Объявление (declaration) — задает имя функции, тип возвращаемого значения, число и тип аргументов, которые должны присутствовать при вызове функции.

```
double func(int, double);
```

Определение (definition)— объявление функции, в котором присутствует тело функции.

```
double func(int arg1, double arg2) {
 double result;
 ...
 return result;
}
```

Вызов функции

Вызов функции — это выражение, результатом которого является значение, возвращаемое функцией.

Формальные аргументы — те, которыми будет оперировать реализация функции.

Фактические аргументы — те, которые будут переданы в функцию при ее вызове.

Ссылки

Ссылка (reference) является альтернативным именем объекта.

```
int inc_pow_2(int &a) {
  a++;
  return a*a;
}

int main() {
  int value = 1;
  int result;
  result = inc_pow_2(value);
  // точка 1
}
```

Ссылки

```
void swap (int& p, int &q) {
   int temp = p;
   int p = q;
   int q = temp;
}
```

Возвращаемое значение

Функция должна возвращать значение, если она не объявлена как **`void`**.

Возвращаемое значение задается инструкцией **`return`**.

Возвращаемое значение

```
float & foo () {
  float b;
  return b;    // возвращать ссылку
               // на локальную переменную
               // НЕЛЬЗЯ!!!
}
```

Рекурсивные функции

```
int factorial(int n) {
  return (n == 1 || n == 0) ?
   1 : factorial(n - 1) * n;
}
```

Перегруженные имена функции

```
void print (long);
void print (double);
...
print (1L);    // вызывается print(long)
print (2.0);   // вызывается print(double)
print (1);     // ошибка
```

Указатели на функции

тип_возвращаемого_значения (*имя_указателя) (параметры);

```
int Get7() {
  return 7;
}

int Get8() {
  return 8;
}

int main() {
  int (*functionPointer)() = Get7;
  functionPointer = Get8;
  functionPointer();
  return 0;
}
```

Указатели на функции

```
bool TestAllPositive() {
  ...
  return true;
};

bool TestAllNegative() {
  ...
  return true;
};

bool TestAllNull() {
  ...
  return true;
};
```

Указатели на функции

```
int main() {
   ...

  bool (*testCases[])() =
   { TestAllPositive, TestAllNegative, TestAllNull };

  size_t testCasesSize =
   sizeof(testCases)/sizeof(testCases[0]);

  for (size_t i = 0; i < testCasesSize; i++) {
    testCases[i]();
  }

   ...
}
```

Указатели на функции

```
typedef bool (*unitTest)();

...

unitTest testCases[] =
 { TestAllPositive, TestAllNegative, TestAllNull };
```

Шаблоны функции

```
template<typename T>

T min (T a, T b) {
  return a < b ? a : b;
}
```

Шаблоны функции

```
template<typename T>

T min (T a, T b) {
  return a < b ? a : b;
}


double b = min (2.0, 7.2);

double b = min <double> (2, 7);
```

Аргументы по умолчанию

```
void print (int value, int base);
```

Аргументы по умолчанию

```
void print (int value, int base);

void print (int value, int base = 10);

print(7, 2);  // распечатать 7 в двоичной
              // системе счисления

print(9);     // трактуется как print(9,10);
              // распечатать 9 в десятичной
              // системе счисления.
```

Аргументы по умолчанию

```
void print (int value, int base);

void print (int value, int base = 10);

print(7, 2);  // распечатать 7 в двоичной
              // системе счисления

print(9);     // трактуется как print(9,10);
              // распечатать 9 в десятичной
              // системе счисления.

print(int value);
print(int value, int base);
```

Функции с неуказанным количеством аргументов

```
#include <cstdarg>

...

double CountAverage(int num...) {
  va_list args;
  double sum = 0.0;
  va_start(args, num);
  for (int i = 0; i < num; i++) {
    sum += va_arg(args, int);
  }
  va_end(args);
  return sum/num;
}

int main() {
  ...
  cout << "Average of 2, 3, 4, 5 = "
       << CountAverage(4, 2, 3, 4, 5) << endl;
  cout << "Average of 5, 10, 15 = "
       << CountAverage(3, 5, 10, 15) << endl;
  ...
}
```

Функция **main**

Сигнатура — **int main (int argc, char * argv [])**
либо **int main ().**

Свойства:

- нельзя вызывать рекурсивно;
- нельзя взять адрес ;
- нельзя объявлять и нельзя перегружать ;
- нельзя объявить как **inline**, **static** или **constexpr** ;
- в теле функции **main** не обязателен оператор **return**: при завершении функции **main** без оператора **return** эффект будет тот же самый, как при выполнении **return 0;**.
- тип возвращаемого значения функцией **main** не может быть выведен (**auto main() {...}** не разрешен).

Статические переменные в функциях

```
void showstat( int curr ) {
  static int nStatic;
  nStatic += curr;
  cout << "nStatic is " << nStatic << endl;
}

int main() {
   for ( int i = 0; i < 5; i++ )
      showstat( i );
}
```

Статические переменные в функциях

```
void showstat( int curr ) {
  static int nStatic;
  nStatic += curr;
  cout << "nStatic is " << nStatic << endl;
}

int main() {
   for ( int i = 0; i < 5; i++ )
      showstat( i );
}
```

```
nStatic is 0
nStatic is 1
nStatic is 3
nStatic is 6
nStatic is 10
```

Исходные файлы программы

Основные правила

Обеспечивайте лёгкость повторного использования единожды выполненной работы.

Исключайте влияния между несвязанными вещами.

Минимизируйте число связей между модулями.

Причины разделения на модули

- снижение времени компиляции
- выделение интерфейсов
- повторное использование
- совместная работа разработчиков
- удобство навигации

Модульность

Пользователь предоставляет компилятору исходный файл. Сначала производится обработка файла препроцессором; то есть делаются макроподстановки и выполняются директивы #include, вставляющие код из заголовочных файлов. Результат обработки препроцессором исходного файла называется единицей трансляции. Она и является тем, над чем работает компилятор.

Модульность

Для того, чтобы сделать возможной раздельную компиляцию, программист должен предоставить объявления, дающие информацию о типах, необходимую для анализа единицы трансляции отдельно от остальной части программы. Объявления в программе, состоящей из нескольких раздельно компилируемых частей, должны быть согласованы абсолютно также, как и в программе, состоящей из единственного исходного файла.

Примеры

file1.cpp

```
int gValue = 1;

void CalculateValue() { … }
```

file2.cpp

```
extern int gValue;

void CalculateValue();

void Process() {
  gValue = CalculateValue();
}
```

Переменная gValue (глобальная) и функция CalculateValue определены в file1.cpp.

При этом используются они в файле file2.cpp.

Согласно требований объект до использования должен быть объявлен. В файле file2.cpp присутствуют два объявления:

- extern int gValue объявляет перемеменную gValue типа int, не определяя её. То есть в данном случае указывается, что такой объект есть, но он определён в другом модуле.

- void CalculateValue() объявляет функцию CalculateValue, не определяя её. Сама функция также определена в другом модуле.

Особенности

Имена функций, классов, шаблонов, переменных, пространств имен и перечислений должны быть согласованы во всех единицах компиляции, если только эти имена явно не определены как локальные.

Любой объект в программе может быть определён только один раз. Он может быть объявлен сколько угодно раз, но типы должны совпадать.

Особенности

Если имеется имя, которое может быть использовано в единице трансляции, отличной от той, в которой оно было определено, то говорят, что имеет место внешняя компоновка (external linkage).

По умолчанию const и typedef имеют внутреннюю компоновку (internal linkage), то есть будучи объявлены как глобальные переменные (вне функций и пространств имён) они не видны в других единицах трансляции.

Особенности

Статические (static) объекты, объявленные вне функций, классов и пространств имён являются локальными для единицы трансляции и не могут быть использованы за её пределами даже после их объявления. Они также имеют внутреннюю компоновку.

Особенности

Встроенная (inline) функция должна быть определена в каждой единице транслации, в которой она используется. То есть её нельзя определить в одной единице трансляции и использовать в другой, даже предварительно объявив.

Заголовочные файлы

Обычно для обеспечения согласованности объявлений объектов они помещаются в заголовочный файл (h-файл), который затем вкючается в файлы реалиазации (cpp-файлы), в которых будут использоваться соответствующие объекты.

Заголовочные файлы

UserInterface.h

```
extern int gDisplayLines;

int  UserMenu();

void DisplayResults();
```

В нём объявлены глобальная переменная и две функции. Чаще всего в проекте присутствует соответствующий файл реализации (cpp-файл), в котором определены указанные объекты:

Заголовочные файлы

UserInterface.cpp

```
int gDisplayLines=22;

int  UserMenu() { … }

void DisplayResults() { … }
```

Заголовочные файлы

Client.cpp

```
#include "UserInterface.h"

int main() {
  gDisplayLines=40;
  UserMenu();
  ...
  DisplayResults();
}
```

Модули, желающие использовать указанную переменную или функции, должны подключить заголовочный файл, после чего они смогут использовать объявленные в нём объекты.

Заголовочные файлы

Client.cpp после обработки препроцессором

```
extern int gDisplayLines;
int  UserMenu();
void DisplayResults();

int main() {
  gDisplayLines=40;
  UserMenu();
  DisplayResults();
}
```

Директива #include предписывает компилятору выполнить полную подстановку указанного файла в текст программы в том месте, где была указана сама директива.

Стражи включения

Директива #include может распогалаться как в cpp-файле, так и в h-файле, обеспечивая таким образом возможность вложенных подстановок. В результате может случиться, что один и тот же файл будет включен дважды в файл реализации:

Стражи включения

file1.h

```
...
```

file2.h

```
#include “file1.h”
...
```

file3.h

```
#include “file1.h”
...
```

file.cpp

```
#include “file2.h”
#include “file3.h”
...
```

Стражи включения

Такое подключение приводит к необходимости дважды обрабатывать содержимое файла file1.h (поскольку он будет включен в file.cpp дважды – через file.2 и file3.h), что приводит к увеличению времени компиляции и, что более важно – не всегда допустимо.

Для того, чтобы обеспечить однократное включение файла file1.h используются стражи включения.

Классические стражи включения выполняются с помощью макросов и выглядят следующим образом:

Стражи включения

file1.h

```
#ifndef FILE1_H_

#define FILE1_H_


… тело заголовочного файла …


#endif
```

Стражи включения

Первая строка проверяет, определён ли макрос, соответствующий данному файлу (мы сами называем его подобно файлу – никаких стандартов на это нет). Если такой макрос не определён (как это должно быть при обработке первого вхождения этого файла в file.cpp), то определяется соответствующий макрос и обрабатывается содержимое файла. Если же такой макрос определён (как это должно быть при обработке последующих вхождений этого файла в file.cpp) всё содержимое файла до директивы #endif игнорируется.

Стражи включения

В Visual Studio (и ряде других компиляторов) есть упрощённая директива, позволяющая добиться того же результата:

file1.h

```
#pragma once

… тело заголовочного файла …
```