

Компиляция программы. Препроцессор.

Основные этапы компиляции программы (`hello.c` → `a.out`)

- **Разбиение на лексемы:** исходный файл разбивается на «слова» (`tokens`) и пробелы
- **Выполнение директив препроцессора:** присоединяются заголовочные файлы, разворачиваются макросы
- **Трансляция:** производится синтаксический и семантический анализ, текст преобразуется в объектный файл
- **Компоновка** (`link`): один или несколько объектных файлов объединяются, добавляются функции внешних библиотек, создается выполняемый файл

Вызов препроцессора (для отладки)

```
gcc -E hello.c
```

Директивы препроцессора

Список директив

```
#include   #define   #undef    #if      #ifdef   #ifndef  
#else     #elif     #endif    #line    #error   #pragma  
#warning  #embed   #elifdef  #elifndef (C23, C++17...23)
```

Основные правила

- Каждая директива должна находиться на отдельной строке
первый символ команды `#`
- Если директива «не помещается» в одной строке, её можно
продолжить поставив в конец строки обратную косую черту `" \ "`

Вставка файла

Директива `#include "file"`

- Вставляет `file` на место строки с `#include`
- В `file` может содержаться вложенный `#include` и так далее
- Имя файла должно быть заключено либо в двойные кавычки `"file"` или в угловые скобки `<file>`

gcc -E hello.c → 841 строка

☞ угловые скобки принято использовать для стандартных заголовочных файлов, а кавычки для файлов относящихся к конкретной программе:
“если файл был указан в двойных кавычках и поиск закончился неудачно, то эта директива переобрабатывается как если бы файл был заключен в угловые скобки”

Проверка доступности «заголовочного» файла

`__has_include("filename")`

C23, C++17

- Препроцессорный оператор позволяющий проверить, что `filename` доступен для использования в директиве `#include`

```
#if __has_include("fun_prt.cc")
#include "fun_prt.cc"
#else
#define EMPTY_FUN_PRT
void fun_prt() { printf("Empty function\n"); }
#endif
int main() {
    fun_prt();
#endifif EMPTY_FUN_PRT
FILE* f = fopen("fun_prt.cc","w");
fprintf(f,"void fun_prt() { printf(\"Good function\\n\"); }\\n");
fclose(f);
#endif
}
```

Макроопределения (макросы)

Директива `#define`

```
#define ИМЯ_МАКРОСА последовательность_символов
```

- Осуществляется **текстовая подстановка**:
ИМЯ_МАКРОСА → последовательность_символов
- Признаком конца директивы является конец строки
- Если ИМЯ_МАКРОСА внутри кавычек, то замены не будет

Именованные константы

```
#define MAX_SIZE 100
#define MIN_SIZE MAX_SIZE / 2
#define LONG_STRING "this is a very long " \
                  "string used as an example\n"
#define TEST this is test
```

```
double array[MAX_SIZE];
for(i = MIN_SIZE; i < MAX_SIZE; i++) ...
printf(LONG_STRING);
printf("TEST\n");
```

«Output: gcc -E »

```
double array[100];
for(i = 100 / 2; i < 100; i++) ...
printf("this is a very long " "string used as an example\n");
printf("TEST\n");
```

Часто используемые предопределенные макросы

- `__DATE__` и `__TIME__` макросы содержащие дату и время вызова препроцессора (С-стринги из 11 символов)
- `__FILE__` имя текущего обрабатываемого файла (С-стринг)
- `__LINE__` номер строки в этом файле (целое число)
- `__FUNCTION__` имя текущей функции (С-стринг) (~~obsolete~~)

Вместо `__FUNCTION__` в стандарте языка (C99,C++) имеется

- `__func__` – переменная содержащая имя выполняемой функции:
`static const char __func__[] = "function name";`

Пример отладочной печати

```
printf(" Error ... at %s, line %d.\n", __FILE__, __LINE__);
printf("I am in %s function\n", __func__);
```

Макроподстановки: макросы с параметрами

Макросы с формальными параметрами

```
#define ABS(x)    ((x) < 0 ? -(x) : (x))
#define MIN(x,y)  (((x)<(y)) ? (x) : (y))
```

Пример

```
result1 = ABS(a);
result2 = MIN(a,b);
```

«Output: gcc -E »

```
result1 = ((a) < 0 ? -(a) : (a));
result2 = (((a)<(b)) ? (a) : (b));
```

Правила написания

- ☞ Нельзя ставить пробел между именем макроса и скобками
- ☞ Лишняя скобка не повредит!

Плохая идея

```
#define SQR(x)      x*x    // ERROR  
result = SQR(a+b);
```

«Output: gcc -E »

```
result = a+b*a+b;
```

Плохая идея

```
#define SQR(x)      x*x    // ERROR  
result = SQR(a+b);
```

«Output: gcc -E »

```
result = a+b*a+b;
```

Плохо продуманная идея

```
#define SQR(x)      (x)*(x)  // ERROR  
result = 1./SQR(a);
```

«Output: gcc -E »

```
result = 1./(a)*(a);
```

Плохая идея

```
#define SQR(x)      x*x    // ERROR  
result = SQR(a+b);
```

«Output: gcc -E »

```
result = a+b*a+b;
```

Плохо продуманная идея

```
#define SQR(x)      (x)*(x)  // ERROR  
result = 1./SQR(a);
```

«Output: gcc -E »

```
result = 1./(a)*(a);
```

Правильный вариант

```
#define SQR(x)      ((x)*(x))    // OK!
```

☞ Макросы не функции!

тестируем макрос

```
#define MIN(x,y) (((x)<(y)) ? (x) : (y))
```

```
a = 1;  
b = 10;  
result = MIN(a++,b);  
printf(" a= %i, b= %i, result= %i\n",a,b,result);  
Result> a= 3, b= 10, result= 2
```

«Output: gcc -E »

```
result = (((a++)<(b)) ? (a++) : (b));
```

Директива #undef имя_макроса

- «удаляет» ранее определенный макрос

Используйте inline functions вместо «функций-макросов»

```
#undef MIN // удаляем макрос MIN

inline int MIN(int x,int y) { // добавляем функцию MIN()
    return (x < y) ? x : y;
}

a = 1;
b = 10;
result = MIN(a++,b); // this is now a function
printf(" a= %i, b= %i, result= %i\n",a,b,result);
Result> a= 2, b= 10, result= 1
```

Условная компиляция: if - else

Директивы #if, #else, #elif и #endif

```
#if константное выражение 1
    «включается» если выражение 1 истинно
#elif константное выражение 2
    иначе, если выражение 2 истинно
#else
    ...
#endif
```

☞ Промежуточные #else и #elif необязательны

```
#if MAX_SIZE>99
    printf("If size of the array is 100 and more\n");
#else
    printf("The case of a small array.\n");
#endif
```

Директивы #ifdef и #ifndef

```
#ifdef ИМЯ_МАКРОСА // идентично #if defined(ИМЯ_МАКРОСА)
    «включается» если ИМЯ_МАКРОСА определено
#endif
#ifndef ИМЯ_МАКРОСА // идентично #if !defined(ИМЯ_МАКРОСА)
    «включается» если ИМЯ_МАКРОСА не определено
#endif
```

- #elifdef ИМЯ_МАКРОСА ≡ #elif defined ИМЯ_МАКРОСА (C23,C++23)
- #elifndef ИМЯ_МАКРОСА ≡ #elif !defined ИМЯ_МАКРОСА (C23,C++23)

```
#ifdef DEBUG
    printf(" x=..."); // отладочная печать, если определен DEBUG
#endif
```

```
#ifdef ABRAKADABRA
    что-то, что мы хотим временно закомментировать
#endif
```

#ifndef в заголовочных файлах

В заголовочных файлах рекомендуется использовать конструкцию:

```
/* example.h */  
#ifndef EXAMPLE_H  
#define EXAMPLE_H  
  
...  
#endif
```

- ✓ Имя макроса проверяется и если оно не определено тут же определяется
- ✓ Имя макроса должно быть уникально для каждого файла, обычно конструируется из имени заголовочного файла

☞ Многократная вставка такого заголовочного файла безопасна

```
/* example.c */  
#include "example.h"  
#include "example.h"      // безопасно
```

Операторы # и

Используются внутри макросов

- # – создает С-стринг из аргумента перед которым стоит
- ## – объединяет (склеивает) две лексемы

Пример с

```
#define showtype(t) printf("sizeof(%s) = %zu\n",#t,sizeof(t))
showtype(long double); // sizeof(long double) = 16
```

Пример с

```
#define INT3(x) int x##1, x##2, x##3
INT3(a); // => int a1, a2, a3;
a1 = 1; a2 = a1++; a3 = a2++;
printf("a1= %i, a2= %i, a3= %i\n",a1,a2,a3); // a1= 2, a2= 2, a3= 1
```

Type generic selection in C11

Новый «препроцессорный» оператор _Generic

- Позволяет во время компиляции сделать выбор на основе **типа**
 - Синтаксис похож на **switch**, проверяется **тип** переменной или выражения, а далее идут пары – имя типа (или **default**) : результат
- ☞ Отсутствует в C++

Пример

```
#define SIN(X)  _Generic( (X), \
                      double: sin(X), \
                      long double: sinl(X), \
                      float:   sinf(X) \
)
...
float y = 2.;
auto sy = SIN(y);
```

Введение в абстрактные типы данных

Структура данных в программировании

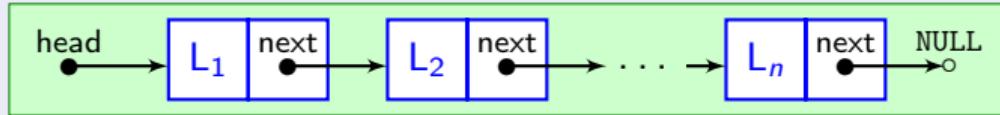
- Структура данных (**data structure**): специальный способ хранения данных удобный для построения эффективных алгоритмов
- Абстрактные типы данных: математические модели структур данных

Некоторые абстрактные типы данных

- Список ([List](#))
- Стек ([Stack](#))
- Очередь ([Queue](#))
- Дерево ([Tree](#))

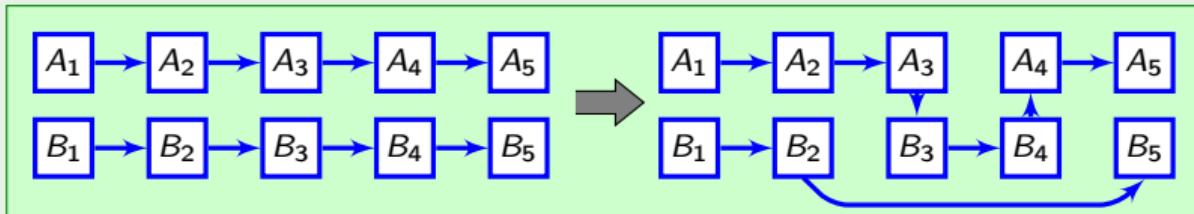
Список (List)

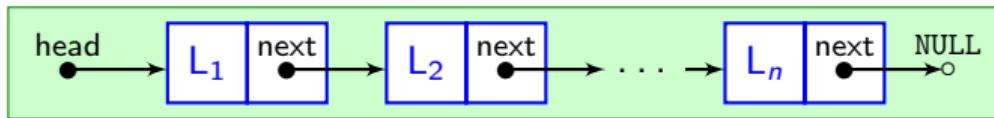
Односвязный список (Singly linked list)



- Последовательность узлов (`nodes`) в которых содержится:
 - ✓ L_i – сколь угодно большие, сложные данные
 - ✓ `next` – указатель на следующий узел, последний в списке = `NULL`
- Указатель на самый первый узел списка: `head`

☞ основное свойство: легко добавлять и удалять элементы списка





Возможная реализации списка

```
struct node_ { // one node == list consisting of one node
    double v;
    struct node_ * next;
};

typedef struct node_ Node;
```

Функциональная спецификация: ожидаемый набор функций

- ① Создание нового узла → список из одного узла
- ② Вставка «другого» списка → в начало, в конец, в заданное место
- ③ Удаление одного узла → из начала, из конца, из заданного места
- ④ Подсчет числа узлов в списке: имеет сложность $O(n)$
- ⑤ Разбиение списка на два в заданном месте

```
Node* CreateNode(double value) { // (1)
    Node* nn = (Node*) malloc(sizeof(Node)); // new node
    nn->v = value;
    nn->next = NULL;
    return nn;
}

void InsertAtHead(Node** head, Node* nn) { // (2a)
    Node* tmp = nn; // search for end node of 'nn'-list
    while ( tmp->next != NULL ) { tmp = tmp->next; }
    tmp->next = *head;
    *head = nn;
}

void AppendToEnd(Node** head, Node* nn) { // (2b)
    if ( *head == NULL ) {
        *head = nn;
    } else {
        Node* tmp = *head; // search for end node
        while ( tmp->next != NULL ) { tmp = tmp->next; }
        tmp->next = nn; // append to the end
    }
}
```

```
void DeleteFromHead(Node** head) { // (3a)
    if ( *head == NULL ) {
        printf("\n%s ERROR: List is empty\n",__func__);
    } else {
        Node* tmp = *head;
        *head = tmp->next;
        free(tmp);
    }
}

size_t PrintList(Node* head) { // (4) return size of List
    size_t n = 0;
    while(head) {
        printf("%.1f -> ",head->v);
        n++;
        head = head->next;
    }
    printf("null; [size= %zu]\n",n);
    return n;
}
```

Test of singly linked list

```
Node* head = NULL; // first list
for ( int i = 0; i <= 2; ++i )
    InsertAtHead(&head, CreateNode((double)i));
PrintList(head); // 2.0 -> 1.0 -> 0.0 -> null; [size= 3]

Node* list2 = NULL; // second list
for ( int i = 1; i <= 2; ++i )
    AppendToEnd(&list2, CreateNode(-10.*i));
PrintList(list2); // -10.0 -> -20.0 -> null; [size= 2]

AppendToEnd(&head, list2);
PrintList(head); // 2.0 -> 1.0 -> 0.0 -> -10.0 -> -20.0 -> null;

DeleteFromHead(&head);
DeleteFromHead(&head);
PrintList(head); // 0.0 -> -10.0 -> -20.0 -> null; [size= 3]
```

Другие разновидности списков

Двусвязный список (Doubly linked list)

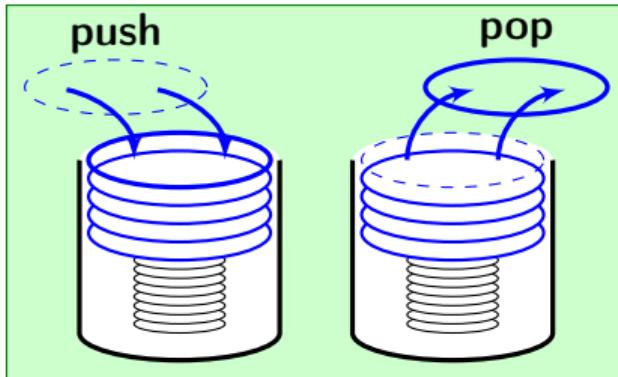


- + в каждом узле имеется ссылка на предыдущий узел: `prev`
- + возможен указатель на последний узел списка: `tail`

Кольцевой список (Circularly linked list)

- + двусвязный список, где `next` в последнем узле списка указывает на первый узел, а `prev` в первом узле указывает на последний узел
- ☞ список (`list`) в Python это массив указателей на PyObjects и не имеет отношения к рассматриваемой структуре

Стек (Stack)



Принцип организации

- Последним пришел, первым вышел (last-in, first-out: LIFO)
- Доступ, извлечение только последнего элемента

Функциональная спецификация

- ① Создание стека → функция создающая пустой стек
- ② Проверка является ли стек пустым
- ③ Добавление элемента → сохранение элемента в стеке: **push**
- ④ Извлечение верхнего элемента: **pop**

● на основе списка

```
struct stack_sl {  
    Node* head; // singly linked list  
};  
typedef struct stack_sl Stack;
```

● на основе массива

```
struct stack {  
    double* v;  
    int top;  
};
```

```
void init(Stack* S) { S->head = nullptr; }                                // (1)  
bool is_empty(Stack* S) { return (S->head == nullptr); }                  // (2)  
void push(Stack *S, double val) {                                            // (3)  
    InsertAtHead(&S->head, CreateNode(val));  
}  
double pop(Stack* S) {                                                       // (4)  
    if ( is_empty(S) ) {  
        printf("\n%s ERROR: Stack is empty\n", __func__);  
        exit(1);  
    }  
    double val = S->head->v;  
    DeleteFromHead(&S->head);  
    return val;  
}
```

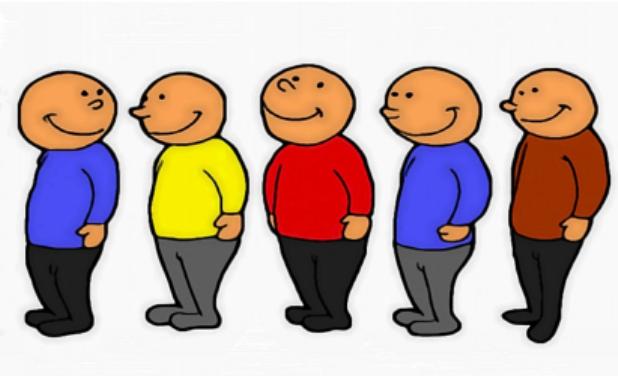
Test of Stack

```
Stack S;
init(&S);                      // initialize

for(int i = 1; i < 5; ++i) {
    push(&S, 1.1*i);           // push a few elements
}

while( !is_empty(&S) ) {
    printf("%g ", pop(&S));   // use return value in printf
}
printf("\n");                   // 4.4 3.3 2.2 1.1
```

Очередь (Queue)



Принцип организации

- Первым пришел – первым вышел (first-in, first-out: FIFO)
- Доступ, извлечение только первого элемента

Функциональная спецификация

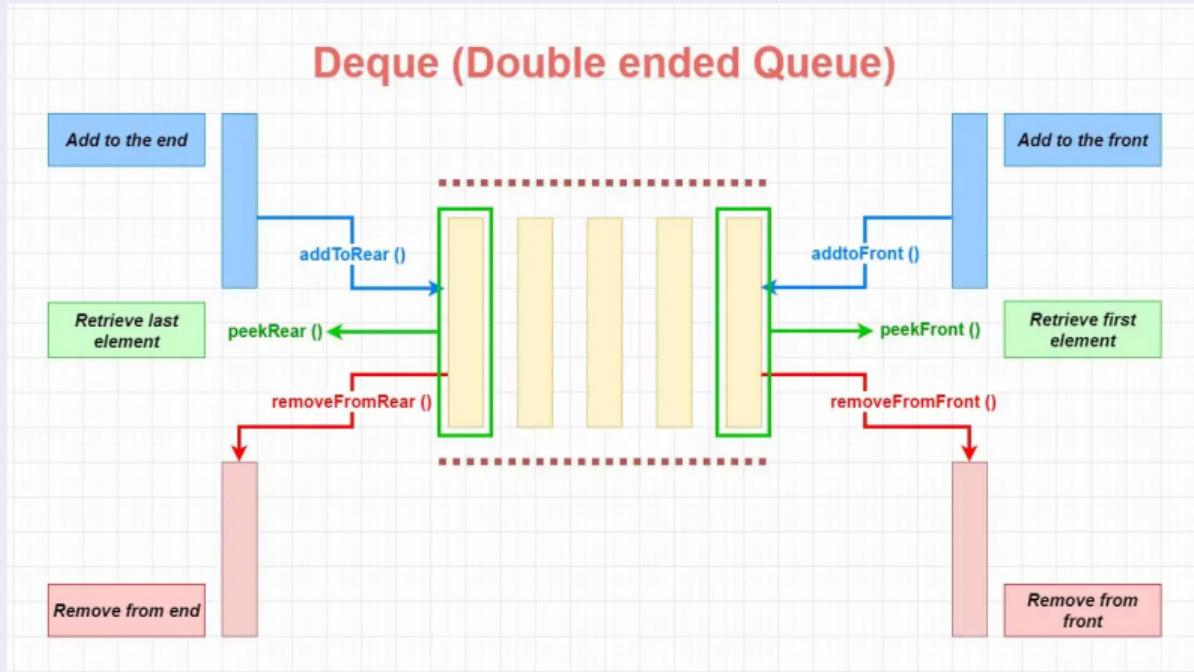
- ① Создание → функция создающая пустую очередь
- ② Проверка является ли очередь пустой
- ③ Добавление элемента в конец очереди
- ④ Извлечение первого элемента из очереди

Очередь на основе односвязного списка

```
struct queue_sl {  
    Node* head; // singly linked list  
};  
typedef struct queue_sl Queue;  
// Все функции как в Stack за исключением (3)  
void push(Queue *Q, double val) {  
    AppendToEnd(&Q->head, CreateNode(val));  
}
```

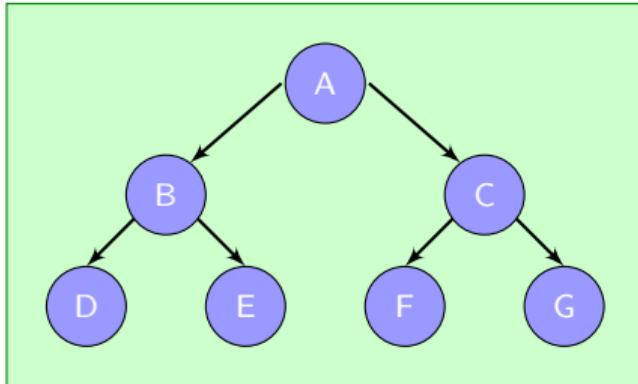
```
Queue Q;                      // Queue for testing  
init(&Q);                    // initialize  
for(int i = 1; i < 5; ++i) {  
    push(&Q, 1.1*i);          // push a few elements  
}  
while( !is_empty(&Q) ) {  
    printf("%g ", pop(&Q)); // use return value in printf  
}  
printf("\n");                  // 1.1 2.2 3.3 4.4
```

Дек (Double-Ended Queue): двухсторонняя очередь



☞ Обобщает концепции стека и очереди: элементы можно добавлять, удалять и читать как в начало, так и в конец DEQ

Дерево (Tree)



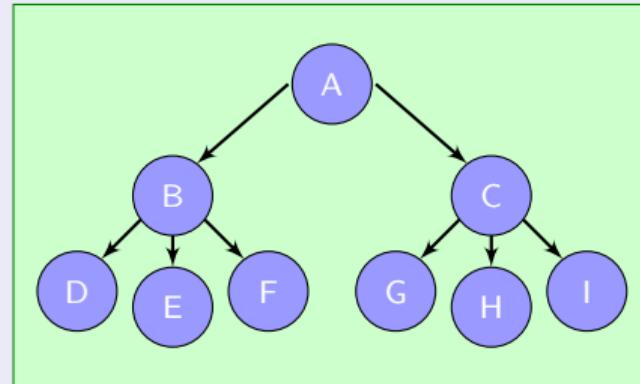
Основное свойство

👉 Дерево – иерархический набор узлов (**node**), связанных ребрами (**edge**), организованный так, что имеется единственный путь (**path**) от одного узла к другому

Терминология

- A – корень (**root node**)
- Соединение двух узлов – ребро (**edge**); для N узлов всего $N - 1$ ребро
- Путь – последовательность узлов и ребер между узлами: $\text{path}(AD) = A-B-D$, длина пути – полное число узлов в пути (**3**)
- Глубина дерева (**Depth of Tree**) – самый длинный путь из корня

Дерево общего вида



Терминология

- A,B,C – родительские (*parent*) узлы; B и C – родственники (*siblings*) и дети (*childes*) узла A
- Бинарное дерево – число детей любого узла не больше двух
- узлы D,E,F,G,H,I без детей – листья (*leaf*) или терминальные (*terminal*) узлы
- Каждый дочерний узел формирует поддерево (*sub-tree*)

① возможная реализация на C++

```
// like sl-list where node* next is replaced by vector<node*> children
struct node {
    node(int p = 0) : pdg_code(p){}; // constructor
    int pdg_code = 0;
    std::vector<std::unique_ptr<node>> children = {};
};
```

«Обход» дерева – печать элементов

```
void print_tree(const node& A, int lv = 0) {
    std::cout << A.pdg_code << '(';
    for ( size_t i = 0; i < size(A.children); ++i ) {
        print_tree(*A.children[i],lv+1); // recursion
    }
    std::cout << ')' << (lv > 0 ? ',' : '\n');
}
```

Пример: дерево распадов $\eta \rightarrow \pi^+ \pi^- \pi^0$; $\pi^0 \rightarrow \gamma\gamma$

```
auto eta = std::make_unique<node>(221); // root: eta = 221
eta->children.emplace_back(std::make_unique<node>(211)); // pi+ = 211
eta->children.emplace_back(std::make_unique<node>(-221)); // pi- = -221
eta->children.emplace_back(std::make_unique<node>(111)); // pi0 = 111

auto& pi0 = eta->children[2];
pi0->children.emplace_back(std::make_unique<node>(22)); // gamma = 22
pi0->children.emplace_back(std::make_unique<node>(22)); // gamma = 22

print_tree(*eta); // 221(211() -221() 111(22() 22() ) )
```

Дополнительные слайды

Атрибуты, начиная с C++11 и C23

- ☞ Стандартизованный способ передать компилятору дополнительную информацию используя синтаксис: `[[attribute-list]]`

некоторые стандартные атрибуты: [список для C++, и C23](#)

```
[[maybe_unused]] int debug_fun() {...}; // C++17 attribute
[[maybe_unused]] int i = f();
assert(i>0); // may be discarded by compiler in final version
[[nodiscard]] int fun() {return 1;} // C++17 attribute
fun(); // Error or warn by compiler
auto i = fun() // OK!
```

- Кроме стандартных атрибутов, имеются нестандартные:

```
[[gnu::const]] void f(); // gcc specific
[[clang::always_inline]] void f(); // clang specific
```

- «Неизвестные» атрибуты не вызывают ошибок при компиляции ([C23,C++17](#))

Проверка (assertion) при компиляции

`static_assert(int_expr, message)`

since C11, keyword in C23

- Определено в `<cassert.h>`: проверка «логического» выражения **во время компиляции** и если выражение равно нулю (ложно) остановка компиляции с выводом сообщения `message`
- ☞ в C++11 имеется похожая декларация:

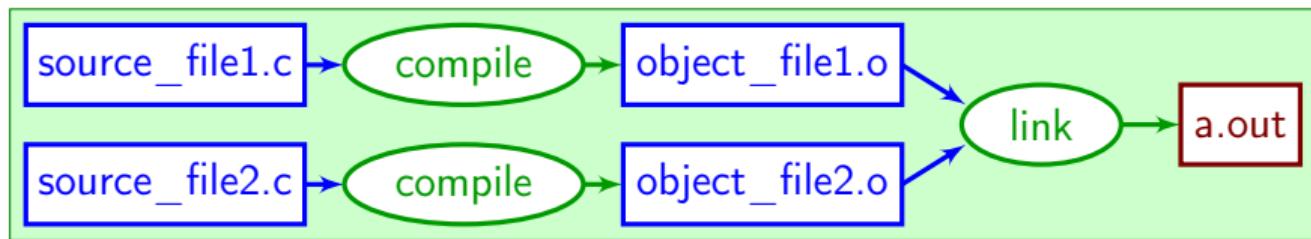
`static_assert(bool_constexpr, message)`

Пример: проверка, что `long` занимает 8 байт

```
#include <cassert.h>
static_assert(sizeof(long) == 8, "long int must be exactly 8 bytes");
// возможная ошибка во время компиляции:
// error: static assertion failed: "long int must be exactly 8 bytes"
```

Раздельная компиляция

- ☞ независимая трансляция частей программы разделенной на отдельные файлы с последующим объединением их компоновщиком в один исполняемый файл:



Цели и преимущества

- разделение большой программы на небольшие, более понятный части
- отладка каждой из частей делается независимо
- изменениях в одной из частей ведет к компиляции только этой части

main.c

```
#include "functions.h"
int main() {
    fun_debug = 1; // debug mod
    init();
    run(10);
    stop();
}
```

functions.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
extern int fun_debug; // external variable
void init();
int run(int nevents);
void stop();
#endif
```

functions.c

```
#include <stdio.h>
#include "functions.h"

// definition of fun_debug
int fun_debug = 0;

// invisible outside this file
static int init_done = 0;

void init() {
    if( init_done ) return;
    if( fun_debug ) {
        printf("This is init()\n");
    }
    init_done = 1;
}
```

```
int run(int nevents) {
    if( fun_debug )
        printf("This is run()\n");
    for(int i=0; i<nevents; ++i) {
        if( fun_debug )
            printf(" event# %i\n",i);
    }
    return 1;
}

void stop() {
    if( fun_debug )
        printf("This is stop()\n");
    init_done = 0;
}
```

Спецификатор `extern`

- спецификатор `extern` используется чтобы *декларировать* переменную или функцию, определённую в другом файле
- все функции по умолчанию `extern`
- определение функции или выделение памяти для переменной должна выполняться только один раз, в одном месте
- проверка, что переменная или функция действительно существуют выполняется компоновщиком

Note: сравните со `static`

☞ `static` для глобальной переменной или функции ограничивает их область видимости единицей трансляции («файлом») в котором они определены

Компиляция примера

- За один заход:

```
gcc main.c functions.c
```

- Поэтапно:

```
gcc -c main.c
```

```
gcc -c functions.c
```

```
gcc main.o functions.o
```

Output:

```
This is init() function
```

```
This is run() function
```

```
event# 0
```

```
event# 1
```

```
...
```

```
event# 9
```

```
This is stop() function
```

Введение в make и Makefile

Автоматизация раздельной компиляции с make

- Существует несколько версий `make`: очень популярна `gmake` (GNU make)

Очень простой пример файла Makefile

```
PROGRAM := test                      # variable

all: $(PROGRAM)                      # target-1: component

$(PROGRAM): functions.o main.o      # target-2: component
    @echo "Hello make, example"     # TAB before the command
    $(CC) -o $(PROGRAM) functions.o main.o
```

> `make [target]`

```
cc      -c -o functions.o functions.c
cc      -c -o main.o main.c
Hello make, example
cc -o test                         functions.o main.o
```

Чуть более сложный пример Makefile

```
PROGRAM := test                      # program name
SRCS    := functions.c  main.c       # source files
OBJS    := $(SRCS:.c=.o)             # transform filenames .c -> .o
CFLAGS   := -Wall -I.                # compilation flags
LIBS     := -lm                      # options for linking

# add GSL-library
GSLCFLAGS := $(shell gsl-config --cflags)  # shell-command
CFLAGS    += $(GSLCFLAGS)
GSLLIBS   := $(shell gsl-config --libs)    # shell-command
LIBS      += $(GSLLIBS)

%.o : %.c                          # suffix rule
        $(CC) -c $(CFLAGS) $<

all:      $(PROGRAM)               # target-1

# see next slide
```

... продолжение

```
$(PROGRAM): $(OBJS)          # target-2
    $(CC) $(CFLAGS) $(OBJS) $(LIBS) -o $(PROGRAM)
    @echo "done, $(PROGRAM)"

clean:                      # target-3
    @rm -f *.o $(PROGRAM)

depend:                     # target-4
    @echo "Generating make.depends"
    @rm -f make.depends
    $(CC) -MM $(CFLAGS) $(SRCS) > make.depends
    @echo "done"

include $(wildcard *.depends) # include dependency files
```

- > `make depend` создания файла «зависимостей»
- > `make` сборка программы
- > `make clean` «очистка»

Введение в cmake и CMakeLists.txt

Автоматизация создания Makefile

- Утилита `cmake` создает из файла сценария `CMakeLists.txt` файлы для сборки с помощью `make`, MVS, Android Studio ...

Очень простой пример файла `CMakeLists.txt`

```
CMAKE_MINIMUM_REQUIRED( VERSION 2.4 )
PROJECT(test)
INCLUDE_DIRECTORIES($PROJECT_SOURCE_DIR)
ADD_EXECUTABLE(test functions.c main.c)
```

Как использовать:

```
# создаем папку для Makefiles в которой будем собирать проект
> mkdir build
> cd build
> cmake ../          # папка где лежит CMakeLists.txt
> make
```

② ВОЗМОЖНАЯ реализация дерева на C++

```
struct Node {  
    Node(int p=0) : pdg_code(p) {}; // constructor  
    int pdg_code = 0;  
    std::vector<size_t> children; // position of children in Tree.nodes  
};  
struct Tree {  
    Tree() {nodes.reserve(2);} // constructor  
    std::vector<Node> nodes;  
};
```

«Обход» дерева – печать элементов

```
void print_node(const Tree& tr, size_t ch = 0) {  
    const Node& N = tr.nodes[ch];  
    std::cout << N.pdg_code << '(';  
    for ( auto ch : N.children ) {  
        print_node(tr, ch); // recursion  
    }  
    std::cout << ')' << (ch > 0 ? ',' : '\n');  
}
```

Пример: дерево распадов $\eta \rightarrow \pi^+ \pi^- \pi^0$; $\pi^0 \rightarrow \gamma\gamma$

```
Tree root;
root.nodes.push_back(221); // [0] == root, eta = 221

root.nodes.push_back(211); // [1], pi+ = 211
root.nodes[0].children.push_back(1);
root.nodes.push_back(-211); // [2], pi- = -211
root.nodes[0].children.push_back(2);
root.nodes.push_back(111); // [3], pi0 = 111
root.nodes[0].children.push_back(3);

root.nodes.push_back(22); // [4], gamma = 22
root.nodes[3].children.push_back(4);
root.nodes.push_back(22); // [5], gamma = 22
root.nodes[3].children.push_back(5);

print_node(root); // 221(211() -211() 111(22() 22() ) )
```