第1回C++講座

目次

・C++の特徴
・宣言・定義/変数の生存期間
・ポインタ/構造体
・C++でC言語を使うときの違い
・関数オーバーロード
・名前空間
・コンソール入出力
・参照

C++の特徴

•C言語の拡張。Cと同じようにも記述できる

•クラス・継承・テンプレートなどの新機能が追加され、C言語では難しかった問題を簡単に記述できる

•(C++にしかできない機能はおそらくない)

・STL(Standard Template Library)というライブラリが言語標準で用意されている

コンパイルとリンク

実装ファイル(.cや.cpp)を個別に
機械語に翻訳していく

↓

関数定義や必要のない変数の定義は無視
そういった関数・変数が「ある」と宣言されていればOK
(ここまでがコンパイル)

↓

複数の実装ファイルをまとめる(リンク)
このとき、無視されていた関数や変数の定義を確認する

関数定義や必要のない変数の定義を無視するのは、
関数定義や変数定義がどのファイルに記述されているかわからないため

ヘッダファイルには各種宣言と、クラス定義のみを記述する

変数・関数・クラスの宣言と定義

宣言

詳細は明示しないが、翻訳単位では「存在する」ことを宣言している

ローカル変数は常に「宣言・定義」となり「宣言のみ」は存在しない。

グローバル変数は[extern]修飾子のついた変数が「宣言のみ」に該当する

宣言は何度でもできる！

定義

変数・関数・クラスの定義とは、それらの詳細を明示することである。

また、定義があれば当然「存在する」ので、同時に宣言の意味も含まれる

変数の定義とは、指定した型の詳細通りにメモリ上に領域を確保し、変数に割り当てることである

定義は翻訳単位で一度だけ！

・関数呼び出しは、引数と戻り値がわかればよい(=詳細は必要ない)ので、宣言でOK
・変数の宣言のみなら、クラス・構造体の宣言があればよい
・変数の定義は、メモリ上に領域を確保するのでクラス・構造体の定義が必要
・ポインタ型変数の定義は、メモリサイズが固定なのでクラスの宣言だけでよい
・宣言のみで定義が存在しない場合にリンクエラーが発生する(コンパイルは成功する)

スコープと変数の生存期間

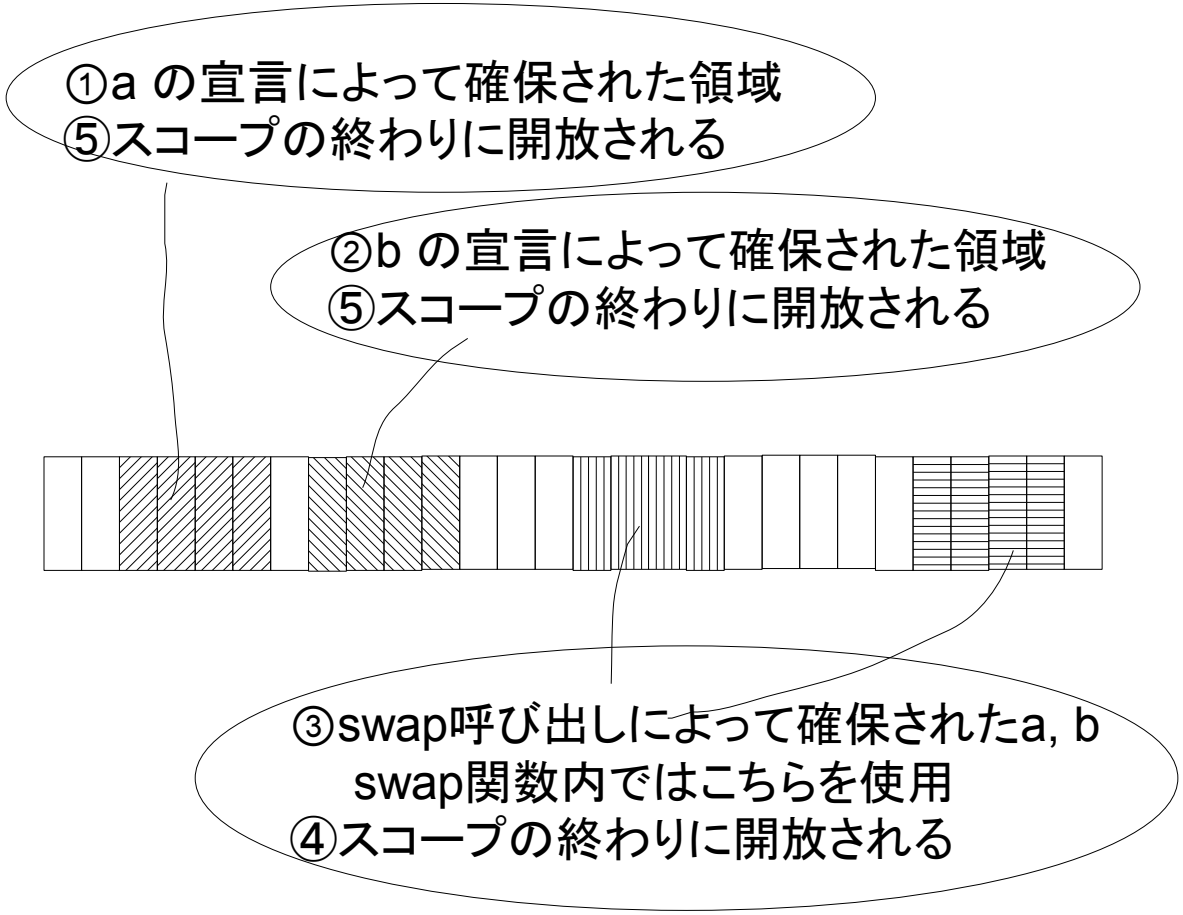
- スコープとは{}で区切られた区間のことである
- 変数は定義(宣言)した場所からスコープの終了まで生存する
- mallocで確保した領域ははfreeするまで生存

	初期化タイミング	生存期間	アクセス
ローカル変数	宣言した場所	宣言したスコープ内	スコープ内からのみ
静的変数	初めてその場所を実行するとき	プログラム終了まで	スコープ内からのみ
グローバル変数	プログラム開始直後	プログラム終了まで	どこからでも
静的メンバ変数	プログラム開始直後	プログラム終了まで	アクセス修飾子による

変数の生成・生存期間と関数

処理の手順を記述する
呼び出されるたび、引数の値をコピーした変数を生成する

```
void swap(int a, int b)←③
{
  int temp = a;
  a = b; b = temp;
}                          ←④

int main(void) {
    int  a;   ←①
    int  b;   ←②
    swap(a, b)←③

    return 0;  ←⑤
}
```



*上のソースではswap関数を
実行しても値は交換されない！

ポインタ

・ポインタ”型”
・「型名*」で使用する(ex. int*, double*)
・int型のポインタではなく [int*型]

保持しているのは4バイトの整数で、その整数に「メモリ上のアドレス」という役割を与えているだけ

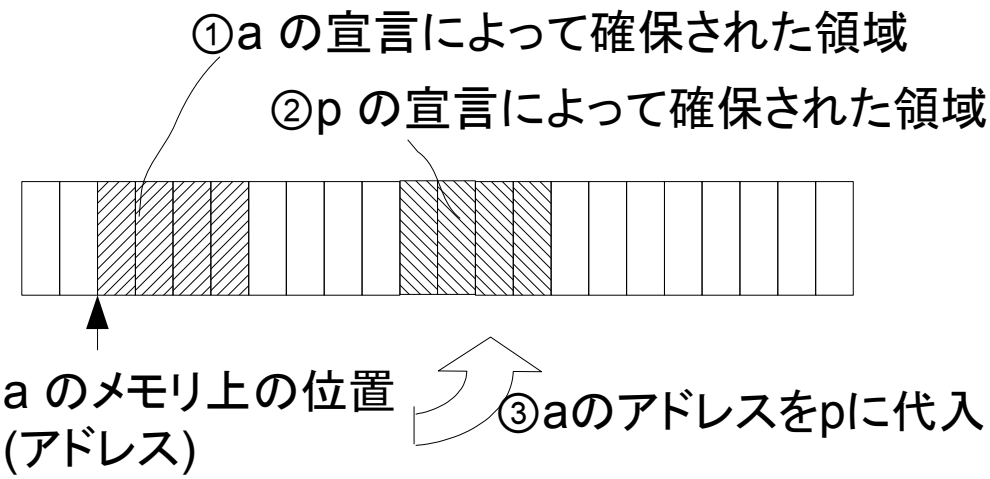
ポインタ型はすべて同じ大きさ！

保持しているアドレスを使って、
ポインタの指している変数を操作できる

何型かの情報が必要！

＊操作するまでクラスの定義は必要ない
＊配列は、型の大きさが必要なので要定義
＊つまり、クラス宣言だけでポインタ定義可

```
int main(void) {
    int  a;   ←①
    int* p;   ←②
    p = &a;←③
    *p = 3; ←④

    return 0;
}
```



構造体

```
struct Point {
    int x, y;
};

int main(void)
{
    struct Point p1;
    p1.x = 3;
    p1.y = 2;

    return 0;
}
```

定義文

```
struct 構造体名 {
    メンバ変数;
    ⋮
};
```

struct Point型変数の宣言

メンバ変数へのアクセスはドットを使う

構造体の定義は新しい型を作ることに相当する。
メンバ変数の扱い以外は普通の変数と同じ。

構造体を引数に取る場合の注意

```
#include <stdio.h>
struct TestData {
    int * pointer;
    int array[100];
};
void test(struct TestData t) {
    *(t.pointer) = 0;
    t.array[0] = 9;
}
int main() {
    int a = 10;
    struct TestData data = {&a};
    test(data);
    printf("%d\n",  a);
    return 0;
}
```

- ポイント

(1)ポインタ渡しは、ポインタ変数の値がコピーされている。

(2)ポインタ渡しでなくても、メンバにポインタを持つ構造体を渡す場合は、そのポインタを経由して値が書き換えられる可能性がある

＊構造体のメンバ自体は変化していない

(3)ポインタ渡しを使う状況は、

(a) 変数の値を変更するとき

(b) 配列を用いるとき

(c) 値渡しでは処理速度に問題があるとき(一般に参照を用いる)

(4)ポインタ渡しでは、値を変更しない場合は常に const を用いると良い

(問) 左の例で、data変数の array[0]はどうなっているか？

C++でC言語を使うときの違い

```
#include <cstdio>
int main()
{
    int x, y;
    puts("Hello world!!);

    bool b = true;
    return 0;
}
```

(1) 標準ヘッダに ”.h” をつけない

(2) C言語の標準ヘッダは、もとの名前の先頭に
' c ‘ をつける (例）<cstring>, <ctime>
※<stdio.h> のようにしても問題ない

(3) 関数の引数を省略すると、
C言語：void または int / C++ : void

(4) スコープの途中で変数が宣言できる

(5)構造体変数の宣言にstruct が必要ない

(6) bool 型と true, false
int 型では 0が偽, 0以外が 真値扱い

```
#include <stdio.h>
void test() {}
int main()
{
    test(1);
}
```

(問) 左のプログラムは、C言語とC++でどのような違いがあるか

関数の多重定義

```
#include <cstdio>
void func()          {puts("void"); }
void func(int)       {puts("int");}
void func(long)      {puts("long");}
void func(float)     {puts("float");}
void func(double){puts("double");}
void func(char*)   {puts("char*");}

int func()   {puts("void");} //エラー

int main()  {
        func();   func(1);  func(3L);
        func(1.0f); func(2.0);
        func("aiueo");
        return 0;
}
```

ポイント

(1) 同じ名前で、引数が違う関数を複数定義できる

(2) どの関数を使用するかは自動で判別される。型の一致する関数が優先的に選ばれ、なければ暗黙の型変換をして関数を選ぶ。

(3) 暗黙の型変換等で、どの関数を使用するか曖昧な場合はコンパイルエラーが起こる

(4) 引数が同じで返り値が違う関数は定義できない

どの関数を使用するか判別できることが大事！

＊暗黙の型変換が2度行なわれることはない
(詳細はクラスのときに)

(問)左のプログラムから void func(int) を消去するとどうなるか。またキャストするとどうなるか

演習

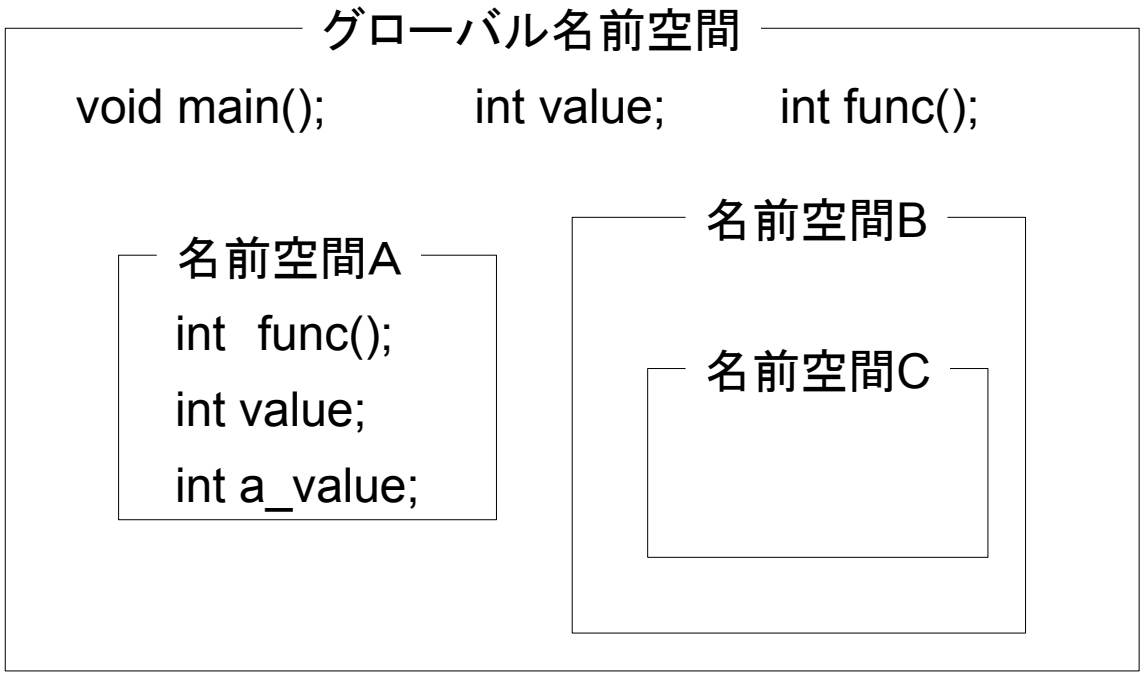
```
#include <cstdio>
void function() {};
int function() {return 0;}

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

1. 左のプログラムはどのような問題があるか
2. int 型、double 型の交換関数 swap を作成せよ
3. 左上、右下のxy座標を持つRectangle構造体を作成し、swap関数を作成せよ
4. 名前、ID、位置座標、矩形当たり判定を持つGameObject構造体を定義せよ
5. bool colligion(GameObject, GameObject);
   で定義される衝突判定関数を作成せよ
6. 三角形の面積を求める area_tri 関数の宣言を
   (a)三辺の長さから求める場合
   (b)正三角形の場合
   について記述せよ。(宣言だけでよい)

名前空間

変数や関数は必ずいずれかの名前空間に所属している。
名前空間とは、変数や関数の所属を区別するもので、
名前の衝突の防止を目的としている

＊これまでに使ってきた変数・関数はグローバルな名前空間に所属している



名前空間

```
// quickLib.h
namespace  quickLib
{
     extern int x;
     void print();
     void read();
     void write();
}
```

```
// secureLib.h
namespace secureLib
{
     extern int x;
     void print();
     void read();
     void write();
}
```

- **ポイント**

(1) 同じ名前の変数、同じ引数の関数を多重定義できる
(2) 複数のライブラリをインクルードしても競合しない
(3) 使用するときは、”::” を用いて以下のように書く

```
int main()
{
    quickLib::print();        // quickLib の print()
    secureLib::print();       // secureLib の print()
    return 0;
}
```

(4) アドレス解決演算子
“::” は、どの名前空間の要素かを指定する
（クラスでも使う）

(5)以下の場合はアドレス解決演算子が不要。

(a)自分の所属する名前空間の要素

(b)自分より上位の名前空間の要素

(c)using で指定した名前空間の要素

にアクセスするとき。

コンソール入出力

```
#include <iostream>
int main()
{
    int x;
    std::cout << “数値を入力してください：“;
    std::cin >> x;
    std::cout << “入力されたのは “ << x << “です\n”;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cout << “数値を入力してください：“;
    cin >> x;
    cout << 入力されたのは “ << x << “です\n”;
    return 0;
}
```

- **ポイント**

(1)左のプログラムは全く同じ内容

(2)<iostream> ヘッダ
C言語の<stdio.h> に相当する

(3) cout, cin
標準入出力
cout, cin は、std という名前空間に定義されている

(4) using namespace std;
は、std 名前空間にある全要素に、
:: なしでアクセスするためのもの

(5) 入出力する変数の型は
cin, cout により自動的に判別される

※using namespace の使用 は、名前空間の利点である名前の衝突の回避を無効化してしまう。

※C++では、入出力処理をストリームという共通のインターフェースで行なう

演習

- 名前空間myStd に、main という名前の関数を作成せよ。

  コンパイルし、その結果の意味を考察せよ
- 名前空間 myStd に、power関数を作成せよ。関数はdouble 型の変数xと、int 型の変数 yを受け取り、xのy乗を返す。(y >= 0)
- 二つのdouble 型変数をメンバに持つPoint構造体を作成し、

  これを用いて二点の距離を返す関数を作成せよ。
- 三つのPoint構造体をメンバに持つ Triangle構造体を作成し、その面積を返す関数を作成せよ。

※C++標準入出力を使用すること

ヒント：ヘロンの公式

三辺の長さをa, b, c とする。

2s = a + b + c、面積をAとすれば、

A = (s * (s – a) * (s – b) * (s – c))^(1 / 2)

参照

- 参照は”型”ではない
- 「型名&」で使用する(ex. int&, double&)
- 宣言以外は普通の変数と同じように使う

参照は、ある変数に別の名前をつけることに相当する

初期化が必須！

参照変数が個別に領域を確保することはなく、初期化した変数と同じ領域を使う

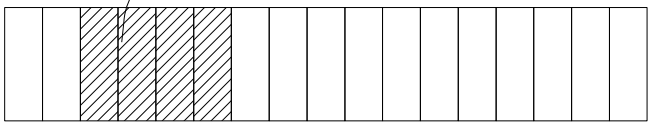
初期化時に使用領域を決めている！

＊初期化とは、変数を宣言した時に値を設定することである
＊変数の宣言後に別の文で値を格納することを代入という

```
int main(void) {
    int  a;   ←①
    int& ref=a;  ←②

    a = 3;
    ref = 4;

    return 0;
}
```

①a の宣言によって確保された領域

②refもこの領域を使う



*変数の生存期間は参照した変数に由来
つまり、ref は a の生存期間と同じ
→参照はあくまで領域を借りてるだけ！

参照と関数

```
#include <iostream>
void swap(int &a, int & b) {
   int temp = a;
   a = b;
   b = temp;
}

int main() {
    int x = 3, y = 4;
    swap(x, y);
    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

swapのaはx と、bはyとおなじ領域を使用する

⇩

a,bを入れ替えるとmain関数のx,yも入れ替わる

- 変数をコピーしないので、値渡しに比べて速度面で有利
- ポインタ同様、関数内で呼び出し元の変数を操作できる

演習

```
int* getIntPointer() {
    int temp = 100;
    return &temp;
}
int& getIntReference() {
    int temp = 100;
    return temp;
}
int getInt() {
    int temp = 100;
    return temp;
}
```

1. 左のプログラムの間違いを指摘せよ。また、どのような問題が起こるか予想し、実際に試してみよ
2. 前回作成したswap関数を参照に変更してみよ