オブジェクト指向と
ゲームプログラミング

○　C++編 － 第１３回　名前空間　○

名前空間

クラス名が異なれば、同じ名前のメンバ関数やメンバ変数を定義できるように、名前空間(namespace)が異なれば、同じ名前のクラス、構造体、列挙体、グローバル関数、グローバル変数を定義できます。これにより、チームでプログラム開発を行っている場合やさまざまなライブラリとの名前の衝突を防ぐことができます。
独自に定義したクラスや関数は、デフォルトではグローバルな空間に配置されます。外部のライブラリを多用すると、それだけ識別子がグローバルな空間に増え、名前の衝突する可能性が高くなります。そのような場合に、名前空間を用います。

C++では、標準ライブラリも"std"という名前空間内で定義されており、名前の衝突がないように配慮されています。独自に文字列を管理するクラスstringを定義したとしても、標準ライブラリのstd::stringとは違うものである、と区別されます。これが名前空間の役割です。

名前空間の定義は、キーワードnamespaceで空間名を宣言し、{}で囲みます。カッコ内で宣言された識別子は、指定された名前空間内に配置されます。

```
namespace 名前 {
    …
}
```

「名前」の部分に、名前空間の名前を指定します。また、名前空間の"}"には、セミコロン";"を付けません。

```
namespace game_package {

// キャラクタークラス定義
class Character {
public:
    void Move(const double add_x, const double add_y);
};

// 関数宣言
bool CollisionDetection(const Character& chara1, const Character& chara2)

// 定数定義
const int   MAX_CHARA = 1024;

}
```

名前空間の中に配置したクラスや関数の定義を外部(ソースファイル)で記述する場合も、namespaceで囲みます。

```
namespace game_package {

// キャラクタークラス － 移動処理
void Character::Move(const double add_x, const double add_y)
{
    // 省略
}

// 衝突検出
bool CollisionDetection(const Character& chara1, const Character& chara2)
{
    // 省略
}

}
```

名前空間の内部では、通常どおり識別子を使用することができますが、名前空間の外部からは、スコープ解決演算::で空間名を指定しなければ使用できません。前述の外部定義をスコープ解決演算::を用いた場合、以下のようになります。

```
// キャラクタークラス ― 移動処理
void game_package::Character::Move(const double add_x, const double add_y)
{
    // 省略
}

// 衝突検出
bool game_package::CollisionDetection(const Character& chara1, const Character& chara2)
{
}
```

この例では、game_packageという名前空間が存在します。この名前空間にアクセスするには、スコープ解決演算子::を用いなければなりません。名前空間によって、game_package内のクラス、関数、変数などの識別子は、空間外との名前の衝突の可能性がなくなります。ただし、名前空間は必ずグローバルな位置で定義しなければなりません。関数内などのローカルな位置で宣言することはできません。
また、同じ名前を持つ名前空間は、複数に分割して宣言することができます。別の場所や別のファイルで宣言しても、空間名が同じであれば、ひとつの名前空間とみなされます。

名前空間は、以下のように入れ子にすることもできます。

```
namespace application {
namespace game {
namespace package {

class Character {
    // 省略
};

}
}
}
```

入れ子により階層化された名前空間へのアクセスは相対的になります。ある名前空間の中の名前空間へアクセスする場合、すべての名前空間の外であれば名前空間のもっとも外側から順番に、目的の名前空間までスコープ解決演算子で指定します。上記の場合は、application::game::package::Characterとなります。名前空間の中からのアクセスであれば、自分の名前までは省略することができます。

名前空間は、別な名前空間名に置き換えることができます。前述のような長い空間名になった場合は、省略した名前を付けることもできます。

```
// 名前空間application::game::packageにgame_packという別名を付ける
namespace game_pack = application::game::package;
```

別名により、application::game::package::Characterは、game_pack::Characterとも記述できます。

無名の名前空間

名前空間には、名前のない名前空間があります。これは、C言語のstaticなグローバル変数に等しいものです。この名前のない名前空間を「無名(または匿名)の名前空間」と呼びます。
通常、グローバル変数はすべての場所からアクセスすることができます。異なるファイルであっても、キーワードexternを使ってアクセスできます。
しかし、無名の名前空間にアクセスできるのは、同一ファイルのみです。同一ファイルであれば、グローバル変数としてアクセスすることができますが、別のファイルから無名の名前空間にアクセスすることはできません。たとえexternを用いても、無名の名前空間にアクセスすることはできません。

```
namespace {
    int   g_Count;  // カウンタ
}
```

無名の名前空間で定義された変数g_Countは、同一ファイル内からアクセスすることはできますが、ほかのファイルからアクセスすることはできません。「extern int g_Count」を宣言しても、アクセスすることはできません。

C++では、グローバル変数を定義する場合、staticなグローバル変数ではなく、無名の名前空間を用いるように推奨しています。

名前空間の中に定義された名前の使用

名前空間の中に定義された名前(識別子)を使用するには、以下の３つの方法があります。

１．スコープ解決演算子::を使って完全な名前を記述する
２．using宣言を使って使用する名前空間および識別子を指定する
３．using指令を使って指定した名前空間をグローバル領域まで引き上げる

１のスコープ解決演算子::を使う方法は、外側の名前空間から順番に、目的の名前空間までスコープ解決演算子で指定するというものです。

```
application::game::package::Character   chara1, chara2;
game_package::CollisionDetection(chara1, chara2);
```

しかし、ある名前空間の識別子を何度も使う場合、この方法で識別子を指定するのは面倒と感じる場合もあります。そのような場合は、２のusing宣言を用います。

```
using 名前空間名::識別子;
```

キーワードusingに続き、名前空間名を記述し、スコープ解決演算子::を付けてその空間内の識別子を指定します。この構文を用いることで、その名前空間の指定した識別子を空間名なしで使用できます。ただし、同じ空間内でも、using宣言されていない識別子を空間名なしで使用することはできません。

```
using application::game::package::Character;
using game_package::CollisoionDetection;

Character   chara1, chara2;
CollisionDetection(chara1, chara2);
```

usingの適応範囲は、変数と同様にスコープによって異なります。グローバルな位置で宣言された場合は、その行以降からファイルの終わりまで、ある関数内で宣言された場合は、その行以降から関数の終わりまでが適用範囲となります。

３のusing指令は、特定の名前空間そのものをグローバル領域に追加するものです。

```
using namespace 名前空間名;
```

名前空間名には、追加する名前空間を指定します。これを用いれば、指定した名前空間の識別子すべてを空間名なしで使用することができます。

```
using namespace game_package;

Character   chara[CHARA_MAX];
CollisionDetection(chara[0], chara[1]);
```

using指令の適応範囲は、変数と同様にスコープによって異なります。グローバルな位置で宣言された場合は、その行以降からファイルの終わりまで、ある関数内で宣言された場合は、その行以降から関数の終わりまでが適用範囲となります。

using指令で問題となるのは、グローバルな領域への名前の追加であるということです。関数内などの特定のスコープ内ならともかく、グローバルな領域でusing指令を用いると、名前の衝突が発生してしまうことがあります。
特に、ヘッダファイルでusing指令を使用すると、ヘッダファイルをインクルードするソースファイルで思わぬ副作用を招きます。ヘッダでのusing指令は、グローバルな領域へ名前が追加されます。一度追加した名前を削除することはできません。そのため、チームでプログラム開発を行っている場合や外部のライブラリを使用している場合には、名前の衝突が起こってしまうことがたびたびあります。名前空間によって名前の衝突をなくそうとしているのに衝突を招いてしまっては、名前空間の意味がまったくなくなってしまいます。