オブジェクト指向と
ゲームプログラミング

○　C++編 － 第１２回　例外処理　○

例外処理

プログラムにとって必要不可欠でありながら、扱いが難しいものの１つに例外(エラー)処理が挙げられます。

例外処理(exception handling)とは文字どおり、「めったに起こらないこと(例外)が起こったとき、その例外を検出した場所から、その例外に対処する場所に処理を引き継ぐこと」をいいます。本来想定されていない状況に陥ったときに、そこから回復させるという目的があります。
例外が起こるのは、関数が実行されているときです。特に、メイン関数から別の関数が何度も呼び出された状態で発生します。このとき、その関数内だけで完全に例外への処理を行うことができるかというと、必ずしもそうではありません。場合によっては呼び出し元の関数に例外処理を頼まなくてはならないこともあります。このように、例外が発生した場所と例外処理を行う場所が異なっている場合には、例外を呼び出し元に通知する手段が必要になります。

例外構文を持たないＣ言語などでは、関数の戻り値とif文を組み合わせる方法がよく使われます。例外の内容を表す定数(エラーコード)を定義しておき、関数内で例外が起きた場合、それに対応したエラーコードを返すというものです。返された値をif文で調べ、例外が起こっていない場合は通常の処理を行い、例外が起こっていた場合はelse節で例外処理を行います。
たとえば、Funcという関数を実行したときに例外が発生したなら０が戻るようにして、

```
if(Func() != 0) {
    // 処理成功
} else {
    // 処理失敗
}
```

とする方法です。これはもっとも素直なプログラムですが、注意しないとたくさんの分岐が出現し、その結果、難解なプログラムになったり、実現しようとしている本来の処理が何であるのかを読み取りにくくしてしまう可能性があります。また、「通常処理」と「例外処理」が入り混じるため、プログラマの思考の負担が重くなるという問題があります。

```
// 処理１を実行
if(Func1() != 0) {
    // 例外が起こっていなければ、処理２を実行
    if(Func2() != 0) {
        // 例外が起こっていなければ、処理３を実行
        if(Func3() != 0) {
            // 例外が起こっていなければ、処理４を実行
            if(Func4() != 0) {
                // すべての処理が成功
            } else {
                // Func4関数の例外処理
            }
        } else {
            // Func3関数の例外処理
        }
    } else {
        // Func2関数の例外処理
    }
} else {
    // Func1関数の例外処理
}
```

戻り値を使って例外を呼び出し元に通知する方法には、いくつかの欠点があります。
１つ目は、戻り値をもっぱら例外通知に使用するようになってしまうことです。戻り値としてエラーコード以外の値を返さなければならない状況はたくさんあります。
２つ目は、前述のように関数の戻り値を常にif文で調べ、処理を分岐させなければならないので、記述が若干複雑になってしまいます。
３つ目は、例外の内容を詳しく伝えるのが難しいことです。ほとんどの場合、整数値で例外の内容を

伝えることになりますが、より詳しい内容(たとえば、例外が起こった状況などに関する情報)が欲しい場合には、整数値だけでは不足してしまいます。整数値の代わりに例外クラスを作成してそのオブジェクトを返す、という方法もありますが、この方法だと例外が起こっていないときでも、オブジェクトを作成して返さなければならないので、実行効率が悪くなってしまいます。
　これらに加え、C++では例外の通知が非常に難しい状況がいくつか存在します。たとえば、演算子のオーバーロードです。

```
X operator+(const X& a, const X& b);     // Xクラスどうしを加算する
X a, b, c;                               // a, b, cはXクラスのオブジェクト
X d = a + b + c;
```

このようなとき、どこで例外が発生し、それにどう対処するか、解決するのが困難になります。

例外の通知

　C++では、例外通知と例外処理が構文レベルで導入されています。これらを利用すれば、例外を検出した場所からその例外を処理する場所にスマートに移行することができます。

　例外を通知するには、キーワードthrowを使用します。throwの直後に通知のための式を記述します。

```
throw 式;    // 例外の通知
```

　式は、整数でも文字列でも構造体やクラスのオブジェクトでも構いません。throwは、どんなオブジェクトでも例外の通知として使用することができます。つまり、例外処理に必要な情報をほとんど何でも送出することができるのです。

```
enum   ERROR_CODE { OK, FILE_NOT_FOUND, …省略…};  // エラーコード
struct EXCEPTION  {…省略…} ex;                    // 例外情報構造体
class  Exception  {…省略…};                       // 例外クラス

throw FILE_NOT_FOUND;        // エラーコードを投げる
throw GetLastError();        // GetLastError関数(Win32API)の戻り値を投げる
throw "Invalid Params";      // 文字列を投げる
throw ex;                    // 構造体を投げる
throw Exception();           // クラスのオブジェクトを生成して投げる
```

　throwによりなんらかのオブジェクトを送出することを「例外を投げる」といいます。エラーや障害が発生したとき、throwで適切なオブジェクトを投げることで、例外に関する情報とともに例外処理を行う場所にプログラムの実行を引き継ぐことができます。
　例外処理のあとは、通常処理に復帰することも、そのままプログラムを終了することもできます。

投げられた例外の処理

　throwにより投げられた例外の処理は、連続した２つのブロック、try{}とcatch(){}で行います。

　tryブロックは、エラーが起こりそうな処理や例外が投げられそうな処理を記述します。つまり、このブロックは、処理を試みる(tryする)部分になります。このブロック内でエラーが起きた場合、例外を投げる(または投げられる)ことになります。

```
try {
    if(関数１ != 成功)
        throw 例外１;
    関数２(関数２内で例外が投げられる可能性がある)
}
```

　tryブロックに続くのが、catchブロックです。tryブロックで投げられた例外は、このブロックで捕まえる(catchする)ことができます。catchブロックは、戻り値のない関数のようなもので、

```
catch(型 引数名) {
    例外処理
}
```

というように、引数をひとつだけ持つことができます。そして、受け取った例外を参照して、例外処理を行うわけです。
　catchブロックは、複数続けて記述することができます。複数のcatchブロックがある場合、最初のブロックから順に調べられ、投げられた引数の型と例外の型が最初に一致したブロックが実行されます。

catchブロックの引数の型は、以下の場合に投げられた例外の型と一致するものとみなされます。

1．完全な一致
2．引数の型を基底クラスとする派生型
3．引数のポインタ型に変換可能なポインタ型

２と３により、catchブロックの順序によっては、意図したcatchブロックが実行されない場合があります。たとえば、派生クラス型を受け取るcatchブロックより前に、基底クラス型を受け取るcatchブロックがある場合、派生クラス型の例外が投げられても、２により基底クラス型のcatchが実行されてしまいます。このような場合は、必ず派生クラス型のcatchを基底クラス型より前に記述します。

また、「catch(...)」と記述すると、すべての型の例外を捕まえることができます。これをcatchブロックの最後に記述しておけば、それまでに一致しなかったあらゆる例外を捕まえることができます。

なお、catchブロックでオブジェクトを捕まえる場合は参照型にしておきます。参照型でない場合、コピー作成のオーバーヘッドが発生するだけでなく、仮想関数が正しく呼び出せなくなります。

(１)例外の投げ方

```
throw 式;
```

(２)投げられた例外の受け取り方

```
try {
    // 例外が投げられる可能性がある
    // 処理を記述する
}
```

tryブロック
tryブロックには、例外を投げる可能性のある処理を記述します。例外が投げられると、処理を中断し、あとに続くcatchブロックに例外を受け取れるところがあるかどうか調べられます。

```
catch(型 引数名) {
    // ここで例外を処理します
}
catch(...) {
    // ここで例外を処理します
}
```

catchブロック
catchブロックは、複数記述することができます。tryブロックで投げられた例外は、引数の型が一致する最初のcatchブロックで捕まえられ、その引数が投げられた例外を参照するようになり、処理が再開されます。
catch(...)と記述すると、すべての型の例外を捕まえることができます。この部分は記述しなくても構いません。

(３)処理の流れの例１

```
class exception {}  // 例外クラス

try {
    処理１;
        throw exception();
    処理２;
}
catch(exception& ex) {
    // exを使った例外処理
}
catch(...) {
    // 例外処理
}
次の処理
```

(４)処理の流れの例２

```
try {
    処理１;
    処理２;
    処理３;
    処理４;
}
catch(exception& ex) {
    // exを使った例外処理
}
catch(...) {
    // 例外処理
}
次の処理
```

(５)例外がキャッチされない場合

```
try {
    ここでint, exception以外の例外を投げる
}
catch(const int n) {
    // nを使った例外処理
}
catch(exception& ex) {
    // exを使った例外処理
}
次の処理
terminate関数へ
```

例外指定

関数の宣言時あるいは関数定義の一部で、例外指定を行うことができます。例外指定を行うと、その関数内で投げられる例外の型を制限することができます。例外指定は、以下の書式で行います。

関数プロトタイプ throw(型, 型, …)

型をひとつも指定しなかった場合は、この関数が直接的にも間接的にも(この関数から呼び出された関数からも)throwを実行しないものとみなします。

```
void Func1() throw(int, const char*, exception&);    // 投げる例外の指定
void Func2() throw();                                // 例外禁止
```

指定されていない型を投げると、実行時エラーとなり、unexpected関数が呼び出されます。unexpected関数は、デフォルトではterminate関数を呼び出してプログラムをただちに終了させ、制御をＯＳに戻します。これにより、想定していない例外が投げられた場合、プログラムを強制終了させることができます。

なお、例外指定がない場合は、すべての型の例外を投げることができます。

標準例外

C++標準ライブラリには、標準例外クラスstd::exceptionがヘッダファイル<exception>に定義されています。

```
class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

std::exceptionクラスは、継承されることを前提に設計されています。また、すべてのメンバ関数で空の例外指定throw()が宣言されています。そのため、オーバーライドしても、メンバ関数内で例外を投げることはできません。
std::exceptionを継承した派生クラスを作成した場合、必要ならば、仮想関数whatがエラーメッセージ(文字列)を返すようにオーバーライドします。std::exceptionから派生したクラスのオブジェクトを例外として投げれば、

```
catch(const std::exception& ex) {
    std::cerr << ex.what() << std::endl;
}
```

のように捕まえることができます。また、std::exceptionから継承されたいくつかの例外クラスがヘッダファイル<stdexcept>に定義されています。

例外処理の仕組み

例外を処理するためにプログラムがたどる過程を詳しく説明します。

tryブロック内またはその内部から呼び出された関数でthrowにより例外が投げられると、まず現在の関数内から該当するtryブロックの終わりを探し始めます。現在の関数の内部にtryブロックの終わりが見つからない場合、その関数の呼び出し元の関数に戻り、その内部で該当するtryブロックの終わりを探します。そこでも見つからない場合、さらにその関数の呼び出し元の関数に戻り……というように、関数を次々と巻き戻していくようにたどっていきます。
このとき重要なのは、スコープの範囲から出る自動変数のオブジェクトに対してデストラクタが呼び出され、変数の破棄が行われる点です。これは、関数がreturn文を実行するときによく似ています。こうすることで、オブジェクトの解放漏れを防いでいるのです。

適切なtryブロックが見つかると、今度はtryブロックの直後にあるcatchブロックを探します。投げられた例外の型と一致した場合、そのcatchブロックに移行しますが、一致しない場合、今度はもうひとつ外側にtryブロックがあるものとし、適切なcatchブロックが見つかるまで巻き戻しを続けます。最後まで適切なcatchブロックを見つけることができなかった場合は、terminate関数が呼び出されます。terminate関数は、デフォルトではabort関数を呼び出します。abort関数は、プログラムをただちに終了させ、制御をＯＳに戻します。