2.1. Указатели

Указатель – адрес переменной в ОП.

```
int x = 1;
int* p = &x;
int y = *p;
*p = 2;
int** pp = &p;
```

← взятие адреса переменной

← взятие значения под указателем

```
p += 1;
```

← переход по памяти на 1*sizeof(int) байт

```
vector<int> v{1, 2, 3, 4, 5};
int* p = &v[0];
*(p+1) == v[1];
*(p+2) == v[2];
```

memory reuse

```
int a = 1;
int* p = &a;
{
    int b = 2;
    p = &b;
}

std::cout << p << '\n';
std::cout << *p << '\n'; // UB, but most likely 2

int c = 3, d = 4, e = 5, f = 6;
std::cout << &c << ' ' << &d << ' ' << &e << ' ' << &f << '\n';

++*p; // still UB, and it's possible for one of c, d, e, f to be changed

std::cout << c << d << e << f; // possibly something different from 3456
std::cout << *p; // possibly not 2 already
```

возможный вывод:

```
3457
7
```

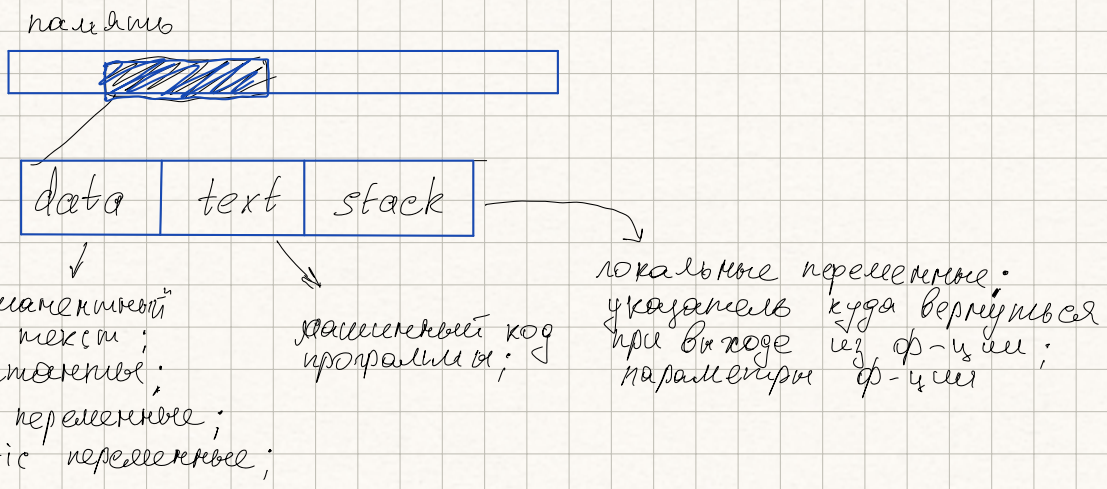
`void*` – указатель на произвольный байт в памяти, неизвестно какой тип под ним

```
foo(void* p)
{
    (int*)p;
    *p;
    ++*p;
    void* pp = nullptr;
}
```

нельзя

– тип "нулевой указатель" неявно может кастоваться к любому указателю

2.2. Kinds of memory



data – static memory – определяется во время компиляции

text – во время компиляции

stack – automatic memory – обычно ~8 Мб

В stack заносятся локальные переменные по принципу "стека". Но компилятор может их переупорядочивать в рамках одной области видимости.

↑ см. пример memory reuse (перем. f легла на место a)

heap – dynamic memory – runtime память,

```
int * p = new int(5);
*p = 1
delete p;
```

– попросить" у ОС выделить 4 байта из динамической памяти (всё сложнее)

– "вернуть" эту память ОС

Если забыли "вернуть" – утечка памяти. Память будет занята пока не завершится программа. memory leak

2.3. Arrays

```
int a[10];
int b[] = {1, 2, 3}

int* p = a + 2;
p[-1];
```

а можно неявно кастовать к указателю

p[1] <=> *(p+1)

1[p] <=> *(1+p)

p[-1] <=> *(p-1)

```
sizeof(a) / 40
```

Массивы почти указатели

2.4. References

Ссылки – это просто другое именование объекта.

В некоторых случаях компилятор ссылки хранит как указатели, например, переменные в функции, переданные по ссылке, хранятся как указатели.

```
int x = 0;
int& y = x;
&y == &x; // true
```

```
int* p = nullptr;
int*& r = p;

int&* // нельзя
int& a[10] // нельзя

int (&a)[10] = b;

void (&f)() = g; // ссылка на ф-цию, f теперь ссылка на g

void& y = x; // нельзя
int& 5 = 5; // нельзя
```

```
int& f() {
    int x = 0;
    return x;
}
```

Dangling reference (висячая ссылка)

Это UB

```
int x = f();
```

Тоже UB (x уже снят со стека)

```
void f(int);
void f(int&);
```

Ambigious call (или CE?)

Т.е. так нельзя

2.6. Constants

Константные – тип, над которым запрещены некоторые операции (те, которые модифицируют объект)

Точнее, **модифицирующие операции** — это не операции, которые нельзя выполнять у константы.

Операции: `=`, `+=`, ..., `++`, `--`.

```
const int x = 5;

int * p = &x;          ← CE
const int* p = &x;     ← значение под указателем неизменяемо
++p;
int a = 1;
int* const b = &a;     ← сам указатель неизменяем
const int * const c =

const int * d = &a
++*d;
++a;
++d;
```

```
int a = 1
const int& b = a;
++b;                   ← b тоже поменялось
++a;
int& c = b;            ← CE
```

Как передавать объекты в ф-цию?

```
void f(T x);           ← создаём временный объект
                         (примитивные типы передаём так)
void f(T& x);          ← хотим менять исходный объект в ф-ции
void f(const T& x);    ← не копируем и не меняем объект
```

```
void f(const T& x);
f("abc");
const string& s = "abc";
```

Lifetime prolongation

временный объект "умрёт" только, когда исходная ссылка "умрёт" (выйдет из своей области видимости)

Почему этого нет с обычными ссылками?

```
int x = 5;
const double& y = x;   ← тут создастся временный объект типа
                         double из x
double& z = x;         ← CE создастся временный объект, а
                         не const ссылки нельзя инициализировать
void swap(int& , int&);  временные объекты
```

```
double x = 2.0, y = 5.0;
swap(x, y);
```

← Если бы можно было, то пусть бы мы меняли временный объект а не исходный

```
void f(T& x);
void f(const T& x);

void f(T x)
```

Перегрузка разрешена
← для константных и временных объектов попадаем сюда
← Эта перегрузка не разрешена с такими ссылками

2.7 Type conversions

1. `static_cast< >( )` — только если объект можно кастовать (явное приведение). int → double или с помощью оператора приведения к типу
2. `reinterpret_cast< >( )` — „возьми биты одного типа и почитай их как другой типа" Если не совпадают размеры, то UB
3. `const_cast< >( )` — „снимает" константность (или навешивает)
4. `( )x` — „Си-style" каст:
   Сначала пытается static_cast< >,
   потом static_cast< const_cast< >>,
   затем reinterpret_cast
   затем reinterpret_cast< const_cast< >>,
   и только потом CE

Си-style cast по code style запрещён категорически.

5. `dynamic_cast< >( )` — изучим на теме наследования