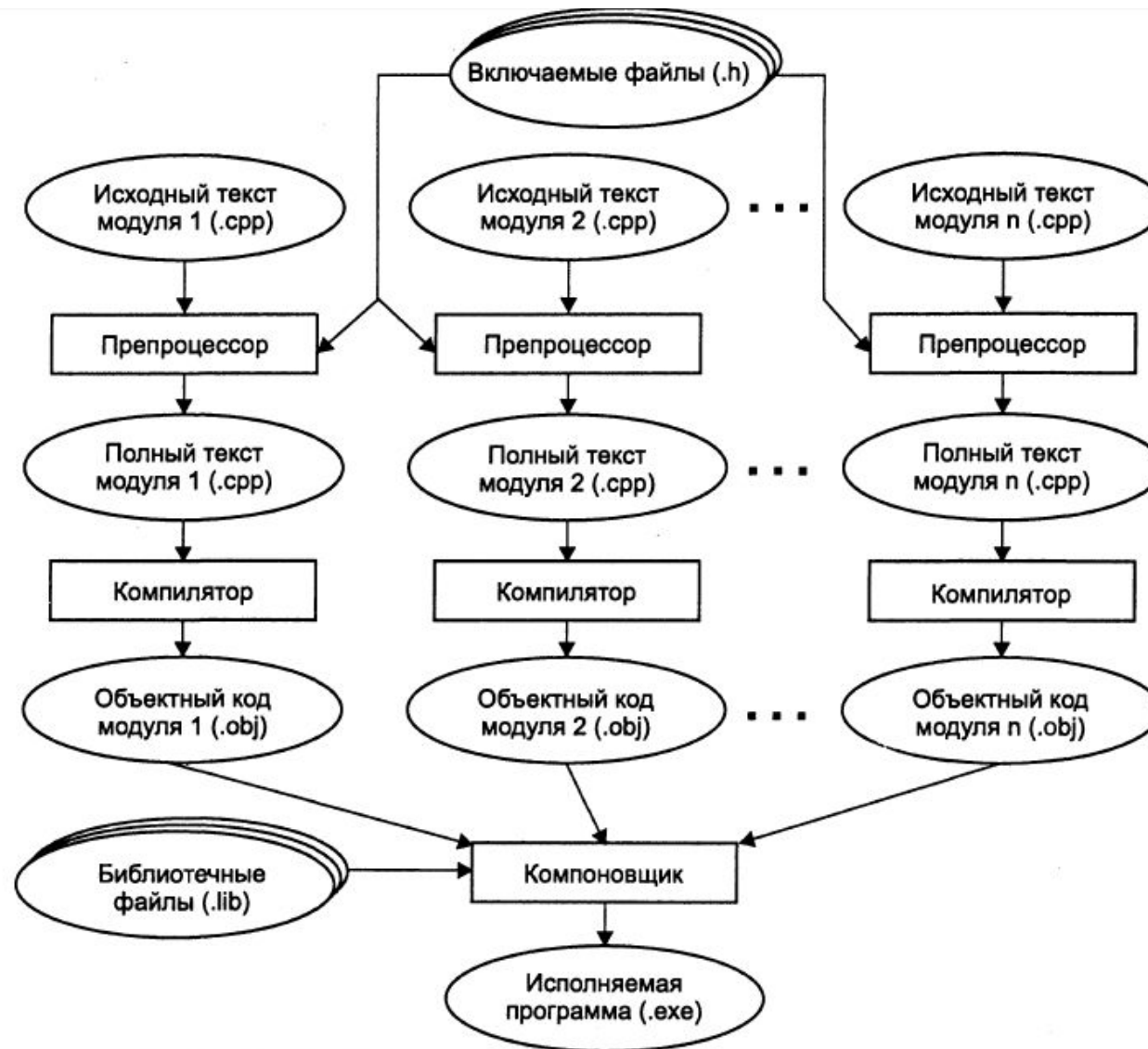


Создание многофайловых приложений

Причины использования многофайловых программ

- разделение работы разработчиков над проектом;
- разделение интерфейса и реализации;
- разделение по функциональной направленности;
- уменьшение времени перекомпиляции.

Этапы создания исполняемой программы



Типы файлов в проекте

- .c - исходный код на языке C;
- .C - исходный код на языке C++ в UNIX, где строчные и прописные буквы в именах файлов различаются;
- .crr, .cxx .cc – исходный код в ОС, где регистры не различаются (разные реализации могут использовать разные соглашения обозначения исходников);
- .h .hpp .hxx .H - заголовочные файлы для C и C++ программ также могут иметь разные расширения в разных реализациях (это одна из причин того, что стандартные заголовочные файлы C++ не имеют расширения).

Содержимое заголовочного файла

1. Определения типов, констант, переменных, встроенных функций, шаблонов, перечислений.
2. Объявления функций, данных, имен, шаблонов.
3. Пространства имен.
4. Директивы препроцессора.
5. Комментарии.

Объявление и определение переменных

//Файл А

```
int var = 0; // определение
```

```
void foo() {var++;}
```

//Файл В

```
extern int var; //объявление
```

```
extern int var10 = 10; //определение
```

```
int main() {
```

```
var = 10;
```

```
cout << var;
```

```
}
```

Внешние константы

```
// Файл А
//определение внутренней константы
const float pi = 3.14;
//определение внешней константы
extern const int var = 10;

//Файл В
//объявление внешней константы
extern const int var;
// к pi - нет доступа
```

Внешние функции

```
// Файл sum.cpr
//определение внешней функции
int sum(int a, int b){
return a + b;
}
```

```
//Файл с main
//объявление внешней функции
int sum(int, int); //м.б. использован extern
int main(){
//обращение к внешней функции
int c = sum(3, 5);
}
```

*extern используем, если проект собираем из нескольких файлов, но объявления не выносим в отдельный заголовочный файл

Внутреннее связывание для переменных

```
// Файл A.h
```

```
static int varInside;
```

```
int varOutside;
```

```
//Файл B
```

```
//#include "A.h"
```

```
static int var;
```

```
int main() {
```

```
cout << varOutside;
```

```
//cout << varInside; - нет доступа
```

```
}
```

Внутреннее связывание для функций

// Файл A

```
static int add(int a, int b) {  
    return a + b;  
}
```

//Файл B

```
//другая функция с такой же сигнатурой  
static int add(int a, int b) {  
    return a + b + 2;  
}
```

Именные пространства

```
namespace имя_пространства{  
описание_переменных_и_функций  
}
```

```
namespace n1{  
int a = 0;  
int add (int);  
}  
//...какой-то код...
```

```
namespace n1{  
int prod (int, int);  
}
```

Именные пространства (продолжение)

```
int  n1::add (int i) { /*тело функции*/ }
```

```
int n1::prod (int x, int y) { /*тело функции*/ }
```

```
int  add (int i) { /*тело другой функции*/ }
```

```
int main ()
```

```
{
```

```
n1::a = 5;
```

```
int x = n1::add(10);
```

```
int y = add(5); //другая
```

```
int z = n1::prod(x, n1::a);
```

```
}
```

Использование именных пространств

```
namespace n1{  
int a = 1;  
}
```

```
namespace n2{  
int a = 2;  
}
```

```
int main(){  
using namespace n1; //все имена из n1  
cout << a; //1  
using n2::a; //далее под "a" понимается n2::a  
cout << a; //2  
//float a = 5.5; error - redeclaration of a  
}
```

Использование именных пространств

использование одного пространства имен не отменяет действие другого

```
namespace n1{  
int a = 1;  
}
```

```
namespace n2{  
int a = 2;  
}
```

```
int main(){  
using namespace n1;  
using namespace n2;  
//cout << a; //error - ambiguous of a  
}
```

Синоним пространства имен

```
namespace A_very_long_namespace_name{  
/*...*/  
}
```

```
namespace my_nsp = A_very_long_namespace_name;
```

Вложенные пространства имен

```
namespace n1{  
    int a = 1;  
    int add (int);  
    namespace n2{  
        int a = 2, b = 10;  
    }  
}
```

```
int main() {  
    int a = 3;  
    int b = a; //3  
    b = n1::a; //1  
    b = n1::n2::a; //2  
    b = n1::n2::b; //10  
}
```


Глобальная видимость членов пространства, дополнение пространства неименованного

```
//fileA
namespace{
int var = 111;
}
namespace fA{
int var = 222;
}
void funcA(){
cout << var << fA::var << fA::scndVar;
} // 111222

//fileB
namespace fA{
int scndVar = 333;
}
```

Неоднозначность вызова функции и namespace

```
// Файл A.h
int add(int a, int b) {
    return a + b;
}

//Файл B.h
int add(int a, int b) {
    return a + b;
}

//Файл main
#include "A.h"
#include "B.h"
int main() {
    //неоднозначность
    //int c = add(3,4);
}
```

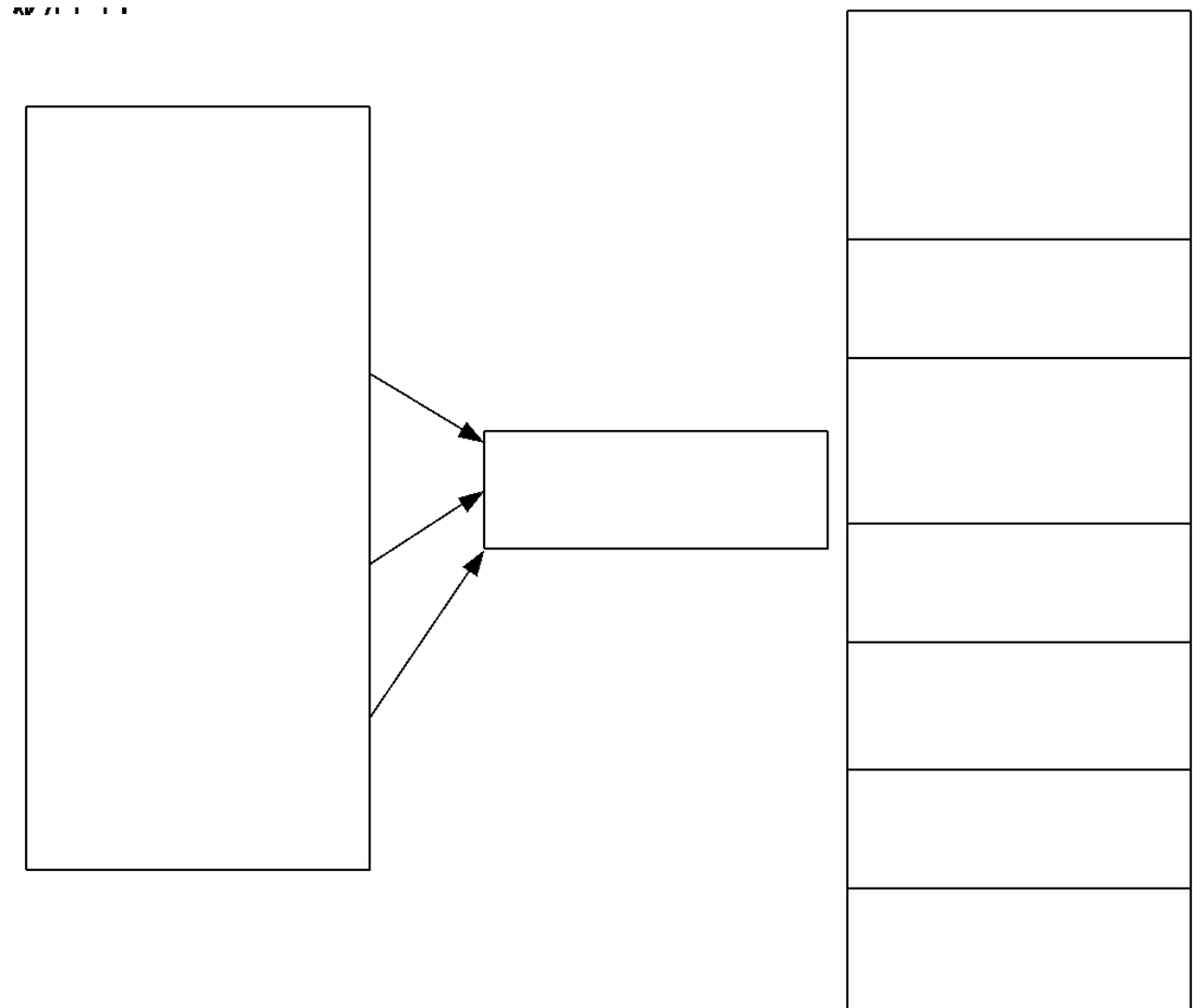
```
// Файл A.h
namespace A{
    int add (int, int) {}
}

//Файл B
namespace B{
    int add (int, int) {}
}

//Файл main
#include "A.h"
#include "B.h"
int main() {
    int b_add = B::add(3,4);
    int a_add = A::add(3,4);
}
```

Встроенные функции

- дополнительные инструкции, необходимые для вызова небольшой функции, могут занять столько же памяти, сколько занимает код самой функции, поэтому экономия памяти может в конечном счете вылиться в дополнительный ее расход,
- в подобных ситуациях можно вставлять повторяющиеся последовательности операторов в те места программы, где это необходимо, однако в этом случае теряются преимущества процедурной структуры программы — четкость и ясность программного кода. Возможно, программа станет работать быстрее и занимать меньше места, но ее код станет неудобным для восприятия.



Встроенные функции

```
inline int sum (int x, int y) {  
    return x + y;  
}
```

```
int main() {  
    int c = sum(2, 4);  
    double d = sum(3, 5) / 10.0;  
}
```

`inline`-функция будет трактоваться как обычная не подставляемая функция, если:

- подставляемая функция является рекурсивной;
- вызов функции размещается до ее определения;
- функция вызывается более одного раза в выражении;
- функция содержит циклы, переключатели и операторы переходов;
- функция, имеющая слишком большой размер, чтобы сделать подстановку.

Перегрузка функций (полиморфизм функций)

Перегрузка функций — возможность использования одноимённых подпрограмм в языках программирования.

```
float sum (float a, float b, float c) {  
    return a + b + c;  
}
```

```
float sum (float a, float b) {  
    return a + b;  
}
```

```
float sum (float a, int b) {  
    return a + b;  
}
```

```
int main() {  
    cout << sum(1.1, 2.2, 3.3) << sum(1.1, 2.2) << sum(1.1, 2);  
}
```

Правила перегрузки

1. Перегруженные функции должны находиться в одной области видимости.
2. Если есть параметры по умолчанию, то их значения в разных функциях должны совпадать.
3. Функции не могут быть перегружены, если они отличаются только типом возвращаемого значения.
4. Функции не могут быть перегружены, если их параметры отличаются **только** модификатором const или использованием ссылки.

```
int foo(int a, int b);  
//int foo(int a, int &b);  
//int foo(const int a, int b);
```

```
int main() {  
    int x, y;  
    foo(x, y);  
}
```

Правила поиска экземпляра перегруженной функции

- поиск точного соответствия параметров;
- преобразование порядковых типов (например, bool и char в int);
- стандартные преобразования типов (int в double, char и short в int, указатели в void*);
- преобразования типа, определенные пользователем и поиск соответствия за счет переменного числа аргументов функции.

```
void foo(void*); //void bad(short);  
void foo(int*); //void bad(long);  
int main() {  
    int x;  
    float y;  
    foo(&x); //int t;  
    foo(&y); //bad(t);  
}
```


Пример неоднозначности при перегрузке

```
void fun (float f) {  
    cout << f << endl;  
}
```

```
void fun (double d) {  
    cout << d << endl;  
}
```

```
int main() {  
    float x = 1.1;  
    double y = 1.1;  
    fun(x);  
    fun(y);  
    fun(10); //неоднозначность  
}
```

Перегрузка операций

Возможна перегрузка всех операций, кроме:

. (точка) .* ?: :: # sizeof

Синтаксис:

```
тип operator операция (список параметров) {  
    тело функции  
}
```

Правила перегрузки операций

- сохраняется количество аргументов, приоритеты и правила ассоциации стандартных типов данных;
- для стандартных типов данных перегружать операции нельзя;
- не может иметь аргументов по умолчанию;

Пример перегрузки операции

```
struct Point{
int x, y;
Point(int x = 0, int y = 0) : x(x), y(y){}
};

Point operator + (Point p, int k){ // p1 + 2
return Point(p.x + k, p.y + k);
}

istream& operator >>(istream& is, Point &p){
is >> p.x >> p.y;
return is;
}

ostream& operator <<(ostream& os, const Point &p){
os << p.x << ' ' << p.y;
return os;
}

//Point p = Point(1,2) + 3;
```

Возврат значения по ссылке

```
float a;
```

```
float& seta() {  
    return a;  
}
```

```
int main() {  
    //ВВОД a  
    seta() = 3.5; //a = 3.5  
    //ВЫВОД a  
}
```

Перегрузка операции индексации

```
enum Coord{X, Y};
struct Point{/*...*/
int& operator [] (char index);//Coord
};

int& Point::operator [] (char index){
if (index == 'x') return x;
else
    if (index == 'y') return y;
    else throw std::out_of_range("error");
}

int main(){
Point a, b;
cin >> a;
b = a + 2;
cout << b << endl;
b['x']++; //b.x++;//b[X]
cout << b;
}
```

Шаблоны функций

```
template<class T>  
    заголовок  
    {тело функции}
```

```
template <class T> //можно исп. typename  
T abs (T n) {  
    return (n < 0) ? -n : n;  
}
```

```
int main() {  
    int i; abs(i);  
    float f; abs(f);  
    double d; abs(d);  
}
```

- * объявление и реализация шаблонной функции должны быть описаны в одном файле
- ** шаблонная функция будет работать для любого типа в котором определены операции и методы, используемые в теле шаблонной функции

Пример шаблонной функции

```
template <class T>
int find(T* arr, T value, int size){
    for(int j = 0; j < size; j++)
        if(arr[j] == value)
            return j;
    return -1;
}
```

```
int main(){
    char chArr[] = { 1, 3, 5, 9, 11, 13 };
    char ch = 5;
    long loArr[] = { 1L, 3L, 5L, 9L, 11L, 13L };
    long lo = 11L;
    std::cout << find(chArr, ch, 6) << std::endl;
    std::cout << find(loArr, lo, 6) << std::endl;
    //для шаблонных функций не происходит неявного приведения типов
    //std::cout << find(loArr, ch, 6) << std::endl;
```


Аргументы шаблона

```
template <class T1, class T2>
    T1 f (T2) {
        /*тело функции*/
    }
```

...

```
//автоматическое определение типа аргументов шаблона
//int i = f ('s');
```

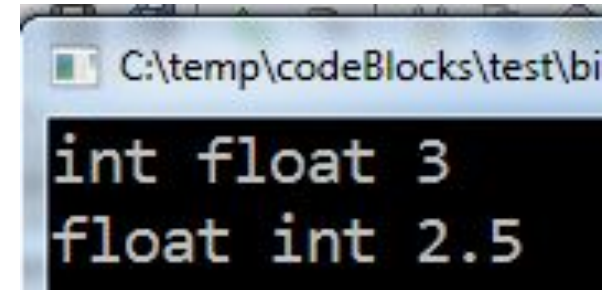
```
//явное задание типа аргументов шаблона
int i = f <int, char>('s');
```

Пример шаблонной функции с несколькими аргументами

```
template <class T1, class T2>
T1 fu(T2 arg) {
    T1 s = arg / 10.0;
    return s;
}
```

```
int main() {
    int i = 25;
    float f = 35;
    std::cout << "int float " << fu<int, float>(f)
               << std::endl
               << "float int " << fu<float, int>(i);

}
```



C:\temp\codeBlocks\test\bi

```
int float 3
float int 2.5
```

*обязательно явное задание типов аргументов при вызове данной функции, т.к. компилятор не видит возвращающий тип

Директивы препроцессора

#include <> #include ""	включение
#if константное выражение	условная компиляция,
[#elif константное выражение]	например,
[#else константное выражение]	#if defined(AAA), #if !defined(AAA), *
#endif	где AAA – константа #define
#ifdef, #ifndef	аналог #if defined(AAA), #if !defined(AAA)
#define, #undef	определить, снять определение
#pragma	
#error	остановка компиляции
#line	изменение __LINE__ и __FILE__
#	помещает аргумент, перед которым он стоит, в ""
##	конкатенация

*defined - оператор времени компиляции

#error

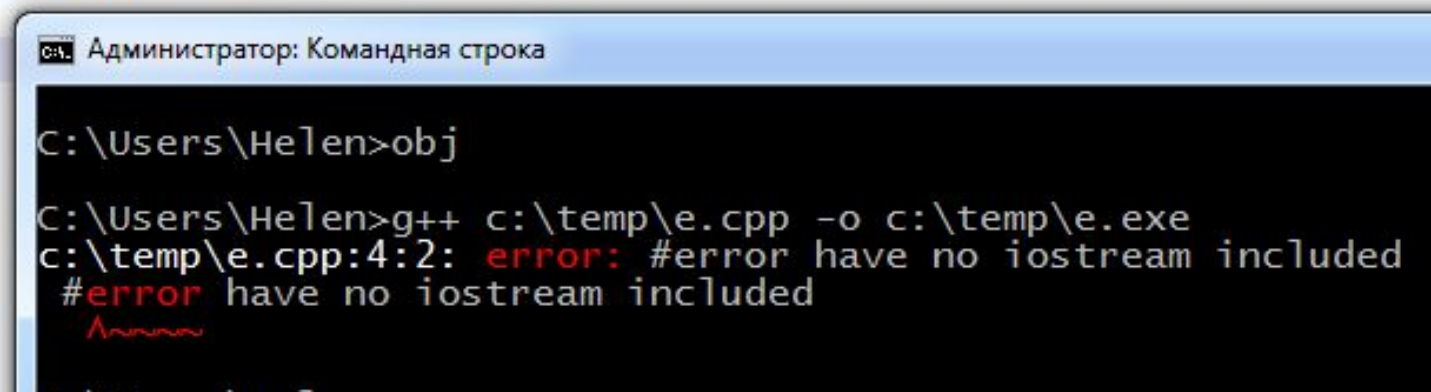
- указывает компилятору в случае ее обнаружения остановить компиляцию. Как правило, она используется для отладки.

#error сообщение_об_ошибке

- компилятор выводит сообщение в следующем виде и завершает компиляцию:

имя_файла номер_строки: Error directive: сообщение_об_ошибке

```
1  #include <iostream>
2
3  #ifndef IOSTREAM
4  #error have no iostream included
5  #endif
6
7  #define IOSTREAM
8
9  using namespace std;
10
11  int main(){
12
13  }
```



```
Администратор: Командная строка
C:\Users\Helen>obj
C:\Users\Helen>g++ c:\temp\e.cpp -o c:\temp\e.exe
c:\temp\e.cpp:4:2: error: #error have no iostream included
#error have no iostream included
~~~~~
```

и

```
#define MakeSInQuots(s) # s
```

Препроцессор превратит строку

```
std::cout << MakeSInQuots(I like C++);
```

в строку

```
std::cout << "I like C++";
```

```
#define CONCAT(a, b) a ## b
```

Препроцессор преобразует

```
std::cout << CONCAT(x, y);
```

в

```
std::cout << xy;
```

#line

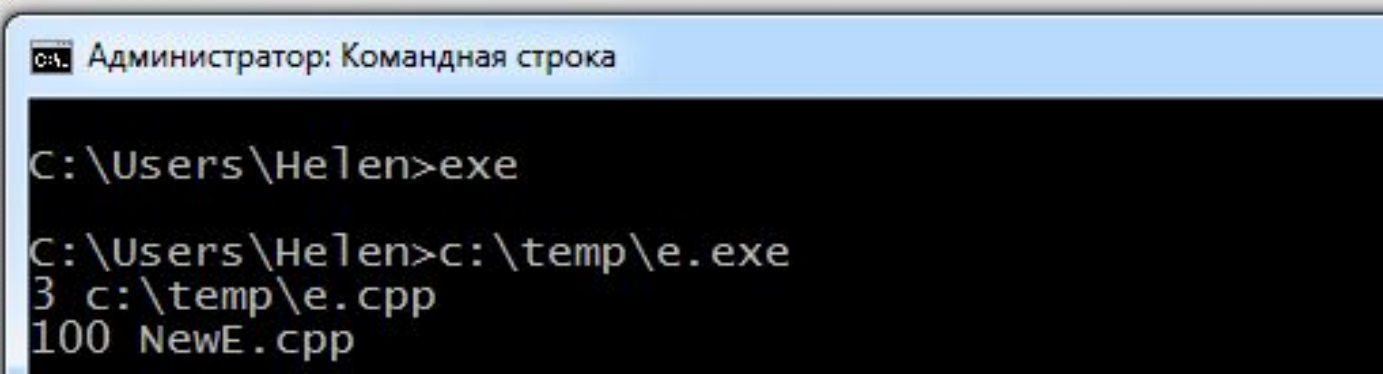
#line число "имя_файла"

позволяет изменить predetermined макросы:

- `__LINE__` номер компилируемой строки,
- `__FILE__` имя компилируемого файла.

число - это любое положительное число, а необязательный параметр *имя_файла* является любым допустимым файловым идентификатором.

```
1  #include <iostream>
2  int main(){
3      std::cout << __LINE__ << ' ' << __FILE__ << '\n';
4      #line 100 "NewE.cpp"
5      std::cout << __LINE__ << ' ' << __FILE__ << '\n';
6  }
```



```
C:\Users\He1en>exe
C:\Users\He1en>c:\temp\e.exe
3 c:\temp\e.cpp
100 NewE.cpp
```

Предопределенные макросы

`__LINE__`, `__FILE__`

`__DATE__` содержит строку в виде месяц/день/год, соответствующую дате перевода исходного кода в объектный.

`__TIME__` содержит время, когда был откомпилирован исходный код, строка в формате *часы: минуты: секунды*.

`__STDC__` если макрос определен, значит, программа была откомпилирована с использованием стандарта ANSI C со включенной проверкой на совместимость. В противном случае макрос не определен.

```
1  #include <iostream>
2  int main(){
3      std::cout << __DATE__ << ' ' << __TIME__ << '\n';
4      #if defined(__STDC__)
5          std::cout << "C OK!";
6      #endif
7  }
```

```
C:\Users\Helen>exe
```

```
C:\Users\Helen>c:\temp\e.exe
```

```
Apr 28 2021 16:01:38
```

```
C OK!
```

```
C:\Users\Helen>
```

#define

```
#define A B
```

```
namespace A{  
int x = 2;  
}
```

```
int main() {  
std::cout << A::x; //2  
std::cout << B::x; //2  
}
```


Макросы пользователя

`#define имя_макроса последовательность_символов`

`#define abs(n) ((n < 0) ? (-n) : (n))`

Недостатки макросов:

- не проводится проверка соответствия типов;
- тело состоит только из одного выражения.

Еще примеры макросов

```
#define PROD(x, y)    ((x) * (y))
#define SQR(x)         x * x
#define SQR1(x)        (x) * (x)
#define SQR2(x)        (x) * (x) + 1
#define SQR3(x)        ((x) * (x) + 1)
```

```
int main() {
int a = 2, b = 3;
cout << PROD(a, b)          << endl // 2*3
      << SQR(a)              << endl // 2*2
      << SQR(a + 1)          << endl // 2+1*2+1
      << SQR1(a + 1)          << endl // (2+1)*(2+1)
      << SQR2(a + 1) * 2      << endl // (2+1)*(2+1)+1*2
      << SQR3(a + 1) * 2      << endl; // ((2+1)*(2+1)+1)*2
}
```

Пример сложного макроса

```
#define MAX(x, y, r) { do {int maCRoMAX_X = (x);  
    int maCRoMAX_Y = (y);  
    (r) = ( (maCRoMAX_X) > (maCRoMAX_Y) )  
        ?  
        (maCRoMAX_X)  
        :  
        (maCRoMAX_Y); }  
    while(0);  
    }//должен быть записан в одну строку  
  
int main() {  
    int a (10), b (20), c (0);  
    MAX(a, b, c);  
    std::cout << a << ' ' << b << ' ' << c << std::endl;  
    MAX(a += b, b, c);  
    std::cout << a << ' ' << b << ' ' << c << std::endl;  
    MAX(a - b, b--, c);  
    std::cout << a << ' ' << b << ' ' << c << std::endl;  
}
```

Ключи компиляции

- -o <name> - имя выходного файла, если компилятору не указать имя выходного файла то по умолчанию именем будет a.out

Пример

```
g++ -o e.exe e.cpp
```

Пример сборки с несколькими исходными файлами:

```
g++ -o e.exe e1.cpp e2.cpp
```

- -c - создание объектного файла

Пример

```
g++ -c -o e.o e.cpp
```

Пример сборки с несколькими объектными файлами:

```
g++ -o e.exe e1.o e2.o
```

*подробнее `g++ --help`

Код программы после работы препроцессора

- **ключ -E** позволяет посмотреть код после его обработки препроцессором
- g++ -E имя_файла

Например:

```
g++ -E c:\temp\e.cpp -o c:\temp\preE.cpp
```

e.cpp

```
#define MAX(x, y, r) do {int maCRoMAX_X = (x); int maCRoMAX_Y = (y); (r) = ( (maCRoMAX_X)
int main(){
int a (10), b (20), c (0);
MAX(a += b, b, c);
}
```

preE.cpp

```
# 1 "c:\\temp\\e.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "c:\\temp\\e.cpp"

int main(){
int a (10), b (20), c (0);
do {int maCRoMAX_X = (a += b); int maCRoMAX_Y = (b); (c) = ( (maCRoMAX_X) > (maCRoMAX_Y) )
}
```

#include

- sum.h

```
int sum(int, int);
```

- sum.cpp

```
int sum(int a, int b) {  
    return a + b;  
}
```

- e.cpp

```
#include "sum.h"  
  
int main() {  
    int c = sum(2, 3);  
}
```

```
C:\Users\Helen>g++ -E c:\temp\e.cpp  
# 1 "c:\\temp\\e.cpp"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "c:\\temp\\e.cpp"  
# 1 "c:\\temp\\sum.h" 1  
int sum(int, int);  
# 2 "c:\\temp\\e.cpp" 2  
  
int main(){  
    int c = sum(2, 3);  
}
```

Неоднозначность вызова из-за повторного включения заголовка. Применение директив препроцессора

```
// Файл A.h  
int sum(int a, int b) {  
    return a + b;  
}
```

```
//Файл B.h  
#include "A.h"
```

```
//Файл main  
#include "A.h"  
#include "B.h"  
int main() {  
    //неоднозначность  
    int c = sum(3, 4);  
}
```

Устранение неоднозначности вызова функции при повторном включении заголовка

```
#pragma once
```

или

```
#ifndef XXX  
#define XXX  
какой-то код  
#endif
```

```
// Файл A.h  
#ifndef AH  
#define AH
```


#pragma

- самостоятельное изучение
- <https://en.cppreference.com/w/cpp/preprocessor/impl>

Линковка

- при линковке (работа компоновщика) происходит подстановка адресов функций в места их вызова
- имена функций преобразуются (mangle) кодируя параметры, например у функции:

```
void foo(int, double)
```

mangle-имя:

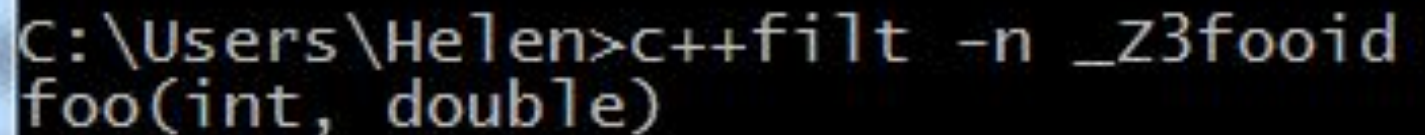
```
_Z3fooid
```

- возвращаемый тип функции в кодировании не участвует
- узнать кодированное имя можно при ассемблировании - компиляция с **ключом -S**

```
g++ -S c:\temp\e.cpp -o c:\temp\asmE.cpp
```

- узнать сигнатуру функции по mangle-имени (последнее - имя):

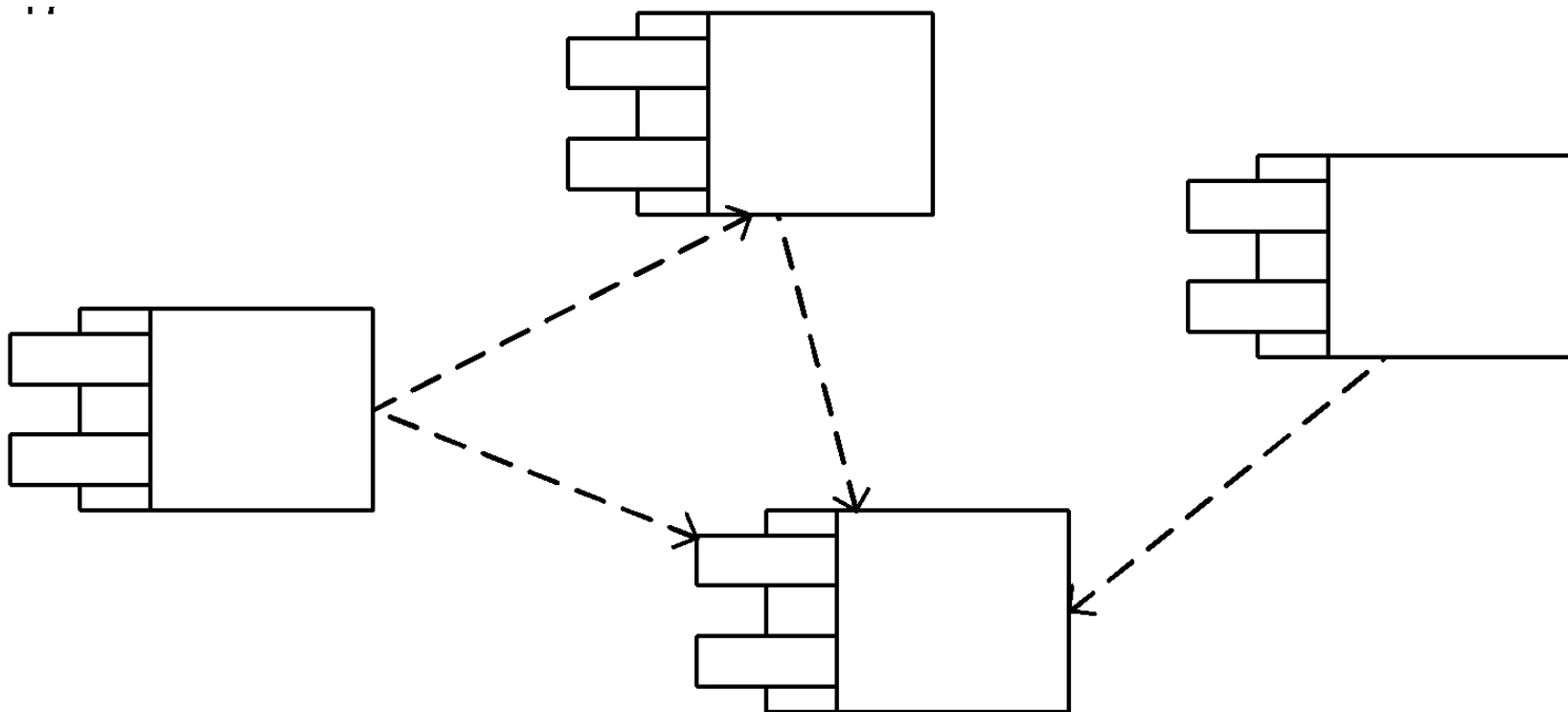
```
c++filt -n _Z3fooid
```



```
C:\Users\Helen>c++filt -n _Z3fooid  
foo(int, double)
```

Диаграмма компонентов (UML)

Диаграмма компонент дает представление о том, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой



Заключение

1. Модульность повышает уровень абстракции программы.
2. Для использования модуля достаточно знать его интерфейс, а не реализацию.
3. Скрытие деталей реализации называется инкапсуляцией.
4. Интерфейсом модуля являются заголовки всех функций и описания доступных извне типов, переменных и констант.
5. Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции.

Приложение А

Инструкция по созданию статической библиотеки

- создать новый проект с шаблоном "статическая библиотека;
- добавить в проект сrr модуль и заголовочный файл модуля;
- реализовать функции модуля, описать их прототипы в заголовочном файле модуля.
- скомпилировать библиотеку (собрать проект);
- для использования .lib в другом проекте необходимо включить ее в ресурсы текущего проекта, в тексте программы текущего проекта подключить заголовочный файл.

Приложение Б

Удобное объявление указателей

```
int *p1, *p2, *p3;
```

или

```
typedef int * ptrInt; // новое имя для указателя int*
```

```
ptrInt p1, p2, p3; // 3 указателя
```

```
ptrInt a[10]; // массив из 10 указателей
```

Перенаправление файлов

Большинство ОС поддерживает перенаправление файлов, позволяющее ассоциировать именованный файл со стандартным устройством ввода и стандартным устройством вывода

\$addItems <infile> outfile – читает транзакции из файла infile и записывает ее вывод в файл outfile в текущем каталоге (addItems.exe – откомпилированная программа, просто addItems в UNIX-системах)

addItems > res.log - запишет выходные данные программы addItems в файл res.log

или

addItems >> res.log допишет в конец файла res.log выходные данные программы addItems

* подробнее – самостоятельное изучение

Еще раз о const_cast

```
const int a = 1;  
//int *pa = &a; // invalid casting  
int *pa = const_cast <int*> (&a);  
*pa = 2;  
cout << a << " " << *pa; // 1 2
```