Занятие 1

Общие принципы написания программ на C++

1. Препроцессор

Препроцессор *это специальная программа, предназначенная для предварительной обработки исходного кода.*

Если пошагово рассмотреть, как обычно происходит преобразование исходного кода программы написанной C или C++ в машинный код, то можно увидеть, что самой первой программой, которая работает с исходным текстом, является препроцессор. В задачу препроцессора входят: обработка макросов, обеспечение условной компиляции, основанной на макросах, а так же включение заголовочных файлов или “хедеров”, (от headers (англ.)). Рассмотрим эти задачи более подробно.

1.1 Включение заголовочных файлов

Как все Вы знаете, файл является единицей хранения информации в файловой системе и не менее традиционной единицей компиляции.

Как правило, невозможно хранить законченную программу в одном файле (за исключением тривиальных случаев). Разбиение программы на файлы помогает подчеркнуть ее логическую структуру, облегчает ее понимание другими, и позволяет компилятору обеспечить эту логическую структуру. Когда единицей компиляции является файл, весь он должен быть перекомпилирован при внесении изменений в него, или в любой другой файл, от которого он зависит. Даже в случае программы скромных размеров, количество времени, потраченного на компиляцию, может быть снижено за счет разбиение программы на несколько файлов.

Пользователь предоставляет компилятору *исходный файл*. Как уже было сказано, сначала производится обработка файла препроцессором; то есть делаются макроподстановки (см. раздел 1.2), и выполняются директивы *#include* вставляющие код из заголовочных файлов. Результат обработки препроцессором исходного файла называется ***единицей трансляции.*** Рассмотрим пример (файлы book.h, book.cpp)

```
/* book.h */

class Book
{
private:
    char* name;
    char* author;
public:
    Book();
    Book(class Book&);
    const char* getName() const;
    const char* getAuthor() const;
};

/* book.cpp */

#include "book.h"

Book::Book()
```

```
{

    name = 0;
    author = 0;
}

...
```

Когда мы компилируем файл book.cpp, реально компилируется объединение файлов book.h, book.cpp (так как первой инструкцией в файле book.cpp стоит *#include "book.h"*) Задача объединить эти исходные в единицу трансляции – одна из задач препроцессора.

Стоит отметить, что выполнение препроцессором директивы #include не является подключением библиотеки, как многие из Вас думают. Это некоторая терминологическая тонкость, приводящая, порой, к непониманию Вас Вашими собеседниками и поэтому лучше избегать подобных высказываний. #include означает подключить лишь файл с исходным кодом, который в общем случае библиотекой не является.

1.2 Обработка макросов

Хотя использование макросов в C++, по утверждению Страуструпа, встречается гораздо реже, чем в C, необходимость в них все же есть.

Простой макрос определяется следующим образом:

```
#define NAME 1
```

Когда в программе встретится лексема *NAME,* после обработки программы препроцессором она будет заменена на 1. То есть, если на вход препроцессору подать программу вида

```
i = NAME;
```

на вход компилятору будет передано уже следующее:

```
i = 1;
```

Макрос можно определить с аргументами. Например:

```
#define MAX (x,y) ((x)>(y)? (x) : (y))
```

Соответственно, если в программе встретится выражение

```
std::cout << MAX(a,b);
```

после прохода препроцессора это выражение будет заменено на

```
std::cout << ((a)>(b)? (a) : (b));
```

Имена макросов нельзя перегружать. Кроме того, рекурсивные вызовы создают проблемы, которые препроцессор не может решить.

```
#define PRINT (a,b) std::cout << (a) << (b)
#define PRINT (a,b,c) std::cout << (a) << (b) << (c)
/* проблема: замещение, а не перегрузка */
```

```
#define FAC(n) ((n)>1)? (n)*FAC((n)-1) : 1 /* проблема: рекурсивный макрос */
```

При желании объявление макроса можно отменить. Для этого следует использовать директиву #undef. Например

```
#define ZERO 0
...
int x = ZERO /* правильно */
...
#undef ZERO
...
int y = ZERO /* ошибка, макрос ZERO не объявлен */
```

В макросах иногда есть необходимость использовать комментарии. В таких случаях лучше пользоваться стилем "/* ... */", а не "//". Дело в том, что иногда для компиляторов с C++ используются старые препроцессоры, написанные еще для C, и поэтому использование комментариев "//" может привести к проблемам.

Отмечу, что часто в C++ почти всегда можно найти замену макросам. Таковыми могут являться механизмы *const, inline, template и namespace.* Пожалуй, единственным случаем, когда использования макросов не избежать, является **условная компиляция**.

1.3 Условная компиляция, основанная на макросах

Рассмотрим простой пример (файлы person.h, teacher.h, student.h, main.cpp)

```
/* person.h */
#include <string>

class Person
{
private:
        long key;
        std::string last_name;
        std::string first_name;
public:
        Person();
        Person(const Person&);
        const std::string& getLastName() const { return last_name; }
        const std::string& getFirstName() const { return first_name; }
        void setLastName(const std::string&);
        void setFirstame(const std::string&);
};

/* student.h */

#include "person.h"

class Student : public Person
{
//...
```

```
};

/* teacher.h */

#include "person.h"

class Teacher : public Person
{
//...
};

/* main.cpp */

#include "teacher.h"
#include "student.h"

void main()
{

        Teacher teacher;
        Student student;
}
```

Если мы попробуем откомпилировать файл main.cpp, то получим что-то вроде "*Error E2238 person.h 5: Multiple declaration for 'Person'*" (текст будет зависеть от компилятора, но суть, я думаю, ясна). Проблема заключается в том, что описание класса Person встретилось компилятору 2 раза. Действительно, первый раз это описание встретилось в момент компиляции текста файла teacher.h, второй раз – в момент компиляции файла student.h. Выходов из сложившейся ситуации, по крайней мере, три. Первый заключается в перегруппировке классов по модулям. То есть мы, например, можем объявление всех классов поместить в один файл и на этом успокоиться. Минусом такого подхода является нарушение, с большой вероятностью, принципов модульности по которым бы не должны помещать в один модуль слабо связные классы. Второй вариант заключается в перемещение директив "*#include*". Мы можем, например, убрать из teacher.h директиву *#include "person.h"*, а вместо нее поставить *#include "student.h"*. Кроме того, нам придется удалить *#include "student.h"* из main.cpp, иначе мы получим множественное объявление не только для Person, но еще и для *Student*. Проблемы, возникающие при таком подходе, не менее гадкие, чем в первом случае. Во-первых, мы опять нарушаем один из принципов модульности по которому не следует связывать модули содержащие не связанные классы, и во-вторых нам приходится еще и думать над тем, как разместить *#include* для того что бы программа вообще скомпилировалась. Думать же над такой рутиной это тратить свое время зря. Третий вариант решения описанной проблемы заключается в использовании макросов. Суть здесь такова. Препроцессор, встречая директиву "*#define*" помещает имя, следующее за "*#define*" в свою временную таблицу и хранит это имя (а так же и сопоставленное с именем значение, если последнее задано) там до конца своей работы или до того момента, пока не встретит *#undef* для этого имени. Существуют средства, позволяющие выяснить, было ли объявлено некоторое имя, или нет (фактически занесено ли в таблицу препроцессора это имя, или нет). Это директивы #ifdef (если объявлено) и #ifndef (если не объявлено). Одновременно с установлением факта наличия/отсутствия имени в таблице, препроцессор определяется с тем нужно ли включать в код, подаваемый компилятору все начинающееся с *#ifdef/#ifndef* и до ближайшей директивы *#endif*. Рассмотрим пример.

```
#define MACRO_NAME /* задавать значение для макроса необязательно */

#ifdef MACRO_NAME
std::cout << “Макрос объявлен !”;
/* Это строка будет включена в код, передаваемый компилятору, так как макрос MACRO_NAME был объявлен */
#endif

std::cout < “Hello world!”;
/* Это строка будет включена в код, передаваемый компилятору, в любом случае, так как находится после  директивы #endif */

#ifndef MACRO_NAME
std::cout << “Макрос не объявлен”;
/* Это строка не будет включена в код, передаваемый компилятору, так как макрос MACRO_NAME был объявлен */

#endif
```

Таким образом, мы видим, что можем управлять тем, что будет реально передано компилятору с помощью директив *#define, #ifdef, #ifndef* и *#endif*. Применим это к нашей проблеме с повторным объявлением класса *Person*. Для этого немного изменим файл person.h (см. файл person2.h).

```
/* person.h */

#ifndef PERSON_H
#define PERSON_H

#include <string>

class Person
{
private:
    long key;
    std::string last_name;
    std::string first_name;
public:
    Person();
    Person(const Person&);
    const std::string& getLastName() const { return last_name; }
    const std::string& getFirstName() const { return first_name; }
    void setLastName(const std::string&);
    void setFirstame(const std::string&);
};

#endif
```

И рассмотрим, каким образом будет проходить компиляция файла main.cpp. Во-первых, файл будет передан препроцессору. Препроцессор, встретив директиву *#include "teacher.h”*, начнет рассматривать файл teacher.h, где наткнется на *#include "person.h"* и переключится на person.h. Первое, что увидит препроцессор в файле person.h будет директива *#ifndef PERSON_H*.

Так как препроцессор не найдет в своих таблицах макрос PERSON_H, он начнет рассматривать данный файл дальше где сразу встретит директиву *#define PERSON_H* выполняя которую занесет PERSON_H в свои таблицы. Стоит отметить, что при первом рассмотрении препроцессором файла person.h объявление класса *Person* будет занесено во входной файл для компилятора.

Опустим несколько шагов препроцессора и продолжим рассмотрение с обработки *#include "person.h"* в файле student.h. Начав рассмотрение person.h во второй раз, препроцессор опять наткнется на *#ifndef PERSON_H*. При просмотре своих таблиц препроцессор найдет такой в них такой макрос и пропустит все, до ближайшего *#endif*. Таким образом, объявление класса *Person* не будет включено во входной файл для компилятора повторно, и ошибки с повторным определением данного классы мы уже не получим.

Из всего вышесказанного можно сделать вывод, для того, что бы избежать проблем с повторным определением идентификаторов следует каждый заголовочный файл обрамлять триадой #ifndef, #define и #endif. Для имен макросов, используемых для предотвращения повторного включения определений удобно использовать имена, совпадающие с именем файла.

1.4 Дополнительные возможности препроцессора

Помимо рассмотренных директив, препроцессор так же распознает директивы *#if, #elif, #else, #line, #error, #pragma*. В дальнейшем мы рассмотрим некоторые из них на конкретных задачах.

2. Правила хорошего тона

Во всех не тривиальных проектах, в проектах, где участвуют более одного программиста, очень важным является соблюдение участниками проекта единых правил по наименованию переменных, классов, функций, а так же правил расстановки отступов, скобок, оформление комментариев и т.д. Все эти правила называются стилем программирования. Если группа, работающая над проектом, будет поддерживаться различных стилей, то каждому из ее участников придется тратить массу времени на понимание структуры элементарных конструкций, например, искать, где же кончается цикл или условный оператор. Учитывая тот факт, что со времени чаще всего не хватает, глупо тратить его на подобные вещи, гораздо проще договориться о едином стиле.

На сегодня существует несколько широко используемых стилей. Рассмотрим один из таких.

2.1 Общие правила

1) Один оператор – одна строка.

2.2 Правила именования

1) Называйте переменные осмысленно. Если, например, Вы хотите объявить переменных класса *Book* (книга), логично назвать ее *book*. Если это массив объектов класса Book, то название может быть, например, *books*. Неудачными названиями для рассмотренных случаев являются такие имена как *i* или *x*.
2) Используйте имена, состоящие из одной буквы лишь как переменные цикла (обычной практикой является использование *i,j,k*)
3) Называйте переменные, класса, функции и т.д. английскими словами. Если не знаете языка, пользуйтесь словарем.
4) Если имя переменной состоит из нескольких слов, разделяйте слова знаком подчеркивания или начинайте каждой не первое слово с большой буквы. Например,

old_objects или *personal_card*, или *accountability_disposal* (последнее, возможно и длинновато, но, зато, понятно, о чем идет речь).

5) Для переменных простых типов возможно использование префикса в виде первой буквы названия типа.
6) Используйте для имен переменных нижний регистр (за исключением пункта 4).
7) Имена классов подбирайте такими образом, что бы из имени было ясно, что представляет собой класс. Например, для абстракции студента подходящим названием будет *Student*.
8) Каждое слово в имени класса начинайте с большой буквы. Например, *Book, Student, RingRoutePurchaseDocument, Faculty, ObjectFactory*.
9) Для имен классов допустимым является добавление префикса “C” (class) или “T” (type)
10) Функции называйте таким образом, что бы из названия было ясно, что она делает. Например, insert(), createContractor() или getLastName().
11) Каждое слово в имени функции, кроме первого, начинайте с большой буквы. Остальные буквы должны быть маленькими.
12) Макросы и константы называйте осмысленно, большими буквами разделяя слова символом подчеркивания.

2.3 Отступы и скобки

Для операторов if, for, do, while, case, switch, try, catch и т.д. открывающаяся фигурная скобка ставиться на той же строке, где и записан сам оператор. Закрывающаяся скобка для данного оператора ставится на отдельной строке, на том же уровне что и сам оператор. Например:

```
for (;;) {
        ...
}

if (x > y) {
        ...
}
switch (i) {
        case 0: {
                ...
                break;
        }
        case 1: {
                ...
                break;
        }
}
```

1) Для функций, классов, структур, объединений, пространств имен открывающаяся и закрывающаяся скобки ставятся на отдельных строках на одном уровне с первой буквой заголовка функции, класса и т.д. Например.

```
void main()
{
        ...
}
```

```
class Person
{
        ...
}
struct Date

        ...{
}
namespace std
{
        ...
}
```

2) Все операторы одного блока (внутри операторных скобок одного уровня) должны быть записаны с одинаковым отступом. Отступ должен быть равен одной табуляции (последняя, если позволяет среда, должна равняться 2 или 4 пробелам). Каждый блок уровня вложенности n, должен иметь отступ от блока уровня вложенности n-1 ровно на 1 табуляцию.