Boost.勉強会 #6 札幌 (2011-11-05)

C++ TIPS 1 #INCLUDE編

概要

- 主に cppll ML でご紹介してきた tips をC++の仕様をより掘り下げた形でまとめ直してみました。
- 今回は#includeにフォーカスした内容です。

C++ Tips

#INCLUDE

#includeってなに？

- 指定されたファイルの内容を#include指令の記述の位置に展開する機能です。
  - コンパイルの直前にプリプロセスによって展開されます。
  - コンパイラに対して展開するのであって元のファイルは書き換わりません。
  - ISO/IEC 14882 では 2003 年版および 2011 年版ともに §16.2 Source file inclusion (JIS X 3014 では「ソースファイルの取込み」)で記述されています。

#includeってなに？

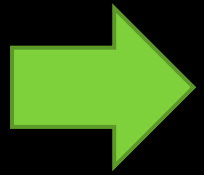
【a.h】

```
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
```

【a.cpp】

```
…
#include "a.h"

int main()
{
    …
}
```



【展開結果】

```
…
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}

int main()
{
    …
}
```

#includeってなに？

- Java の import や C# の using とは似て非なるモノです。
  - 最近の他言語の類似した機能とその本来の目的は似たようなものなのですが、CおよびC++の#include指令は仕様上、プリプロセスによってただ機械的に指定されたテキストファイルをその場所にぶちまけるだけです。

#includeってなに？

- cppコマンドを使って #include の展開結果を確認すると #line 指令も挿入されます。
  - この#line指令によりコンパイルエラーの位置情報や __FILE__, __LINE__ といったマクロが展開前の位置で指し示されるようにします。

#includeってなに？

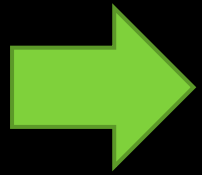
【a.h】

```
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
```

【a.cpp】

```
…
#include "a.h"

int main()
{
    …
}
```



【展開結果】

```
… // ←a.cppの19行目だとします。
#line 1 "a.h"
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
#line 20 "a.cpp"

int main()
{
    …
}
```

#includeってなに？

- プリプロセスは #include 指令以外にも #define 指令や #if 指令などを処理します。
- 他のヘッダファイルに依存しているヘッダファイルは必ずその依存先のヘッダファイルを#includeしておきましょう。
  - これをちゃんとやっておかないとメンテナンス性が著しく低下します。

#includeってなに？

- #include する時の <hoge.h> と "hoge.h" の違い
  - ファイル探索範囲に Ａと Ｂがあったとして、 <hoge.h> は Ａしか見ないが、 "hoge.h" はまず Ｂ を見た上で見つからなかったら Ａからもファイルを探す。
    - ただし Ｂ はそもそも存在しないことも許されており、その場合、ファイル探索範囲の違いはなくなる。
    - Ａ と Ｂ のファイル探索範囲およびその指定方法は処理系依存。
  - 一般的には時の <hoge.h> は標準ライブラリおよび処理系に組み込んだライブラリのヘッダを読み込む場合に使用され、"hoge.h" はユーザープログラムのヘッダを読み込む場合に使用されます。

#includeってなに？

- ファイルの指定に利用可能な文字列の種類については ISO/IEC 14882 の2003 年版では§ 2.8 Header names 、 2011 年版では同名の§ 2.9 (JIS X 3014 では「 § 2.8 ヘッダ名 」)で記述されています。
  - ISO/IEC 14882 の2003 年版では§ 2.2 Character sets 、 2011 年版では同名の§ 2.3 (JIS X 3014 では「 § 2.3 文字集合 」)も参照のこと。
  - 簡単且つ乱暴にまとめるとASCII文字(タブや改行も含む)は使っていいけど、日本語を直接使用するのはダメで、国際文字名を使いたい場合は ¥uXXXX あるいは ¥UXXXXXXXX の形式で指定。(Xは16進数文字)
  - 例外としてヘッダ名を < と > で囲む場合は > が、二つの " で囲む場合は" が使えない。

インクルードガード

- 依存関係にあるヘッダファイルをヘッダファイル内で#includeしていると、複数のヘッダファイルを#includeした際に、共通で依存関係にあるヘッダファイルの#includeが重複し、コンパイラからするとその中で行われる定義も重複することになりエラーになってしまいます。

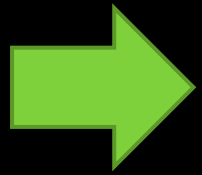
インクルードガード

【a.h】
```
#include "c.h"
```

【b.h】
```
#include "c.h"
```

【c.h】
```
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
```

【a.cpp】
```
#include "a.h"
#include "b.h"
```



【展開結果】
```
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
```

定義が重複してしまいエラーになる！

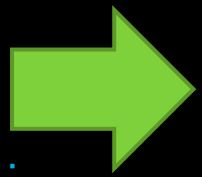
インクルードガード

- そこで出てくるのがインクルードガードと呼ばれるテクニックです。
- #if 指令と #define 指令を使って２つ目以降は展開されないようにします。

インクルードガード

【c.h】

```
#if !defined(c_h)
#define c_h
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
#endif
```



【中間展開イメージ】

```
#if !defined(c_h)
#define c_h
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
#endif
#if !defined(c_h)
ここは #if の条件が成立しないので
展開されない。
#endif
```

【最終展開イメージ】

```
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
```

インクルードガード

- インクルードガードで #define するマクロは名前はなんでもいいのですが他と名前が被るといろいろまずいことが起きることが予想されますので、まず被らないだろうって名前にする必要があります。
- #includeされる側ではなく#includeする側でインクルードガード行うこともあります。
  - #includeする側でインクルードガードするのをどこかでうっかり忘れてしまうリスクがあるので多用はされていないようですが。

インクルードガード

【a.h】

```
#if !defined(c_h)
#define c_h
#include "c.h"
#endif
```

【b.h】

```
#if !defined(c_h)
#define c_h
#include "c.h"
#endif
```

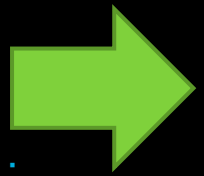
インクルードガード

- 拡張仕様で #pragma once と記述しておくだけでインクルードガードをやってくれるコンパイラもあります。

インクルードガード

【c.h】

```
#if !defined(c_h)
#define c_h
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
#endif
```



【c.h】

```
#pragma once
inline int nabs(int a)
{
    return 0 <= a ? -a: a;
}
```

インクルードガード::使い分け

- コンパイルの速度的には#pragma once や#includeする側で行う手製インクルードガードがよいとされています。
- 移植性の面では#includeされる側で行う手製インクルードガードがベストです。
- 直接は手を入れたくないあるいは手を入れられない第三者から提供されるヘッダファイルにインクルードガードが施されていない場合は#includeする側で行う手製インクルードガードしかありません。
  - 手製インクルードガードでラップしたヘッダファイルを使うのもありです。

インクルードガード::補足

- 定義だから重複エラーになるのであって、宣言だけなら重複エラーになりません。

意図的な再インクルード

- ヘッダファイルでいろいろな小細工をしたい場合に意図的に再インクルードさせる場合などもあります。
  - 躊躇無く複数のヘッダファイルを使える状況下ではまず必要になることはないテクニックです。
  - このテクニックを利用する場合、#pragma once などに頼らず自前で且つ適切にインクルードガードを行う必要があります。
  - 特定のマクロ群のOn/Offを切り替えさせる為。
  - ひとつのヘッダファイルに複数の機能を持たせる為。
    - マクロの指定により異なる内容のヘッダファイルのように振る舞わせたり。

ソースファイルのインクルード

- C++のテンプレートの機能を使ったコードを書くと一般的にはソースファイルに書くべきコードまでヘッダファイルに書かざるを得ませんが、ソースファイルをヘッダファイルから #include することで、形だけは従来どおりにヘッダファイルとソースファイルへ書き分けることができます。

データファイルのインクルード

- 移植性が高く且つ超お手軽なデータ埋め込み方法としても #include は使えます。

データファイルのインクルード

```
…
int data[] = {
#include "data.csv" // 42,42,42,
};
…
```

ストリームのインクルード

- 役に立つことはまずないですが /dev/tty (linux)だの con (windows)だのといったストリームをインクルードすることも可能です。
  - プロンプトの類いが出せないのが辛いところです。

ストリームのインクルード

```
…
int confidential_value = // この値はコンパイル時に手入力する。
#if defined(__WIN32__) || defined(_WIN32)
#include <con>
#else
#include <dev/tty>
#endif
;
…
```

#include __FILE__ について

- そのファイル自身をインクルードしようとして #include __FILE__ とするのは構文的は間違いではないが、規格上も実際も #include で指定する形式と __FILE__ が一致している保証がなく、またカレントディレクトリの扱いが処理系によって異なる問題もある為、規格的には有効に機能する保証はない。

#line

- 明示的に利用することで次のようなことができます。
  - コンパイル環境の物理的なファイルパスの隠蔽。
    - __FILE__ がどのような形のナロー文字列になっているのかは処理系依存。
    - ユーザー名が含まれるパスでコンパイルして実行形式ファイルに __FILE__ として埋め込まれる事態を防げます。
  - 同じファイル内でのブロックの明示。
    - __function__ 疑似マクロなどが使える場合はあまりその必要性はありませんが。
- #line で指定できる行番号は C++03だと 1～32,767 の範囲で C++11だと1～2,147,483,647の範囲になります。この値域から外れる数値が指定された場合の動作は未定義となります。
- ISO/IEC 14882 では 2003 年版および 2011 年版ともに § 16.4 Line control (JIS X 3014 では「行制御」)で記述されています。

C++ Tips 1 #include編

質疑応答

C++ Tips 1 #include編

ご清聴ありがとうございました。