Лекция 10. Препроцессор

В чем риск использования
и когда все же норм?

Использование макросов

- Лучше не пользовать макросы, если вы можете этого не делать. Они унаследованы из C, а вместо них в C++ есть:
  - Константы
  - Шаблоны
  - inline функции

- Макросы обрабатываются препроцессором до компиляции. Поэтому они ничего не знают ни о пространстве имен, ни о типах.

- Отладочные средства (например, дебагер) и средства разработки часто не могут заглянуть внутрь макроса.

Применимость макросов

- Когда макросы полезны
  - Включение файлов и #include guard
  - Конкатенация токенов
  - Отладочный вывод
  - Условная компиляция
  - Повторяемый код (boost.preprocessor)*

- Есть три типа макросов: директивы, макро-константы и макро-функции

Определение констант

- Простой синтаксис

#define <identifier name> [value]

- Символ # должен быть первым в строке (кроме пробельных), можно также вставлять whitespace между # и define.
- Можно определить и без значения - используется для условной компиляции.
- Будьте осторожны с контекстом – оборачивайте в скобки.

```
1.  #define PI         3.1415926
2.  #define PI_PLUS_1 PI + 1
3.  ...
4.  auto x = PI_PLUS_1 * 5; // results in PI + 1 * 5 (Oops!)
5.  // do like this
6.  #define PI_PLUS_2 (PI + 2)
```

Условная компиляция

- Незаменима при кроссплатформенной разработке
- Помогает настроить версии используемых библиотек, платформу, операционную систему и т.д.
- Можно использовать для временного комментирования кода

```
1.  #ifdef WIN32
2.  #   ifdef _MSC_VER > 1800
3.      //...
4.  #   endif // _MSC_VER
5.  #else
6.  //...
7.  #endif // WIN32
8.
9.  // boolean expressions
10. #if defined (__linux__) && !defined(NDEBUG)
11. //...
12.
13. #if 0
14. /* ... */ //...
15. #endif
```

Include guard

- Позволяют избежать повторного включения хедера в код единицы трансляции
- Тот же эффект, что и #pragma once

```
1.  #ifndef __MY_INCLUDE_H__
2.  #define __MY_INCLUDE_H__
3.  // C++ header body
4.
5.  #endif // __MY_INCLUDE_H__
```

Макро функции

- Определяется почти как константа

#define MACRO_NAME(arg1, arg2, ...) [code to expand to]

- Нельзя перегружать*
- Не работает с рекурсией
- Не понимает шаблонов* Запятая является разделителем параметров шаблона, на ‘<‘ и ‘>’ макрос не обращает внимание
- Не вычисляет аргументы перед вызовом - подставляется как есть. Избегайте side effect в аргументах

```
1.  #define MAX(a,b) a > b ? a : b
2.  MAX(x + 4, x--);            // results in
3.  x + 4 > x-- ? x + 4 : x--; // ok, let's wrap args
4.
5.  #define ADD_ONE(x) (x) + 1
6.  ADD_ONE(var) * 5 // results in (x) + 1 * 5
7.
8.  // solution:
9.  #define ADD_ONE(x) ((x) + 1)
```

Требование ';' после макроса

- Если уж пишем макро-функции, хотелось бы, чтобы к ним были применимы те же синтаксические правила вызова, что и к обычным функциям:

```
1.  // any multiple statements macro
2.  #define SWAP(x, y) (x) ^= (y); (y) ^= (x); (x) ^= (y);
3.
4.  // Example #1: this should work as expected
5.  if (x > y)
6.      SWAP(x, y);
7.  do_something();
8.
9.  // Example #2: This should not result in a compiler error.
10. if (x > y)
11.     SWAP(x, y);
12. else
13.     SWAP(y, z);
14.
15. // Example #3: This should not compile
16. do_something();
17. SWAP(x, y) // no semicolon
18. do_something();
```

Требование ';' после макроса

```
1.  #define SWAP(x, y) x ^= y; y ^= x; x ^= y;
2.
3.  if (a > b)
4.      SWAP(x, y); // oops!
5.
6.  #define SWAP(x, y) { x ^= y; y ^= x; x ^= y; }
7.
8.  if (a > b)
9.      SWAP(a, b); // compilation error
10. else
11.     //...
12.
13. // solution, requires ';' in the end (+ use ‘(‘ and ‘)’)
14. #define SWAP(x, y) do { x ^= y; y ^= x; x ^= y; } while(0)
15. // or
16. #define SWAP(x, y) (x ^= y, y ^= x, x ^= y, (void)0)
```

- «Фейковый» цикл с постусловием
- (statement1, statement2, …, (void)0)

Превращение токена в строку

```
1.  #define STR_IMPL(X) #X
2.  #define STR(X) STR_IMPL(X)
3.
4.  #define SHOW(X) \
5.  do{ cout << STR_IMPL(X) << ": " << (X) << endl;\
6.  }while(0,0)
7.
8.  #define VALUE 2.718281828459045
9.
10. int main()
11. {
12.     int x(2), y(5);
13.     SHOW(x + y); // x + y: 7
14.
15.     cout << STR_IMPL(VALUE) << endl; // VALUE
16.     cout << STR(VALUE) << endl; // 2.718281828459045
17. }
```

Конкатенация токенов

- Аналогично «стрингизации» требуется делать трюк со вложенным макросом

```
1.  #define CONCAT_IMPL(X, Y) X##Y
2.  #define CONCAT(X, Y) CONCAT_IMPL(X, Y)
3.
4.  #define DECL_PTR(T) using CONCAT(T, _ptr) = unique_ptr<T>
5.
6.  DECL_PTR(int);
7.
8.  int main()
9.  {
10.     int_ptr p;
11. }
```

Multiline macro

- Нужны исключительно для облегчения чтения кода

```
1.  #define Verify(expr)                                          \
2.  do{ if (!(expr))                                              \
3.  {                                                             \
4.      std::stringstream ss;                                     \
5.      ss << "Verification failed: " << STR(expr) << “. ”  \
6.      << BOOST_CURRENT_FUNCTION << "\" in " <<                  \
7.          __FILE__ << ":" << __LINE__;                          \
8.      LogError(ss.str());                                       \
9.      throw verify_error(ss.str());                             \
10. } } while (0, 0)
```

Predefined defines

- __LINE__ номер текущей строки
- __FILE__ название текущего файла
- __FUNCTION__ (или BOOST_CURRENT_FUNCTION) название функции
- __linux__, WIN32 и т.д. - настройка платформы
- __COUNTER__ самоинкрементирующаяся константа

```
1.  #define CONCAT_IMPL(x, y) x##y
2.  #define MACRO_CONCAT(x,y) CONCAT_IMPL(x,y)
3.  #define LOCK(name) \
4.      scoped_lock MACRO_CONCAT(lock, __COUNTER__)(name);
5.
6.  void foo()
7.  {
8.      LOCK(mutex)
9.      // ...
10.     LOCK(mutex)
11.     // ...
12. }
```

assert

```
#ifdef NDEBUG
#define Assert(expr)                \
do                                  \
{                                   \
    if (0, 0)                       \
    {                               \
        (void)(expr);               \
    }                               \
} while (0, 0)                                          \
#else                                                   \
#define Assert(expr)                                    \
do{                                                     \
if (!(expr))                                            \
{                                                       \
    std::stringstream ss;                               \
    ss << "Assertion failed: " << #expr << " at \""\
    << BOOST_CURRENT_FUNCTION << "\" in " <<            \
    __FILE__ << ":" << __LINE__;                        \
    AssertLogError(ss.str());                           \
    assert(expr);                                       \
} } while (0, 0)
#endif // NDEBUG
```

Вариативный макрос*

- Макросы не понимают шаблоны. Запятая, отделяющие параметры шаблоны воспринимается как отделение параметров макроса.
- Вариативный макрос не позволяет перебрать параметры, только отдать дальше все их вместе

```
1.  // invocation with 2 arguments
2.  PROCESS(map<int, double>)
3.
4.  #define PROCESS(...)                                          \
5.  void process(processor& proc, __VA_ARGS__ const& object)\
6.  {                                                            \
7.      typedef __VA_ARGS__ type;                                \
8.      //...
```

Перегрузка макросов*

- В языках C и C++ нет перегрузки макросов.
- Но бывает, что такая перегрузка могла бы существенно улучшить выразительность кода, поэтому...

```
1.  // what do we have:
2.  #define FOO2(x,y) ...
3.  #define FOO3(x,y,z) ...
4.
5.  // what do we want:
6.  #define FOO(x,y) ...
7.  #define FOO(x,y,z) ...
```

Перегрузка макросов(2)*

- Можно все же найти способ:

```
1.  #define FOO2
2.  #define FOO3
3.
4.  #define GET_MACRO(_1,_2,_3,NAME,...) NAME
5.  #define FOO(...) GET_MACRO(__VA_ARGS__,             /
6.                             FOO3, FOO2)(__VA_ARGS__)
7.
8.  FOO(x,y) ...
9.  FOO(x,y,z) ...
```

На десерт:

Pimpl идиома

```
1.  // owner.h
2.  struct owner
3.  {
4.      owner();
5.      owner& operator=(owner const&);
6.      //...
7.  private:
8.      struct impl;
9.      unique_ptr<impl> pimpl_;
10. };
11.
12. // owner.cpp
13. struct owner::impl
14. {
15.     void foo();
16.     //...
17. };
18.
19. void owner::impl::foo(){/*...*/}
```

- Осторожно: объявление `struct impl* pimpl_` будет иметь другой смысл. В чем же различия?

Pimpl. Бонус #1: время сборки

```
1.  struct my_type
2.  {
3.      //...
4.  private:
5.      some         sm_;
6.      complicated cp_;
7.      classes      cl_;
8.  };
```

- Благодаря невидимости лишних объявлений удается значительно сократить время компиляции.
- Все такие объявления потребуются лишь в cpp-файле с объявлением impl’a

Pimpl. Бонус#2: разрешение перегрузки

```
1.  struct owner
2.  {
3.      void foo(string);
4.
5.  private:
6.      void foo(const char*);
7.  };
8.
9.  //....
10. owner ow;
11. ow.foo("hello");
```

- В таком коде нас может ждать сюрприз.
- Перенос перегрузки в impl позволит его избежать, поскольку закрытая функция будет релизована в impl классе.

Pimpl. Бонус #3: безопасность исключений

```
1.  struct owner
2.  {
3.      owner& operator=(owner const&);
4.      //...
5.  private:
6.      T1 t1_;
7.      T2 t2_;
8.  };
```

- Удастся ли здесь организовать строгую гарантию, если объекты типов T1 и T2 могут не обеспечивать даже базовую гарантию?
- Как Pimpl может помочь в этом случае?

Вопросы?