C++. Первый курс. Теоретический минимум.

Группа M3109

1. Стандартные типы данных и представление чисел в памяти

По Бьёрну Страуструпу, **фундаментальные типы данных** — это базовые типы, которые гарантированно есть в языке C++.

1) `int`

Тип `int` представляет целое число, обычно занимает **4 байта** (32 бита). Диапазон значений:

$$[-2^{31},\ 2^{31}-1]$$

(то есть от `-2147483648` до `2147483647`).

Дополнительный код (two's complement). Это способ представления отрицательных чисел в памяти.

- Старший бит — **знаковый** (0 — положительное, 1 — отрицательное число).
- Чтобы получить отрицательное число $-X$, нужно:
  1. инвертировать все биты X (заменить 0 на 1 и наоборот);
  2. прибавить 1.

  Так как один бит занят под знак, остаётся 31 бит для значения, что даёт диапазон:

$$[-2^{31},\ 2^{31}-1].$$

2) `long`

Тип `long` может занимать **4 или 8 байт** в зависимости от архитектуры (32- или 64-битной). Используется редко, поскольку его размер нефиксирован. В современных программах предпочтительно использовать `int` или `long long`.

3) `long long`

Тип `long long` занимает **8 байт** (64 бита). Диапазон значений:

$$[-2^{63},\ 2^{63}-1].$$

4) `char`

Тип `char` — это **целочисленный тип**, занимающий ровно 1 байт. Может быть:

- `signed char`: диапазон `[-128, 127]`;
- `unsigned char`: диапазон `[0, 255]`.

С ним можно выполнять арифметические операции, но важно учитывать возможное переполнение.

5) `float`

Тип с плавающей точкой, занимает **4 байта**. Примерный диапазон:

$$\pm 1.4 \times 10^{-45} \text{ до } \pm 3.4 \times 10^{38},$$

точность — около 6–9 значащих цифр.

6) `double`

Тип с плавающей точкой двойной точности, занимает **8 байт**. Диапазон:

$$\pm 4.9 \times 10^{-324} \text{ до } \pm 1.8 \times 10^{308},$$

точность — около 15–17 значащих цифр.

Число с плавающей точкой состоит из трёх основных компонентов:

- **Знак** (S) — один бит, указывающий знак всего числа
- **Порядок** (E) — целое число, определяет масштаб
- **Мантисса** (M) — целое число, определяет точность

Математическое представление

Число представляется в виде:

$$(-1)^S \times M \times B^E$$

где:

- S — знак (0 для положительных, 1 для отрицательных)
- B — основание системы счисления (обычно 2 или 10)
- E — порядок (экспонента)
- M — мантисса

Связь с научной нотацией

Десятичное число в научной нотации записывается как:

$$R \times 10^E$$

где R — число в полуинтервале $[1; 10)$.

В нормализованной форме:

- Модуль $|R|$ становится мантиссой M
- E становится порядком
- $S = 1$ тогда и только тогда, когда $R < 0$

Пример

Число -3.5 в нормализованной форме:

$$-3.5 = (-1)^1 \times 3.5 \times 10^0$$

где:

- $S = 1$ (отрицательное число)
- $M = 3.5$ (мантисса)
- $B = 10$ (основание)
- $E = 0$ (порядок)

Терминология

- **Порядок** также называют **экспонентой**
- **Показатель степени** — синоним порядка
- **Нормализованная форма** — когда мантисса находится в определённом диапазоне

Стандарт IEEE 754.

- **float (32 бита):** `S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM` 1 бит — знак, 8 бит — экспонента, 23 бита — мантисса.
- **double (64 бита):** `S EEEEEEEEEEE MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM` 1 бит — знак, 11 бит — экспонента, 52 бита — мантисса.

При арифметических операциях значения `float` часто неявно преобразуются к `double`.

7) `long double`

Тип расширенной точности, занимает **не менее 8 байт** (на практике 10, 12 или 16 байт в зависимости от реализации).

8) `bool`

Тип логического значения, может быть только `true` или `false`. Занимает **1 байт** (1 бит технически невозможен из-за архитектурных ограничений).

Модификатор `unsigned`

Для целочисленных типов можно добавить модификатор `unsigned` — это убирает знак числа. Например:

$$\texttt{unsigned int: } [0, 2^{32} - 1]$$

Весь диапазон значений используется для неотрицательных чисел. Стоит отметить , что при переполнении , знаковые типы хранят в себе мусор , а беззнаковые берут остаток от деления от своего максимального значения

2. Литералы

Литерал — это последовательность символов, обозначающая константное значение. Примеры:

- `1` — целочисленный литерал;
- `'a'` — символьный;
- `"abc"` — строковый;
- `0.2f` — вещественный (`float`);
- `true`, `false`, `nullptr` , `}` — специальные литералы.

Неявные преобразования. Некоторые типы могут неявно преобразовываться:

- `double → int`, `char → int`, `int → bool` и т.д.

Например:

- `int x = 5.0;` — корректно, `5.0` приводится к `int`;
- `5 / 3 = 1`, но `5 / 3.` даёт `1.666...`.

Суффиксы литералов. Постфикс (суффикс) уточняет тип:

`1U` → `unsigned int`, `3.14f` → `float`.

3. Массивы

Массивы объявляются так:

```
type name[количество_элементов];
```

Доступ к элементу осуществляется по индексу: `a[i]`.

- Индексация начинается с 0.
- Размер массива должен быть фиксирован на момент компиляции (для статических массивов).
- При выходе за пределы массива программа получает ошибку времени выполнения (`RE`) или сегфолт (`Segmentation Fault`).

Расположение в памяти. Элементы массива хранятся последовательно.

При запуске программы массив, объявленный в функции, располагается в стеке, а динамически созданный — в куче.

При запуске программы часть памяти забирает ОС — там хранится техническая информация, исполняемый код и прочее. Если мы объявили `int x`, в стеке выделяется 4 байта, по адресу которых лежит значение `x`. Далее в области видимости могут появляться и исчезать переменные, и освобождённые байты переиспользуются под другие данные.

Если объявить массив `int a[100]`, то это сплошные 400 байт памяти. Когда мы пишем `a[5]`, программа делает 5 шагов вправо от начала массива, размер шага — размер типа (4 байта). Отсюда становится понятна нулевая индексация.

Указатели

Указатель — это тип данных, который служит для представления адресов в памяти. Объявляется так:

```
int* x;
```

`int*` — это тип «указатель на `int`», то есть адрес памяти, по которому лежит целое число.

С указателями можно смотреть, что под ними лежит, с помощью унарной звёздочки — оператора разыменования:

```
int* x;
*x;
```

Это значит: возьми указатель и верни то, что под ним лежит. Можно писать и `int *x`, разницы нет — это вкусовщина.

Адрес переменной.

```
int b;
int *y = &b;
```

Унарный амперсанд `&` берёт адрес переменной. Рекомендуется поэкспериментировать, чтобы привыкнуть.

Арифметика указателей. Указатели можно инкрементировать, декрементировать и вычитать друг из друга. Например:

```
int *x;
x = x + 5;
```

Это означает: сделать 5 шагов вправо по типу `int` (то есть на 20 байт). Выражения `+=`, `-=`, `++`, `-` работают аналогично.

Разность указателей. Разность двух указателей — это количество элементов (шагов) между ними.

Связь с массивами. Выражение `a[i]` эквивалентно `*(a + i)`. Поскольку операция коммутативна, `5[a]` тоже корректно!

Можно делать массивы из указателей на что угодно.

Динамическая память. Если нужно создать большой массив (больше, чем позволяет стек), используют **динамическую память** с помощью операторов `new` и `delete`. Это выходит за рамки теоретического минимума.

Функции

Функция — это именованный блок кода, который можно вызвать многократно (не все виды — лямбду только один раз). У функции есть сигнатура — тип, аргументы и название, и есть тело (код в фигурных скобках).

Когда мы вызываем функцию, происходит инициализация параметров значениями аргументов. После завершения работы функции переменные внутри неё уничтожаются из-за области видимости. Оператор `return` возвращает результат выполнения функции, и его тип должен совпадать с типом функции. Если функция ничего не возвращает — она имеет тип `void`.

Указатель на функцию

Функция, определённая в программе, тоже хранится в памяти (в виде инструкций). Чтобы получить её адрес, можно написать:

```
p = &f;
```

Пример:

```
int foo(double a, char* b) { return int(a); }
int (*p)(double, char*) = &foo;
```

Тип переменной `p` должен полностью соответствовать типу функции.
`main` нельзя брать по адресу.

Особый случай: `void*` — это указатель общего назначения, но его нельзя разыменовать.

Итог: Указатель на функцию — это переменная, которая хранит адрес функции в памяти и позволяет вызывать её косвенно.

Структуры:

Структура это один из способов создать свой тип данных , синтаксис у нее такой

```
Struct [name]{
[поле структуры];
[поле структуры];
};
```

Поле структуры - данные , которые хранятся в структуре , условно :

```
Struct S{
int x=1;
char c;
double d;
};
```

Стоит отметить , что при ее инициализации в конце стоит поставить ; , что выглядит довольно специфично . На примере видим , что поля структуры мы можем проинициализировать по умолчанию , если мы делаем так , то при создании структуры у нас будет конкретное поле проинициализированно , в остальных же будет мусор , также можно сделать вот так :

```
S s{2 , 'a' , 2.14};//агрегатная инициализация
```

При такой инициализации обязательно нужно обращать внимание на последовательность типов , иначе мы будем падать по Compile Error , также ,если у нас какие то поля проинициализированы по умолчанию , можно делать так :

```
S s{'a' , 2.14}
```

И это будет работать . Мы обращаемся к полям структуры следующим образом :

```
s.x = 2;
s.c='b';
s.d;
```

Таким же образом можем присваивать значения и все такое , также это работает и для вывода через

```
std::cout<<s.x<<s.d;
```

Если же мы хотим использовать указатель на структуру , то можно обращаться через оператор -> , следующим образом :

```
S* ps = new S{2};
std::cout<<*ps.x<<" "<< ps.c <<" "<< ps.d <<'\n';
std::cout<<ps->x<<" "<< ps.c <<" "<< ps.d <<'\n';
```

Выводы будут такие же , следовательно стрелочка это разыменование и обращение к полю. Помимо полей мы можем объявлять в структурах методы , это операции , которые мы можем применить к структуре .Пример :

```
Struct s{
int x = 1;
double y = 1.1;
char c = 'F';
void f(){
std::cout<<x<<" "<<y<<" "<<c<<'\n';
}
};
int main(){
S s{2};
s.f();
}
```

Тут все вроде как понятно , в структуре мы храним какие то поля , следовательно методы имеют к ним доступ и в сигнатуре не нужно их прописывать . Экземпляры структуры , т.е. в нашем случае s , нразываются объектами . Помимо этого , в метод мы также можем передать какие то параметры в сигнатуру , например :

```
    Struct s{
int x = 1;
double y = 1.1;
char c = 'F';
void f(int z){
std::cout<<x<<" "<<y<<" "<<c<<" "<<z<<'\n';
}
void foo(int x){
std::cout<<x <<" "<< this->x <<'\n'
}
};
int main(){
S s{2};
s.f(3);
s.foo(5);
}
```

Тогда у нас закрадывается вопрос , что если в поле структуры у нас будет наименованна та же переменная , что мы хотим передать в метод ? Смотрим на пример и видим ключевое слово this , это слово позволяет нам явно обращаться к указателю на данный объект из его тела , в примере мы находимся в теле нашей структурки , далее мы явно обращаемся к нашей структуре через указатель , используя this и стрелочкой разыменовываем и обращаемся к конкретному полю нашей структуры.Методы в структурах видят друг друга всегда , то есть разницы нет в каком порядке мы их объявляем и нам в целом не нужен прием forward declaration , также мы можем делать перегрузку методов , посмотрим на пример :

```
    Struct s{
int x = 1;
double y = 1.1;
char c = 'F';
void f(int z){
std::cout<<x<<" "<<y<<" "<<c<<" "<<z<<'\n';
}
void foo(int x){
std::cout<<x <<" "<< this->x <<'\n';
}
void foo(){
std::cout<<"Fyodor Kurilov"<<'\n';
}
};
int main(){
S s{2};
s.f(3);
s.foo(5);
s.foo();
}
```

И у нас все прекрасно работает ! Мы помимо всего прочего можем объявлять метод внутри структуры и определять вне ее , выглядит это так :

```
        Struct s{
int x = 1;
double y = 1.1;
char c = 'F';
void f(int z){
std::cout<<x<<" "<<y<<" "<<c<<" "<<z<<'\n';
}
void foo(int x){
std::cout<<x <<" "<< this->x <<'\n';
}
void foo(){
std::cout<<"Fyodor Kurilov"<<'\n';
}
void cpp();
};
void S::cpp(){
cout<<"CPP"<<'\n';
}
int main(){
S s{2};
s.f(3);
s.foo(5);
s.foo();
}
```

И вот это определение метода вне структуры никак нам не накладывает ограничения , наш метод по прежнему видит все ее поля и другие методы . Мы можем внутри структур объявлять другие структуры и эти структуры будут называться inner structers , то есть внутренние структуры , пример :

```
Struct S{
```

```
int x = 1;
Struct SS{
double d = 1.1;
};
};
int main(){
S::SS ss;
}
```

Из вне мы тоже можем увидеть эту структуру , но мы должны явно указать , октуда мы ее достаем . Поля инициализируются в порядке объявления.Мы можем делать очень много разных вещей с внутренней структурой , SS будет таким же объектом как и S , следовательно мы можем делать с ним все тоже самое , что из S , единственное отличие в том , что чтобы достать SS нам надо обязательно указывать явно откуда мы берем это . Условно :

```
Struct S{
int x = 1;
Struct SS{
double d = 1.1;
int y = x;
};
};
int main(){
S::SS ss;
}
```

Не будет работать , потому что , хоть у нас и SS лежит в S , но это не говорит нам о том , что они как-то связаны . Связаны они только тем , что чтобы создать SS , мы обязаны обратиться к S , чтобы мы смогли увидеть , что у нас есть SS .
Продолжим дальше , еще можно объявлять и определятьь структуры внутри функций , фнукции внутри функций нельзя , структуры внутри структуры можно и структуру внутри функции тоже можно !

```
Struct S{
int x = 1;
Struct SS{
double d = 1.1;
int y = x;
};
};
void g(){
struct Another{
int a = 1;
int b = 2;
};
Another a;
S::SS ss;
}
int main(){
g();
}
```

будет работать , это называется локальные структуры , если они внутри функций . Еще моментик про размер структуры S , мне лень писать пример и контр пример , но если мы выведем размер S , у которого лежит внутри SS и потом выведем размер S и закомментим SS , то размер у нас будет

одинаковый , также размер структуры зависит от того , в каком порядке мы перечисляем поля , потому что поля кладутся подряд , но допустим переменные типа double мы можем ложить только по адресу кратному 8 , в отношении полей компилятор не имеет права их переупорядочивать , т.е. мы ложим все подряд , но компилятор решает пропускать или нет какие то куски памяти , пример:

```
Struct S{
int x = 1;
char c = 'a';
double y = 3.14;
}
Struct SS{
int x = 1;
double y = 3.14;
char c = 'a';
}
```

Эти структуры будут весить по разному , сейчас разберемся . Как лежит в памяти структура S : сначала ложим 4 байта под x , потом 1 байт под c , потом нам нужно пропустить 3 байта и только потом мы можем ложить y , поскольку double мы можем класть только в адреса памяти кратные 8 ! Итого у нас S весит 16 байт . Теперь про SS , мы ложим 4 байта под x , потом пропускаем 4 байта ложим double , но размер его 24 ... почему так ? Дело в том , что память округляется дл наибольшего поля , следовательно мы исходим из того , что следующую SS мы должны положить впритык к предыдущей , поэтому компилятор дополняет до 8 байт , для того чтобы следующая SS лежало сразу за прошлой или намного проще понять его работу можно так . Мы смотрим наибольший тип в случае SS это double , т.е. 8 байт , потом мы заполняем все и выходит 17 байт , далее мы должны найти такое число , чтобы оно было больше или равно 17 и делилось на 8 и жто будет его размер ! . Мы ложим x , это 4 байта , дополняем 4 байта для y и ложим его , итого 4+4+8 = 16 байт и ложим c и становится 17 байт , далее находим число больше или равное 17 и кратное 8 и это 24 .
Теперь поговорим о неименованных структурах , т.е.

```
Struct s{
struct {
int x = 1;
double y = 2;
};
int x = 2;
};
int main(){
std::cout<<s.x<<" "; \\ error : duplicate member;
\\сделаем вид , что сейчас мы закоменитили строку int x = 2;
std::cout<<s.x<<" ";
}
```

обращение вот так не будет работать , поскольку компилятор ничего не понимает . Поэтому стоит обращаться вот так :

```
    struct s{
struct {
int x = 1;
double y = 2;
};
//int x = 2;
};
int main() {
s obj;
```

```
    std::cout << obj.x << " " << obj.y;
}
```

и это работает !!!

Обьединения Обьединения это union , суть работы примерна та же , что и у структур , но делаем исключение для работы с памятью . Все поля обьединения делят одно и то же место в памяти . Синтаксис такой же

```
    union [name] {
    [поле];
    [поле];
};
```

Размер union равен размеру самого большого поля (с учётом выравнивания); В каждый момент времени активно только одно поле — то, которое мы записали последним; Если мы перезаписываем одно поле, другие меняются (или становятся мусором), так как память у них общая.Пример :

```
    union U {
    int x;
    double y;
};

int main() {
    U u;
    u.x = 10;                  // Записали int
    std::cout << u.x << '\n'; // 10
    u.y = 3.14;                // Записали double
    std::cout << u.y << '\n'; // 3.14
    std::cout << u.x << '\n'; // мусор, потому что память общая
}
```

То есть мы не можем инициализироватт сразу несколько полей , поскольку память общая)) Теперь посмотрим на это

```
    union U {
    int a;     // 4 байта
    char c;    // 1 байт
    double d;  // 8 байт
};
int main() {
    std::cout << sizeof(U); // 8
}
```

Размер union будет просто максимальным размером его поля . Указатели и доступ работают ровно также , с this , стрелочками и всем таким , даже еще можно делать методы внутри юниона , но у этих методах не должно быть конструктора или конструктора не по умолчанию , так было до C++11 , но теперь мы можем делать даже конструкторы и деструкторы , но нужно вручную управлять временем жизни полей с помощью placement new и явного вызова деструктора, теперь примеры :

```
    union U {
    int x;
    double y;
};
```

```
int main() {
    U u;
    U* pu = &u;
    pu->x = 42;
    std::cout << pu->x << '\n';
}
```

тут все ок , это так реально работает

```
    #include <string>
#include <iostream>

union Data {
    int i;
    std::string s; // ok в C++11 и выше

    Data() { new(&s) std::string("Hello"); }
    ~Data() { s.~basic_string(); }
};
```

это также работает ! можно это тоже комбинировать , см. пример

```
    struct S {
    int type;
    union {
        int i;
        double d;
        char c;
    };
};

int main() {
    S s;
    s.type = 1;
    s.i = 42;
    std::cout << s.i << '\n';
}
```

тут просто смотрите как можно делать все это вместе ! короче итого: union хранит все поля в одной памяти; используется, когда нужно сэкономить память или интерпретировать одни и те же биты по-разному; размер равен наибольшему полю; можно использовать анонимно внутри struct; до C++11 — только тривиальные типы, с C++11 — можно управлять объектами вручную; методы, указатели и инициализация работают аналогично struct.

Виртуальное адресное пространство

Начнём с того, что существует три основных типа памяти в C++: **статическая**, **динамическая** и **автоматическая**. Все они различаются по времени жизни данных и способу выделения.

Когда мы запускаем программу, операционная система загружает её в **оперативную память** (RAM — Random Access Memory). Для процесса выделяется определённый *непрерывный диапазон виртуальных адресов* — его **виртуальное адресное пространство**. Каждый процесс в системе имеет собственное адресное пространство, и если программа попытается обратиться к адресу за его пределами, произойдёт ошибка доступа к памяти (**segmentation fault**).

Внутри этого пространства память также разделена на области (сегменты). Основные из них:

- **.text** — область, где хранится исполняемый машинный код программы.
- **.data** / **.bss** — область данных: глобальные, статические переменные и литералы.
- **stack** — стековая область памяти, где располагаются локальные переменные и информация о вызовах функций.
- **heap** — динамическая память, используемая при выделении через `new` или `malloc`.

Когда программа исполняется, процессор получает команды не напрямую из файла, а из оперативной памяти. При запуске ОС загружает бинарный код программы (машинные инструкции) в сегмент **.text**, и процессор выполняет их, обращаясь к ним через виртуальные адреса.

—

Сегменты памяти

Статическая память (data segment)

Статическая память (`.data`, `.bss`) содержит все **глобальные и статические переменные**, а также строковые литералы и таблицы виртуальных функций. Эти данные существуют всё время выполнения программы и освобождаются автоматически после завершения процесса.

Пример:

```
int global_var = 10;      // глобальная переменная → data
static int counter = 0;   // статическая переменная → data
```

—

Стековая память (stack segment)

Стек — это **автоматическая память**, в которой размещаются **локальные переменные** и **кадры вызовов функций**.

```
int main() {
    int x = 5;  // переменная x хранится в стеке
}
```

Почему именно "стек"? Потому что принцип его работы — **LIFO** (Last In, First Out). Процессор поддерживает специальный указатель — **stack pointer (SP)**. При создании новой локальной переменной или вызове функции происходит `push` данных на стек (указатель смещается вниз), а при выходе из функции — `pop` (указатель возвращается вверх).

Таким образом, стек управляется аппаратно — при вызове функции создаётся **кадр стека (stack frame)**, в котором хранятся:

- локальные переменные функции;
- адрес возврата;
- сохранённые значения регистров (например, `rbp`, `rip`);
- переданные аргументы.

Когда функция завершается, её кадр удаляется — стек возвращается в прежнее состояние.

```
void foo() {
    int a = 10;  // создаётся новый кадр foo
}
int main() {
    foo();       // push кадра foo
}                // после возврата pop
```

—

Динамическая память (heap segment)

Динамическая память используется, когда объём данных заранее неизвестен и должен определяться во время выполнения программы. Эта память выделяется и освобождается вручную с помощью:

```
int* p = new int(42);   // выделение
delete p;               // освобождение
```

Выделение памяти здесь управляется не процессором напрямую, а библиотекой рантайма (например, `malloc/new`), которая запрашивает у операционной системы блоки виртуальной памяти через системные вызовы (`brk`, `mmap`).

—

Связь между сегментами и виртуальной памятью

Все эти области — **части одного виртуального адресного пространства процесса**. Каждая из них имеет собственные права доступа (например, `.text` только для чтения и исполнения, стек — для чтения и записи). Механизм виртуальной памяти с помощью **MMU (Memory Management Unit)** сопоставляет виртуальные адреса с физическими страницами ОЗУ.

Это позволяет:

- изолировать процессы друг от друга;
- использовать одинаковые адреса в разных процессах без конфликтов;
- эффективно управлять памятью (подкачка, ленивое выделение, защита от переполнения стека).

—

Заключение

Таким образом, **виртуальное адресное пространство** — это логическое представление памяти программы, разделённое на сегменты с различным назначением и правилами жизни данных. Вся работа с памятью в C++ (глобальные, локальные, динамические объекты) фактически происходит внутри этих сегментов.

Стек вызовов

Стек вызовов — это структура, которая хранит информацию о том, какие функции были вызваны и в каком порядке. Каждый раз, когда вызывается новая функция, в стек добавляется новый кадр (frame). Когда функция завершается, этот кадр убирается из стека. Таким образом, стек работает по принципу LIFO — последним вошёл, первым вышел.

В кадре стека хранится адрес возврата (куда нужно вернуться после окончания функции), локальные переменные, аргументы функции и сохранённые регистры процессора. Когда функция вызывается, процессор делает `push` этих данных в стек, а при возврате — `pop`.

```
void bar() { int c = 3; }
void foo() { int b = 2; bar(); }
int main() { int a = 1; foo(); }
```

Порядок работы:

1. создаётся кадр `main`
2. вызывается `foo` — создаётся кадр `foo`

3. вызывается `bar` — создаётся кадр `bar`

4. `bar` завершилась — кадр удаляется

5. `foo` завершилась — кадр удаляется

6. `main` завершилась — стек пуст

В памяти это выглядит так:

```
bar(): c
foo(): b
main(): a
```

Стек вызовов живёт в стековой области памяти. Если программа вызывает слишком много функций (например, при бесконечной рекурсии), стек переполняется — это называется *stack overflow*.

Куча

Куча (heap) — это область памяти, которая используется для динамического выделения объектов во время выполнения программы. В отличие от стека, память в куче не выделяется автоматически, а управляется вручную программистом.

Для работы с кучей в C++ используются операторы `new` и `delete`.

```
int* p = new int(10);   // выделение памяти в куче
delete p;               // освобождение памяти
```

При вызове `new` программа запрашивает память у операционной системы. Если памяти достаточно, возвращается указатель на выделенный участок. Если памяти нет, выбрасывается исключение `std::bad_alloc`.

Объекты в куче живут до тех пор, пока их явно не удалить. Если не вызвать `delete`, произойдёт утечка памяти, потому что память останется занята до завершения программы.

Можно выделять и массивы:

```
int* arr = new int[5];
delete[] arr;
```

Куча не имеет фиксированного размера, но выделение памяти в ней происходит медленнее, чем в стеке, так как система должна найти свободный участок подходящего размера. Также при частом выделении и освобождении память может фрагментироваться.

Куча используется, когда размер данных заранее неизвестен или должен сохраняться за пределами области видимости функции.

Перегрузки функций

Перегрузка функций — это возможность объявлять несколько функций с одинаковым именем, но с разными параметрами. Компилятор сам выбирает, какую версию функции вызвать, в зависимости от типов и количества аргументов.

Пример:

```
void print(int x) { cout << "int: " << x << endl; }
void print(double x) { cout << "double: " << x << endl; }
void print(string s) { cout << "string: " << s << endl; }

int main() {
    print(5);        // вызов print(int)
```

```
    print(3.14);     // вызов print(double)
    print("Hi");     // вызов print(string)
}
```

Здесь все функции называются одинаково — `print`, но принимают разные типы аргументов. Это удобно, потому что одно действие (например, вывод) можно выполнять для разных типов данных.

Перегрузка возможна, если отличаются:

- количество параметров;
- типы параметров;
- порядок параметров.

Возвращаемый тип не участвует в выборе перегрузки — то есть нельзя различать функции только по типу возвращаемого значения.

Перегрузка функций применяется для удобства и читаемости кода. Компилятор сам выбирает подходящую версию на этапе компиляции (во время компиляции, а не выполнения).

Ссылки

Ссылка — это другое имя (псевдоним) для уже существующей переменной. После создания ссылка всегда указывает на один и тот же объект, и изменить, на что она ссылается, нельзя.

Объявление ссылки:

```
int a = 5;
int& ref = a;   // ref - это ссылка на a
ref = 10;       // изменяет a, теперь a = 10
```

Ссылка не создаёт новую переменную, она просто даёт возможность обращаться к уже существующей под другим именем.

Ссылки часто используются:

- для передачи аргументов в функцию без копирования:

  ```
  void change(int& x) { x = 100; }

  int main() {
      int a = 5;
      change(a);  // передаём ссылку
      // теперь a = 100
  }
  ```

- для возвращения результата из функции без лишнего копирования;
- для удобного доступа к элементам контейнеров.

Также есть константные ссылки:

```
const int& r = a;
```

Они позволяют читать значение, но запрещают его изменять.

Ссылки не могут быть `null` и должны быть инициализированы сразу при объявлении.

Namespace

Пространство имён (namespace) — это способ группировать функции, переменные и классы, чтобы избежать конфликтов имён в больших программах.

Объявление пространства имён:

```
namespace math {
    int add(int a, int b) { return a + b; }
    int sub(int a, int b) { return a - b; }
}
```

Чтобы обратиться к функции внутри пространства имён, используется оператор `::` (оператор разрешения области видимости):

```
int main() {
    int x = math::add(2, 3);
    cout << x;
}
```

Также можно подключить всё пространство имён сразу:

```
using namespace math;
int main() {
    cout << add(2, 3);
}
```

Но так делать нежелательно в больших проектах, потому что может возникнуть конфликт имён, если в разных пространствах будут одинаковые функции.

Можно создавать вложенные пространства:

```
namespace app {
    namespace utils {
        void log() {}
    }
}
app::utils::log();
```

Пространства имён позволяют структурировать код и разделять логические части программы.

Препроцессор

Препроцессор — это часть компилятора, которая обрабатывает исходный код перед компиляцией. Он выполняет команды, начинающиеся со знака #, такие как подключение файлов, определение констант и условную компиляцию.

Основные директивы препроцессора:

- `#include` — подключает другой файл в программу.

  ```
  #include <iostream>   // стандартная библиотека
  #include "myfile.h"   // свой файл
  ```

- `#define` — задаёт макрос (константу или подстановку).

  ```
  #define PI 3.14
  #define SQR(x) ((x)*(x))
  ```

После этого препроцессор просто подставляет вместо имени текст макроса:

```
int r = 2;
double s = PI * SQR(r); // подставит 3.14 * (2*2)
```

- `#ifdef`, `#ifndef`, `#endif` — условная компиляция.

```
#define DEBUG
#ifdef DEBUG
cout << "Debug mode\n";
#endif
```

Если макрос `DEBUG` определён, код внутри будет скомпилирован, иначе — пропущен.

Препроцессор не знает о типах и не проверяет синтаксис C++. Он просто заменяет текст до начала компиляции. После работы препроцессора компилятор получает уже изменённый код.