C++入門

佐藤謙成

April 22, 2025

KUT-PG

目次

目次 (続き)

初めに

この C++入門では，プログラムの基本的な文法を学んでもらうことが目標である．既に習得済みの分野については飛ばしても問題はない．C++を完全に理解するためには参考書や youtube 等の活用をおすすめする．また，atcoder 等のプログラム課題については積極的に取り組んでもらいたい．

注意

このスライドは tex の beamer というドキュメントクラスを用いています.

Hello World の出力

Hello world

ここでは，プログラミング言語の最初の慣習として Hello world の出力を行なっていく．

Listing 1: Hello.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Hello␣World!" << endl;
6     return 0;
7 }
```

次のスライドからはこのソースコードの解説を行なっていく，

ソースコード 1 の解説 i

ヘッダーファイル

```
1 #include <iostream>
```

このコードは, ヘッダーファイルと呼ばれる物である．このファイルを読み込むことにより，cout や cin を用いることができる．

ソースコード 1 の解説 ii

main 関数

```
1 int main() {
2     ソースコード
3 }
```

各コードに必ず存在する特別な関数. プログラムは main 関数の最初から順に実行されていく. 基本的には,main 関数内に記述すれば良い. 詳しくは 55 ページで解説している.

cout « ”Hello World”

Hello World という文字列を出力している.
詳しくは 3 ページで解説している.

プログラムの終了

```
1 return 0;
```

このコードを記述することでプログラムを終了することができる. (必要なくても実行が最後まで到達すればプログラムは終了する.)

変数

変数とは

変数

記憶域 (メモリ)[a]に名前をつけてさまざまな値を格納できるようにしたもの．変数は値を格納する箱という認識で大丈夫である．

[a]そういうものがある程度の認識で大丈夫．

型	キーワード	サイズ
文字データ	char	1 バイト
符号付き整数	int	4 バイト
浮動小数点	float	4 バイト
倍精度浮動小数点	double	8 バイト
値なし	void	

表 1: 基本のデータ型

変数の宣言

変数をプログラム内で使用する際には変数の宣言というものが必要となる.

宣言方法

変数宣言の一般的な形式は次の通りである：**型　変数名;**[a]
実際にコード内で書く例は次の通りである：**int counter**[b]**;**

[a] int x, y, z; とまとめて宣言することもできる.
[b] 変数名は自身で設定することができるため, 格納される値に沿った名前にすること.

変数宣言は 1 つの文であるため, 一番最後にセミコロン (;) が必要である.
よく忘れるため注意すること.

変数の種類

グローバル変数

通常の関数や main 関数の外で宣言された変数. そのプログラム内のどの関数からも使うことができる. ただし, どこでどの手順で変数内の値が変更されたか不透明になりやすいため, 基本的には使用を避けたい.

ローカル変数

通常の関数や main 関数内で宣言された変数. 宣言された関数内でのみ使用可能. 変数内の値がどこでどの手順で変更されたか分かりやすい.

変数の代入

代入方法

代入の一般的な形式は次のとおりである：変数名 = 値;
実際の例は次の通りである：count = 5;[a]

[a] int count = 5; のように変数の宣言と同時にすることができる.

入出力

入出力

入力

```
cin >> (入力した値を入れる変数);
```

出力

```
cout << (出力したい変数, 文字列) << endl;
```

- endl で改行する.[a]
- ""(ダブルクォーテーション) で囲んだ文字列はそのまま文字列のまま出力される.

[a] "\n" でも可能.

プログラムの演算

算術式の計算

演算子	意味	優先順位
*	乗算	4
/	除算（余りは切り捨て)	4
%	剰余	4
＋	加算	5
-	減算	5

表 2: 算術演算子

もし, 加算・減算を先に求めたい場合は () を用いることで優先順位を上げることができる.

複合代入演算子

計算式の省略形. 競プロではよく使用されるので覚えておきたい.

複合代入演算子	代入演算子
a += 1	a = a + 1
a -= 3	a = a - 3
a *= 5	a = a * 5
a /= 2	a = a / 2
a %= 9	a = a % 9

表 3: 複合代入演算子

インクリメント, デクリメント

インクリメント

a++; または ++a; である.
この二つは計算の優先順位が存在するため, 式の途中で登場する場合は異なる結果となる.

デクリメント

a–; または –a; である.

次のスライドで実際にどのように違うのか検証していきたい.

インクリメント, デクリメント i

まずは, 前置インクリメント[1]のソースコードである.

Listing 2: Pre_increment.cpp

```
#include <iostream>
using namespace std;

int main(){
int x = 2;
int y;
y = ++x;
cout << "x␣=␣" << x << "␣y␣=␣" << y << endl;
return 0;
}
```

次は, 後置インクリメント[2]である.

Listing 3: Pos_increment.cpp

```
#include <iostream>
using namespace std;

int main() {
   int x = 2;
   int y;
   y = x++;
   cout << "x␣=␣" << x << "y␣=␣" << y << endl;
   return 0;
}
```

[1] ++a のこと.

[2] a++のこと.

前置インクリメント, 後置インクリメント

演算子	説明
前置インクリメント	先に値をインクリメント (+1) した後に代入
後置インクリメント	先に代入した後に値をインクリメント (+1)

表 4: 前置, 後置の違い

関係演算子

この演算子は, 数学等で頻繁に出るため, 簡単に押さえておけばよい.

演算子	関係
>	より大きい
>=	以上
<	より小さい
<=	以下
==	等しい
!=	等しくない

表 5: 関係演算子

論理演算子

1 年生の 2Q(情報代数), 3Q(離散数学) で出てくるため押さえておきたい.

演算子	操作
&&	論理積 (かつ)
||	論理和 (または)
!	否定

表 6: 論理演算子

p	q	p && q	p || q	!p
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

表 7: 真理値表

条件分岐

条件分岐を用いたソースコード

ここに載せているソースコードは条件分岐を用いた実際のコードである. 詳しい解説は次からのスライドで行う.

Listing 4: if.cpp

```
#include <iostream>
using namespace std;

int main() {
    int weather;
    cout << "今日の天気を数字で教えてください␣:␣"
    cin >> weather;

    if(weather == 1) {
        cout << "今日は晴れですね\n";
    }
    else if(weather == 2) {
        cout << "今日は曇りですね\n";
    } else {
        cout << "今日は雨ですね\n";
    }
    return 0;
}
```

if 文

if 文とは

条件式の条件が成立した場合, 特定の処理を行う文. if 文の条件式では, 関係演算子を用いて値を比較する.

Listing 5: if 文の一般形

```
1       if（条件式）{
2       条件が成立した場合に行いたい処理;
3       }
```

再掲 (関係演算子)

演算子	関係
>	より大きい
>=	以上
<	より小さい
<=	以下
==	等しい
!=	等しくない

表 8: 関係演算子

else if 文

else if 文とは

if 文の条件が成立せず, else if 文の条件が成立した場合に行いたい処理;

Listing 6: else if 文の一般形

```
1       if(条件式) {
2       条件が成立した場合に行いたい処理;
3       }
4       else if (条件式) {
5       if文の条件が成立しないかつ, else ifの条件が成立した場合に行いたい処理;
6       }
```

else 文

else 文とは

if 文 (else if 文) の条件が成立しない場合に行われる処理のこと.

Listing 7: else 文の一般形

```
1    if(条件式) {
2    条件が成立した場合に行いたい処理;
3    }
4    else {
5    if文の処理が成立しない場合に行われる処理;
6    }
```

switch 文による多分岐選択

switch 文とは

二者択一ではなく複数の選択肢がある場合に用いる文.

Listing 8: switch 文の一般形

```
switch(変数) {
   case 定数1 :
      実行内容;
      break;
   case 定数2 :
      実行内容;
      break;
   .
   .
   .
   default :
      実行内容;
      break;
}
```

一致する定数が見つからない場合は,default が実行される.

繰り返し処理

繰り返し処理を用いたソースコード

Listing 9: while.cpp

```
#include <iostream>
using namespace std;

int main() {
   for(int i = 0; i < 10; i++) {
   cout << i << "回目の操作です\n";
   }

   while(i = 10) {
   cout << i << "回目の操作です\n";
   i++;
   }
   return 0;
}
```

for 文

for 文とは

同じ処理を何度も行う場合に用いられる.

```
1       for(初期値; 条件判定; インクリメント;) {
2       繰り返して行う処理;
3       }
```

初期値

ループを制御する変数に初期値を設定する文.

条件判定

初期値の変数と目標の変数を突き合わせ, 真の場合ループを実行する. 必ず,for 文の最初に実行される.

インクリメント

初期値で設定した変数をインクリメント (デクリメント) を行う. 必ず,for 文の末尾で実行される.

while 文

while 文とは

条件文が真の間だけ繰り返される文.

```
1    while(条件式) {
2    繰り返す内容;
3    }
```

注意

while 文は繰り返す内容の中に条件が真になるか偽になるような処理を行わないと無限ループに陥ってしまう.[a]

[a] その場合は,Ctrl + c を押すことによって中断することができる.

do-while 文

do-while 文とは

条件分が真の間だけ繰り返される文.

```
1       do {
2       繰り返す内容;
3       } while(条件式);
```

while 文との違い

条件式が末尾で実行されるため, ループ内のコードは必ず 1 回は実行される.

ループのネスト

for 文等のループの内側の中にまたループが存在する時のことを言う．一般的には, 二重 for 文などと呼ばれる．

Listing 10: 二重 for 文

```
1       for(初期値; 条件判定; インクリメント) {
2          for(初期値; 条件判定; インクリメント) {
3             繰り返し実行する内容;
4          }
5       }
```

ループのネストの処理

1. 外側の for 文を一つ回す.
2. 内側の for 文を全て回す.
3. 外側の for 文を一つ回す.

⋮

break 文

break 文とは

break 文を用いることで, 通常の条件判定を待たずにループから脱出することができる. ループの内側で break 文に会うと, ループはすぐに終了する.

Listing 11: break.cpp

```
#include <iostream>
using namespace std;

int main() {
   for(int i = 0; i < 100; i++) {
      if(i == 10) {
      break;
      }
   }
   return 0;
}
```

continue 文

continue 文とは

continue 文に出会うと, そこからループ内のコードを全て無視し, ループの次の繰り返しに進む.

Listing 12: continue.cpp

```
#include <iostream>
using namespace std;

int main() {
   for(int i = 0; i < 10; i++) {
      if(i == 5) {
      continue;
      }
   }
   return 0;
}
```

配列と文字列

1 次元配列

配列とは

全てが同じ型をもち, 共通の名前によってアクセスされる変数のリストのこと. つまり, 複数の変数をまとめたもの.

配列の宣言

型　変数名 [サイズ[a]];
例として, int a[10];

[a]配列の数.

配列は 0 番目から始まるので、注意すること. サイズを 10 とした場合,0–9 までの計 10 個が配列の添え字[a]となる.

[a]添え字とは, 各要素が何番目かを表すもの.0 番目の要素の添え字は 0.

1 次元配列の初期化

1 次元配列の初期化は以下のようにすることができる. ただし, 競プロでは入力例から値を受け取るため使用することはほとんどない.

1 次元配列の初期化

型　配列名 [サイズ] = {値のリスト};
例として,　int　a[3]　=　{1, 2, 3};

1 次元配列と変数の違い i

配列と変数の違いについて視覚的に分かりやすい図を用いて考えていく．変数は一つ宣言すると一つの値しか代入することができない．しかし、配列は一つ宣言しても自分が欲しい数の値だけ挿入することができる.(ただし, 最初に宣言した個数しか代入することができない.)

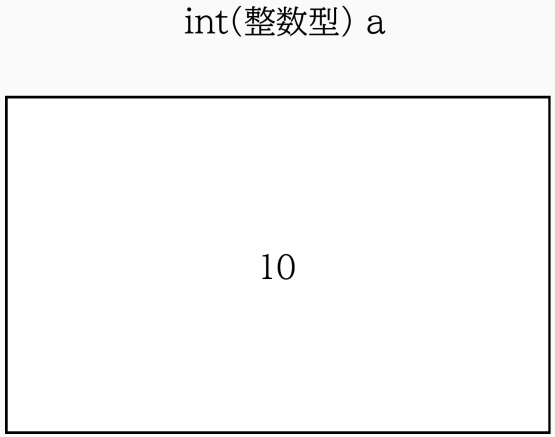

図 1: 変数

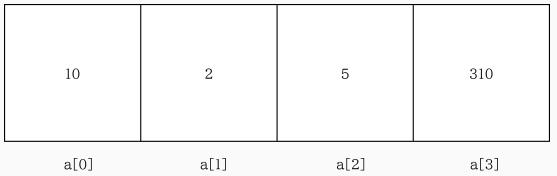

図 2: 配列

1 次元配列と変数の違い ii

変数と配列の違いは「データがメモリ上に一列に並べられているかどうかである.」[3]変数の場合は, メモリ上にバラバラに存在しているが配列は一列に並んでいる. そのため, 配列を用いると各データへ効率よくアクセスすることができるというメリットがある.

[3] イメージとしては 51 ページのような感じ

文字列

文字列型「String」は変数ではあるが,char[4]を 1 次元配列化したものとして考えることができる.
string 型の変数 s に"ABCD" を代入したとき, 以下のように考えることができる.[5]

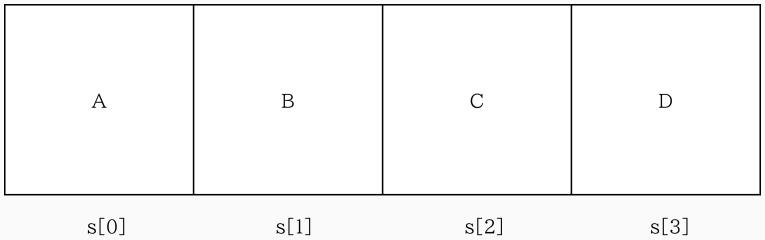

図 3: char 型の 1 次元配列

[4]文字型
[5]実際に,s[0] とプログラム上で書くと'A' のみが返ってくる.

文字列の使い方

文字列の使い方は C++のリファレンスを参照していただきたい.
`https://cpprefjp.github.io/reference/string/basic_string.html`

以下は, 文字列を扱う際によく用いるものの代表例を取り上げている.

名前	説明
begin	配列の要素の添え字を取得
end	最後の要素の添え字を取得

表 9: イテレータ[6]

名前	説明
size	文字列の長さを取得

表 10: 領域

[6] ポインタと同様に扱うことができるもの.

多次元配列

1 次元の配列だけでなく,2 次元以上の配列を作成することができる.(2 次元配列以上は使わなくてもプログラムを作成することは可能であるため, あまり使用しない.)

2 次元配列の宣言

型　変数名 [サイズ 1][サイズ 2];
例として,　int　a[10][12];

以下は,2 次元配列を視覚的に分かりやすくしたものである.

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]

図 4: 2 次元配列の図

2 次元配列の初期化

2 次元配列の初期化も 1 次元配列と同様に初期化することができる.

2 次元配列の初期化

型　配列名 [サイズ]　＝　{{値のリスト 1}, {値のリスト 2}　· · · };
例として,　int　a[2][2]　＝　{{1, 2}, {1, 3}}

配列と for 文との組み合わせ i

配列が一番有効活用できるときは,for 文等の繰り返し処理を行う文法と組み合わせて使うときである. 以下に, その例を挙げる.

Listing 13: arr_for.cpp

```
#include <iostream>
using namespace std;

int main() {
    int n;
    cin >> n;
    int a[n];
    for(int i = 0; i < n; i++) {
        cin >> a[i];
    }
    for(int i = 0; i < n; i++) {
        cout << a[i] << endl;
    }
    return 0;
}
```

配列と for 文との組み合わせ ii

ソースコード 13 はサイズが n の配列に n 回,for 文を回すことにより, 配列 a の全要素に値を入れることができる. これは

1. n 個の変数を一度に宣言し,
2. n 個の変数に for 文を回すだけで値を代入することができる

ということができ, 大量のデータを扱うことができるようになるのでぜひ使えるようにしていきたい.

可変長配列”vector”

今までの配列は最初にサイズを指定した後にサイズを変更することができなかった固定長配列というものである. しかし, プログラムの実行途中に配列の要素の増減を行いたい場合も出てくる. そのような場合に, この可変長配列「vector」を用いる.

可変長配列とは

プログラミングで用いられる配列変数の一種であり, 配列のサイズが固定されておらず, 実行途中の必要に応じて要素を追加, 削除することができる配列である.

可変長配列を使用する際の注意点

可変長配列「vector」を用いる場合には, ヘッダーファイルに vector を追加しなければならない.[a]

[a] ヘッダーファイルが bits/stdc++.h なら不要.

vector の宣言

vector の宣言は通常の配列とは異なるので注意する必要がある.

vector の宣言

vector<型>　変数名（最初の要素数）;
例として,　vector<int>　a(10);

vector の初期化

vector の初期化は固定長配列と同様に行ってもよいが, すべての要素に同じ値を代入したい場合は本項のやり方がおすすめである.

vector の初期化

vector<型>　配列名（要素数, 初期値);
例として,vector<int>　a(10, 1[a]);

[a] a[0] ～ a[9] までに 1 が代入される.

この方法を用いることで for 文を回すことなく配列を初期化することができる.

vector の使い方

vector の使い方については C++のリファレンスを参照していただきたい.
`https://cpprefjp.github.io/reference/vector/vector.html`

以下は,vector を扱う際によく用いるものの代表例を挙げている.

名前	説明
push_back	末尾へ要素を追加
pop_back	末尾から要素を削除
insert	要素の挿入

表 11: コンテナの変更

ポインタ

ポインタの使用例

ここでは, ポインタの簡単な使い方について以下のソースコードを記述する.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a =5;
6      int *p;
7      p = &a;
8      cout << p << endl;
9      cout << *p << endl;
10     return 0;
11 }
```

ポインタとは i

ポインタは難しい分野のため, 徐々に理解していけばそれで充分である.

ポインタとは

ポインタとは, 変数等がメモリのどの場所に保存されているかを指し示す仕組みである. つまり, データ本体ではなく, それがある場所のことである.

ポインタの宣言

型　*変数名;
例として,　　int *p;

ポインタとは ii

ポインタ演算子

ポインタを用いる際にはポインタ演算子というものを用いる．ここでは, p をポインタ変数として扱う．

演算子（変数そのもの)	説明
p	アドレス自体を意味
*p	アドレスが指し示す値
&q　(q はただの変数)	q が保存されているアドレスを指し示す

表 12: ポインタ演算子

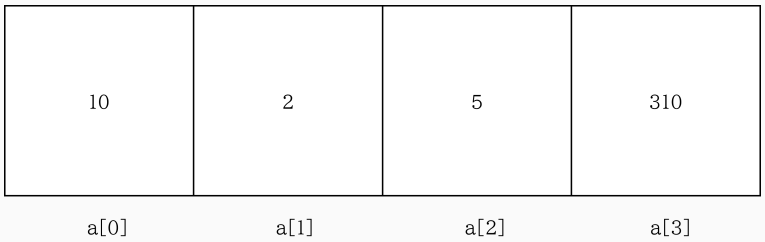

図 5: 配列とポインタの関係性

ポインタの演算制限について

ポインタの演算制限

ポインタでは整数の加算と減算以外は実行することができない. 即ち, ポインタ変数に適用できる演算は以下の 6 つのみである.

*, &, +, ++, -, −

ポインタ変数でのインクリメント, デクリメント

ポインタ変数で*p++としたとき, ポインタが指している値ではなく, ポインタ自体をインクリメントしている. ポインタが指している値をインクリメントしたい場合は,(*p)++とするのが正解である.

配列とポインタの関係性 i

ポインタと配列はとても密接に関わりあっている.

配列とポインタの関係

添え字をつけずに配列名の場合は, 配列の先頭を指すポインタとなってしまう. そのため, 変数として配列を扱いたい場合は添え字を忘れてはいけない.

Listing 14: p.cpp

```
#include <iostream>
using namespace std;

int main() {
    int a[3] = {1, 1, 1};
    int *p;
    p = a;
    cout << *p << *(p + 1) << *(p + 2) << endl;
    cout << a[0] << a[1] << a[2] << endl;
    return 0;
}
```

配列とポインタの関係性 ii

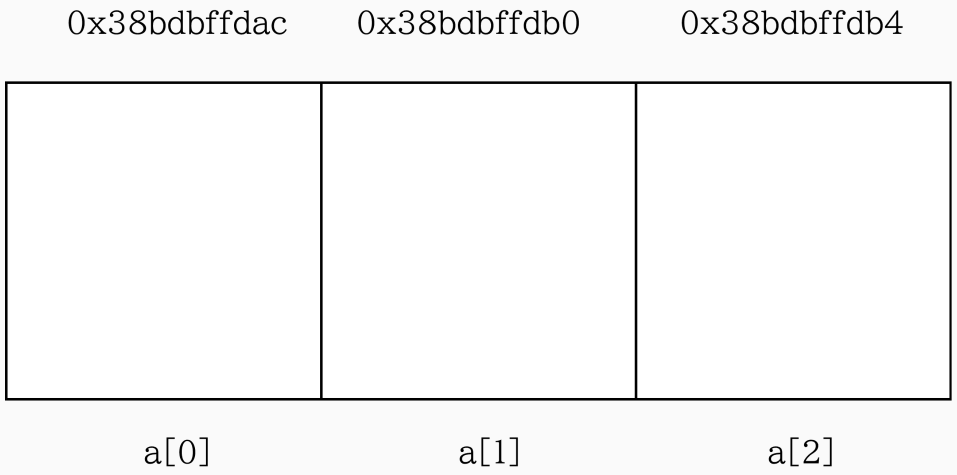

図 6: 配列とポインタの関係性

多重間接参照

多重間接参照とは

ポインタを使い別のポインタを指すことができる. その場合, 最初のポインタは 2 番目のポインタのアドレスを持ち,2 番目のポインタは変数を指す. しかし, 多重間接参照は連鎖をたどるのが面倒なため極力使用は避けたい.

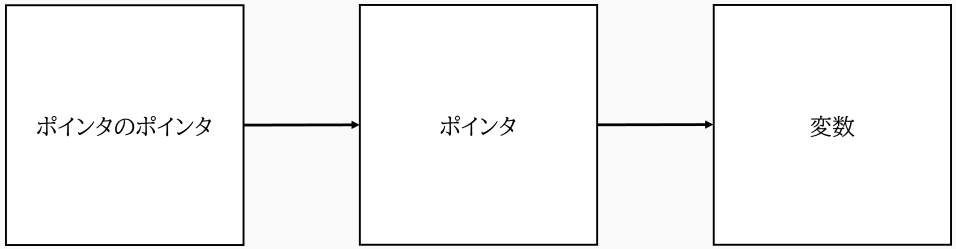

図 7: 多重間接参照の図

関数

関数とは

関数とは

ある処理をまとめて名前を付けたものを関数と呼ぶ.

関数を使うメリット

関数を使うことによってコードの重複を減らせることができることが理由である.

main 関数

Listing 15: main 関数

```
1        int main() {
2        実際に処理する内容;
3        return 0;
4        }
```

main 関数とは各ソースコードに存在する特別な関数である．この関数はプログラム実行時にシステムから呼び出される．また,return 0; に到達するとその時点でプログラムを終了する．

関数について

関数は以下の二つに分けることができる.

1. ユーザー定義型関数
2. 標準関数

ユーザー定義型関数

この関数は, プログラマーがそのコードの中で作成し, 実行する関数である. 必要になったタイミングで関数を作って使うことができる.

標準関数

この関数は, ヘッダーファイル等に最初から入っている関数である. そのため, プログラマーが実際に作成することなく使えることができる. ただし, ヘッダーファイルを読み込んでいない場合エラーが発生するので注意すること.

ユーザー定義型関数を作成する際には基本的には以下の二つが必要となる.

1. 関数プロトタイプ[7]
2. 関数定義

[7] 関数定義が main 関数前であれば必要はない

関数の使用例

ここでは, 関数を用いた際の全体の流れとして以下のコードを記述する.

Listing 16: func.cpp

```
#include <iostream>
using namespace std;

int Func_Plus(int n, int m);

int main() {
    int a, b, c;
    cin >> a >> b;
    c = Func_Plus(a, b);
    cout << c << endl;
    }

int Func_Plus(int n, int m) {
    int ans;
    ans = n + m;
    return ans;
}
```

次項から, 詳しい解説を行っていく.

関数プロトタイプ宣言について

関数プロトタイプ宣言とは

関数定義が main 関数の後の場合, ファイルをコンパイルする際にコンパイラに対して関数の情報を渡さないまま関数を呼び出してしまう. そのため関数を呼び出した際にその呼び出し方に誤りがないか確認するための宣言である.
(関数定義が main 関数前に行っておけば, プロトタイプ宣言は必要ない.)

プロトタイプ宣言には以下の 3 つの役割を果たしている.

1. 関数の戻り値の型をコンパイルに伝えること
2. 関数を呼び出すために使用した引数の型と仮引数の宣言時の型との間で不正確な型変換が行われたとき, コンパイラが検出して報告できるようにしている.
3. 関数に渡した引数の数が関数の仮引数の数と一致しない場合にも報告できるようにしている.

関数プロトタイプの宣言

```
返り値の型　関数名　（仮引数リスト）;
```

関数のプロトタイプ宣言の一般形は上に書いてある書き方である.

また, 引数は可変個にすることができるが使用することはない為, 宣言の方法だけ触れておく.

```
int     func     (int i, ...);
```

関数定義

Listing 17: 関数定義

```
1    戻り値の型　関数名　（引数リスト）{
2        処理内容;
3        return 戻り値;
4        }
```

関数定義には以下の要素が必要である.

1. 戻り値の型
2. 関数名
3. 引数リスト
4. 実際に行う処理
5. 戻り値

次項から各項目について解説していく.

戻り値の型

戻り値の型

戻り値の型は, 関数が値を返す[a]際のデータ型のことである.

[a]戻り値については後で詳しく解説する.

再掲

型	キーワード	サイズ
文字データ	char	1 バイト
符号付き整数	int	2 バイト
浮動小数点	float	4 バイト
倍精度浮動小数点	double	8 バイト
値なし	void	

表 13: 基本のデータ型

引数リスト

引数とは

関数を呼び出した際に関数に対して渡す値. 引数の宣言は通常の変数と同様に行う.

Listing 18: 引数の例

```
1        int func (int n, int m) {
2            int ans;
3            ans = n * m;
4            return ans;
5        }
```

ソースコード 18 では,1 行目の () で引数を宣言している. 関数を呼び出して使用する際は必ず関数名の後ろに渡したい値を記述しなければならないので注意すること.

戻り値

戻り値とは

関数が処理を終えた後に返す. 関数の最後にある return 戻り値で値を返すことができる.

Listing 19: 戻り値の例

```
1 int func(void) { 引数が必要ない場合は,voidと記述する.
2     int a = 1;
3     a += 5;
4     return a;
5 }
```

ソースコード 19 では, 関数の中に return a; と書いているため, この関数が呼び出されたときに a が返り値として返る.

値渡しと参照渡し

関数に引数を渡す場合, 以下の 2 つの方法がある.

1. 値渡し
2. 参照渡し

1 つ目の値渡しとは, 引数に値がコピーされる. なので, 呼び出し元の値は影響を受けない.2 つ目の参照渡しでは, アドレスが引数にコピーされる. そのため, 呼び出された関数で値を変更すると呼び出し元の値も変更される.

Listing 20: 参照渡しの例

```
1       void swap(int *i, int*j) {
2            int temp;
3            temp = *i;
4            *i = *j;
5            *j = temp;
6        }
```

ソースコード 20 では, 関数に 2 つの変数のポインタを渡しているので引数が指す実際の値が入れ替わる.

再帰

再帰の使用例

ここでは, 再帰の全体像を確認するために以下のコードを記述しておく.

Listing 21: reur.cpp

```
#include <iostream>
using namespace std;

void recursion(int i);
int main() {
    recursion(5);
    return 0;
}

void recursion (int i) {
    if(i == 0) return;
    cout << i << endl;
    recursion(i - 1);
    return 0;
}
```

再帰とは

再帰関数は非常に難しい分野のため, 無理に理解しようとする必要はない.

再帰とは

関数の中で自分自身（その関数自身）を呼び出すことを再帰と呼ぶ.

Listing 22: 再帰の一般形の一つ

```
戻り値の型  func（引数）{
    if(ベースケース){
        return ベースケースに対する値;
    }

    再帰呼び出し
    func(次の引数);
    return 答え;
}
```

再帰には様々な書き方であるため, ソースコード 22 のような書き方をしなくても良い.

構造体

構造体の使用例

以下のコードでは構造体の例である.

Listing 23: str.cpp

```
#include <iostream>
using namespace std;

struct Student {
    string name;
    int age;
    int id;
} Student1;

int main() {
    cin >> Student1.name >> Student1.age >> Student1.id;
    cout << "以下の人物を登録しました.\n" << "名前␣:␣" << Student1.name << endl << "年
        齢␣:␣" << Student1.age << endl << "学籍番号␣:␣" << Student1.id << endl;
}
```

構造体の基礎 i

構造体とは

構造体とは, メンバと呼ばれる互いに関連のある 2 つ以上の変数で構成される複合データ型である.
配列は全て同じデータ型であるが, メンバはそれぞれ異なる型を持つことができる.

Listing 24: 構造体の定義

```
1        struct タグ名 {
2         型 メンバ1;
3         型 メンバ2;
4         .
5         .
6         .
7         型 メンバn;
8         } 変数リスト;
```

構造体の変数の宣言

struct　タグ名　変数リスト;

構造体の基礎 ii

構造体型をを定義したら, 後はその型の変数をいくつでも生成することができる.

構造体を用いる手順

構造体を用いる際の手順は以下のとおりである.

1. どんなデータを扱うのか考える.
2. 変数として実態を作る.
3. 実際に数値や文字を格納

構造体のアクセス方法

構造体のメンバにアクセスするためには, 以下の形式で書かなければならない.
変数名. メンバ名

構造体を用いるメリット

構造体を用いることでプログラムを綺麗に分かりやすく記述することができる.

構造体の基礎　iii

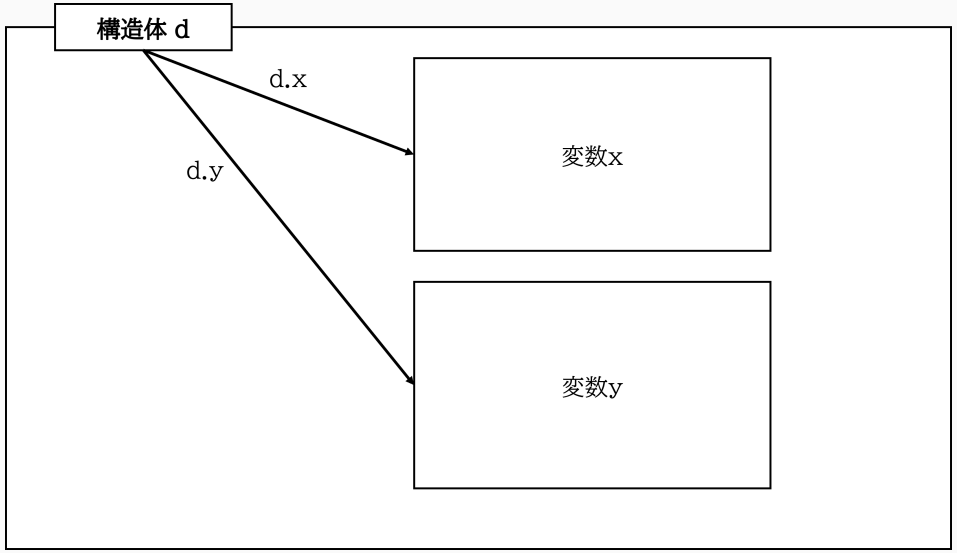

図 8: 構造体とメンバ変数の関係

構造体へのポインタ i

Listing 25: str_p.cpp

```
#include <iostream>
#include <cstring>
using namespace std;

struct s_type {
    int i;
    char str[80];
}s, *p;

int main() {
    p = &s;
    s.i = 10;
    p -> i = 10;
    strcpy(p -> str, "私は構造体が好きです.");

    cout << s.i << "␣" << p -> i << "␣" << p -> str << endl;
    return 0;
}
```

構造体へのポインタ ii

ポインタを介したメンバ変数へのアクセス

ポインタ変数 -> メンバ変数;
上記に書いた方法をアロー演算子と呼ぶ.

アロー演算子を用いることでポインタを用いたプログラムを書く際, とても読みやすくなるため積極的に使用していきたい.

Listing 26: アロー演算子

```
1 pdを構造体ポインタ型変数, dを構造体変数とする.
2 aをdのメンバ変数とする.
3
4 *(pd).a;
5 pd -> a;
```

ソースコード 26 の二つは同じ意味である.

参考文献

参考文献


ハーバード・シルト, 独習 C++ 第 4 版 (2016)


"KUT-PG 高知工科大学 プログラミング集団 Wiki*" https://wikiwiki.jp/kut-pg/ (アクセス日: 2024-4-16)