] = %d\n",*p);
p++;
printf("p[1] = %d\n",*p);
}
```

main関数のはじめの3行を図に表してみると図1.7のようになります。

[7]これも環境によって（以下略
[8]例えばint*やchar*

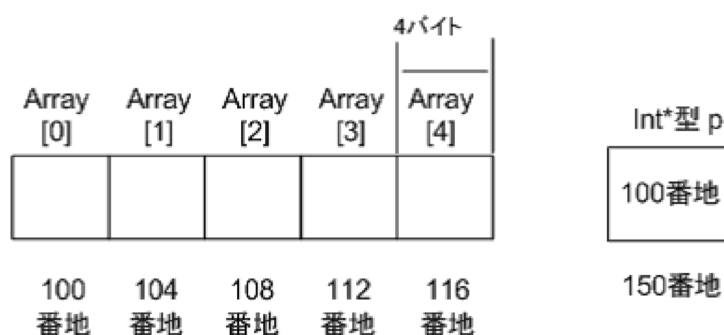

図 1.7: メモリ内部の様子

つまり、図 1.8 のようになります。

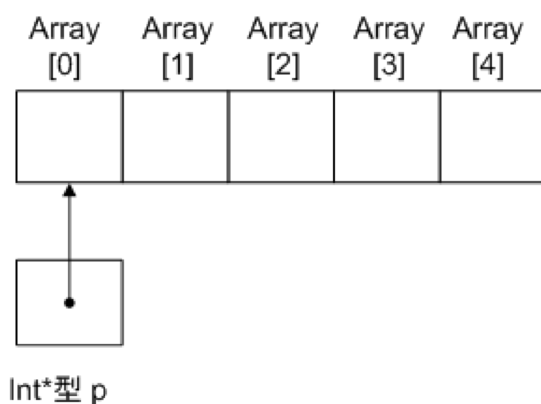

図 1.8: メモリ内部の様子

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 1.9 という感じになります。

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

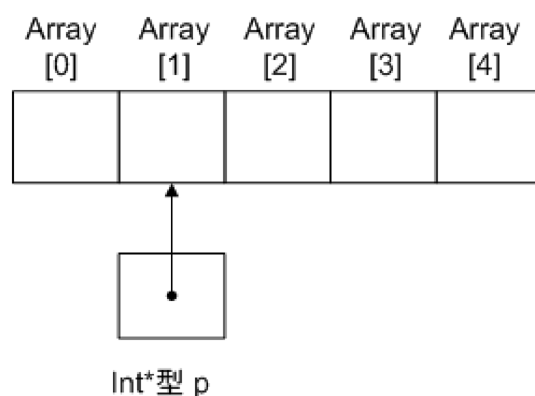

図 1.9: メモリ内部の様子

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

1.9 なぜポインタが必要なのか

今日の講義はこれでおしまいです。とりあえずポインタの基本的な使い方はわかってもらえたと思います。しかし、ここまで勉強してきて結局ポインタは何に使えるのかがまだわからないと思います。

ここで簡単にポインタの使用例を説明すると、ポインタは関数に引数として値を渡す時によく使います。関数に変数を渡すと、そのコピーが作られると説明しました。しかし、コピーの中身をいくら書き換えても、元の変数には何の影響も及ぼしません。それでは困ったことになってしまうので、そういった場合に使うのがポインタです。ポインタを使って値を書き換えたい変数のアドレスを引数として渡し、そのアドレスを通じて変数を書き換えることができます。これらの説明も次の章で説明します。

1.10　演習問題

1.10.1　演習１

次のコードの実行結果を答えなさい。

```
int hoge = 30;
int* piyo = &hoge;

printf("out1 :%d\n",hoge);
printf("out2 :%d\n",*piyo);
```

1.10.2　演習２

次のコードの実行結果を答えなさい。

```
int x = 10;
int y = 20;
int* hoge = &x;
int* piyo = &y;
int* tmp;

tmp = hoge;
hoge = piyo;
piyo = tmp;

printf("out1 :%d\n",x);
printf("out2 :%d\n",*hoge);
```

1.10.3　演習３

次のコードの実行結果を答えなさい。

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

p += 2;
printf("out1 :%d\n",*p);
```

これらのコードは実際に作って実行結果を確かめてみてください。

1.11　本日の講義のおさらい

ポインタは住所（アドレス）を入れる箱である

参考文献

[1] Bjarne Stroustrup 著 長尾高弘 訳: プログラミング言語 C++第三版.

[2] Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language Second Edition.

[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] ハーバートシルト著 トップスタジオ訳: 独習 C++ 改訂版.

[5] 前橋 和弥: ポインタ完全制覇.

[6] 浅井 淳: ポインタが理解できない理由.