Лекции

https://github.com/mtrempoltsev/msu_cpp_lectures
(https://github.com/mtrempoltsev/msu_cpp_lectures)

Рекомендуемая литература

Начальный уровень

Брюс Эккель, Философия C++

Книга старая, но довольно основательная.

Продвинутый уровень

1. Стивен Дьюхерст, C++. Священные знания
2. Скотт Мейерс, смело можно читать все
3. Герб Саттер, аналогично

Из относительно свежего

Препроцессор, компилятор, компоновщик

Процесс трансляции исходного кода в виде текстового файла в представление, которое может быть выполнено процессором - сборка.

Состоит из 3 этапов:

1. Обработка исходного кода препроцессором (preprocessing)

2. Компиляция, то есть перевод подготовленного исходного кода в инструкции процессора (объектный файл) (compiling)
3. Компоновка - сборка одного или нескольких объектных файлов в один исполняемый файл (linking)

square.cpp

```
int square(int value)
{
    return value * value;
}
```

Это файл с исходным кодом, он содержит определения функций.

Компилируются cpp/c/etc файлы, один файл с исходным кодом - один объектный файл. **Это называется единица трансляции.**

> Удобный инструмент: https://godbolt.org (https://godbolt.org)

```
g++ -c square.cpp
```

Вывод:

```
square.o
```

```
objdump -d square.o
```

```
square.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_Z6squarei>:
   0:  55                      push   %rbp
   1:  48 89 e5                mov    %rsp,%rbp
   4:  89 7d fc                mov    %edi,-0x4(%rbp)
   7:  8b 45 fc                mov    -0x4(%rbp),%eax
   a:  0f af 45 fc             imul   -0x4(%rbp),%eax
   e:  5d                      pop    %rbp
   f:  c3                      retq
```

Секции

Блоки данных в откомпилированном файле. Это может быть:

- Код (.text)
- Статические данные (.data)
- Таблицы строк
- Таблицы символов (.symtab)

Символы

То, что находится в объектном файле - кортежи из имени, адреса и свойств:

- Имя - произвольная строка
- Адрес - число (смещение, адрес)
- Свойства

Декорирование (mangling)

В C++ есть перегрузка функций (а еще есть классы), поэтому нужен механизм, чтобы различать перегруженные функции.

```
void print(int value); // _Z5printi
void print(const char* value); // _Z5printPKc
```

Инструмент для обратного преобразования:

```
c++filt _Z5printPKc
```

```
print(char const*)
```

extern "C"

```
extern "C"
{
    void print(int value); // print
}
```

main.cpp

```
int square(int value);

int main()
{
    return square(2);
}
```

```
objdump -d -r main.o
```

```
0000000000000000 <main>:
   0:  55                      push   %rbp
   1:  48 89 e5                mov    %rsp,%rbp
   4:  bf 02 00 00 00          mov    $0x2,%edi
   9:  e8 00 00 00 00          callq  e <main+0xe>
      a: R_X86_64_PC32  _Z6squarei-0x4
   e:  5d                      pop    %rbp
   f:  c3                      retq
```

```
-r - информация о релокациях
```

Символы

Символ - кортеж из имени, адреса и свойств:

- Имя - произвольная строка

- Адрес - число (смещение, адрес)
- Свойства

Связывание (binding) - говорит о том, виден ли символ вне файла:

- Локальный символ
- Глобальный символ
- Внешний символ

Смотрим таблицу символов:

```
objdump -t square.o
```

Вывод:

```
square.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 square.cpp
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss  0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment  0000000000000000 .comment
0000000000000000 g     F .text  0000000000000010 _Z6squarei
```

Первая колонка - связывание:

- l - локальное
- g - глобальное
- пробел - ни один из вариантов

Седьмая колонка - тип, если стоит F - значит это функция.

```
readelf -s square.o
```

```
Symbol table '.symtab' contains 9 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS square.cpp
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     8: 0000000000000000    16 FUNC    GLOBAL DEFAULT    1 _Z6squarei
```

```
objdump -t main.o
```

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.cpp
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss  0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment  0000000000000000 .comment
0000000000000000 g     F .text  0000000000000010 main
0000000000000000         *UND*  0000000000000000 _Z6squarei
```

```
readelf -s main.o
```

```
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.cpp
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000    16 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z6squarei
```

main.cpp

```
int square(int value);

int main()
{
    return square(2);
}
```

square.h

```
int square(int value);
```

Это заголовочный файл, как правило в нем находятся объявления типов и функций.

main.cpp

```
#include "square.h"

int main()
{
    return square(2);
}
```

Препроцессор

```
g++ -E main.cpp
```

Вывод:

```
# 1 "main.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.cpp"
# 1 "square.h" 1
int square(int value);
# 2 "main.cpp" 2

int main()
{
    return square(2);
}
```

Директивы препроцессора:

- `#include "name"` - целиком вставляет файл с именем name, вставляемый файл также обрабатывается препроцессором. Поиск файла происходит в директории с файлом, из которого происходит включение
- `#include <name>` - аналогично предыдущей директиве, но поиск производится в глобальных директориях и директориях, указанных с помощью ключа `-I`
- `#define x y` - вместо x подставляет y

> define - это опасно

```
#define true false // happy debugging
#define true !!(rand() % 2)
```

Условная компиляция

```
g++ -DDEBUG main.cpp
```

```
#define DEBUG
#ifdef DEBUG
    ...
#else
    ...
#endif
```

Компиляция

```
g++ -c main.cpp
```

В результате мы имеем 2 файла:

- main.o
- square.o

Компоновка

```
g++ -o my_prog main.o square.o
```

Вывод:

```
my_prog
```

```
./my_prog
echo $?
4
```

Компоновщик собирает из одного и более объектных файлов исполняемый файл.

Что g++ делает под капотом

```
g++ -o my_prog -v main.cpp square.cpp
```

Вывод:

```
...

/usr/lib/gcc/x86_64-linux-gnu/5/cc1plus
    main.cpp -o /tmp/ccjBvzkg.s

...

as -v --64 -o /tmp/ccM2mLyf.o /tmp/ccjBvzkg.s

...

/usr/lib/gcc/x86_64-linux-gnu/5/cc1plus
    square.cpp -o /tmp/ccjBvzkg.s

...

as -v --64 -o /tmp/cc3ZpAQe.o /tmp/ccjBvzkg.s

...

/usr/lib/gcc/x86_64-linux-gnu/5/collect2
    /tmp/ccM2mLyf.o /tmp/cc3ZpAQe.o
    -lstdc++ -lm -lgcc_s -lgcc -lc -lgcc_s -lgcc
```

Оптимизация

main.cpp

```
int square(int value)
{
    return value * value;
}

int main()
{
    return square(2);
}
```

```
g++ -c main.cpp
objdump -d main.o
```

```
0000000000000010 <main>:
  10:  55                      push   %rbp
  11:  48 89 e5                mov    %rsp,%rbp
  14:  bf 02 00 00 00          mov    $0x2,%edi
  19:  e8 00 00 00 00          callq  1e <main+0xe>
  1e:  5d                      pop    %rbp
  1f:  c3                      retq
```

```
g++ -O2 -c main.cpp
objdump -d main.o
```

```
0000000000000000 <main>:
   0:  b8 04 00 00 00          mov    $0x4,%eax
   5:  c3                      retq
```

Статические библиотеки

```
ar rc libsquare.a squre.o
```

Вывод:

```
libsquare.a
```

> В unix принято, что статические библиотеки имеют префикс lib и расширение .a

```
g++ -o my_prog main.o -L. -lsquare
```

`-L` - путь в котором компоновщик будет искать библиотеки
`-l` - имя библиотеки

> Статические библиотеки нужны только при сборке

Ошибки при сборке

1. Компиляции
2. Компоновки

Ошибки компоновки

Компоновщик не может найти символ

```
g++ -c math.cpp
g++ -o my_prog main.o
```

```
main.o: In function `main':
main.cpp:(.text+0xa): undefined reference to `square(int)'
collect2: error: ld returned 1 exit status
```

Что делать?

Включить необходимый файл в сборку, если нет определения символа - написать его, проверить, что файлы созданы одинаковой версией компилятора и с одними опциями компиляции.

Символ встретился несколько раз - компоновщик не знает какую версию выбрать

math.cpp

```
int square(int value)
{
    return value * value;
}
```

```
g++ -c math.cpp
g++ -o my_prog main.o square.o math.o
```

```
math.o: In function `square(int)':
math.cpp:(.text+0x0): multiple definition of `square(int)'
square.o:square.cpp:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Что делать?

Убрать неоднозначность: переименовать одну из функций, поместить в другое пространство имен, изменить видимость и т.д.

make

Утилита для автоматизации.

Синтаксис:

```
цель: зависимости
[tab] команда
```

Скрипт как правило находится в файле с именем **Makefile**.

Вызов:

```
make цель
```

Цель all вызывается, если явно не указать цель:

```
make
```

Плохой вариант

Makefile

```
CC=g++

all: my_prog

my_prog: main.cpp square.cpp square.h
    $(CC) -o my_prog main.cpp square.cpp

clean:
    rm -rf *.o my_prog
```

Хороший вариант

Makefile

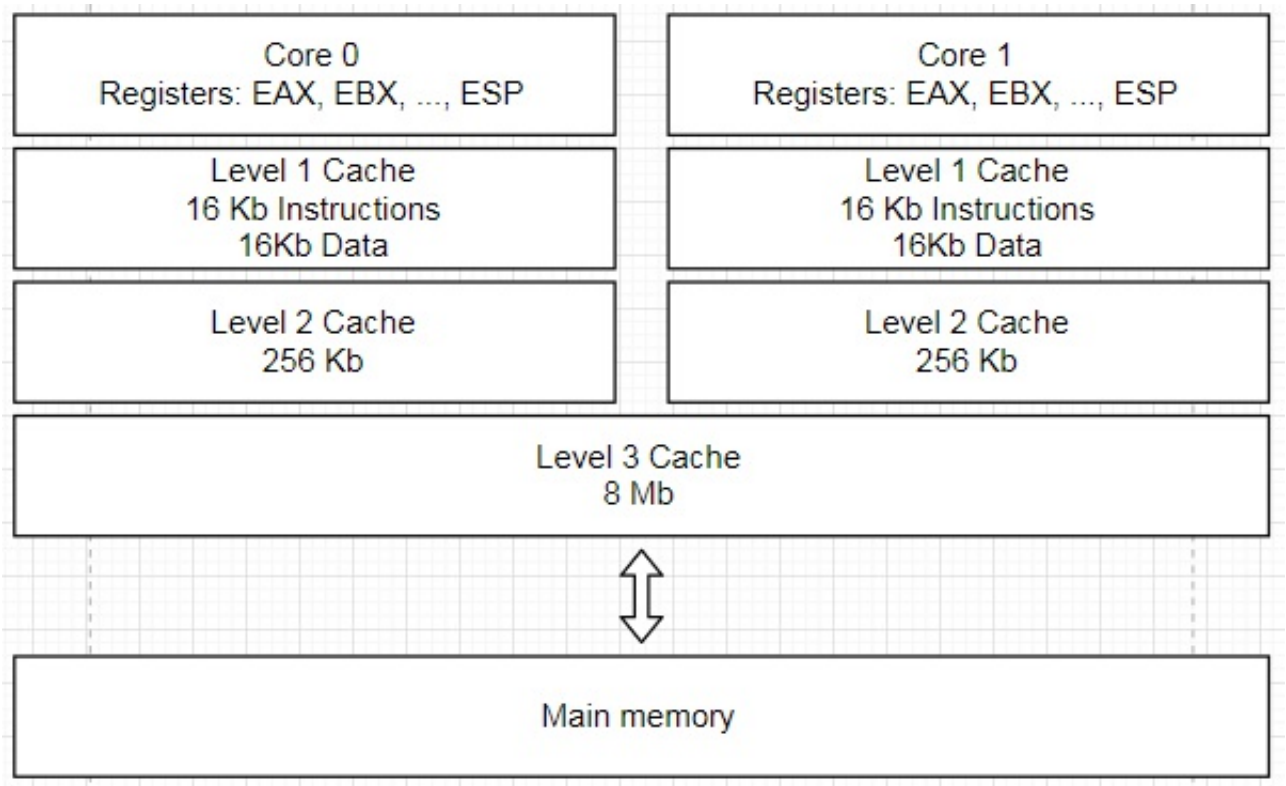
```
CC=g++

all: my_prog

my_prog: main.o square.o
    $(CC) -o my_prog main.o square.o

main.o: main.cpp square.h
    $(CC) -c main.cpp

square.o: square.cpp square.h
    $(CC) -c square.cpp

clean:
    rm -rf *.o my_prog
```

EOF

Память в C++

Кеш, оперативная память, стек и куча, выделение и освобождение памяти

Процессор



Линейное представление памяти

Адрес	Значение (1 байт)
0x0000	...
...	...
0x1000	1
0x1001	2
0x1002	3
0x1003	4
...	...
0xffffffffff	...

Арифметика указателей

```cpp
// Просто хранит какой-то адрес
void* addr = 0x1000;

// Если указатель никуда не ссылается,
// надо использовать nullptr
void* invalid = nullptr;

// Размер указателя, например, 4 - это количество
// байт необходимых для размещения адреса
size_t size = sizeof(addr); // size == 4

// Теперь мы говорим компилятору как
// интерпретировать то, на что указывет
// указатель
char* charPtr = (char*) 0x1000;

// Разыменование - получение значения, находящегося
// по указанному адресу
char c = *charPtr; // c == 1

// & - взятие адреса, теперь в charPtrPtr находится
// адрес charPtr
char** charPtrPtr = &charPtr;

int* intPtr = (int*) addr;
int i = *intPtr; // i == 0x04030201 (little endian)

int* i1 = intPtr;
int* i2 = i1 + 2;

ptrdiff_t d1 = i2 - i1; // d1 == 2

char* c1 = (char*) i1;
char* c2 = (char*) i2;

ptrdiff_t d2 = c2 - c1; // d2 == 8
```

```cpp
T* + n -> T* + sizeof(T) * n
T* - n -> T* - sizeof(T) * n
```

> C-cast использовать в C++ нельзя! Как надо приводить типы в C++ и надо ли вообще будет в другой лекции

Целочисленные типы

Знаковые	Беззнаковые
char	unsigned char
short	unsigned short
int	unsigned или unsigned int
long	unsigned long

> Стандарт не регламентирует размер типов

```
#include <cstdint>
```

Размер, бит	Тип
8	int8_t, int_fast8_t, int_least8_t
16	int16_t, int_fast16_t, int_least16_t
32	int32_t, int_fast32_t, int_least32_t
64	int64_t, int_fast64_t, int_least64_t

Беззнаковая (unsigned) версия - добавление префикса u

mem.cpp

```
#include <iostream>
#include <cstdint>

int global = 0;

int main()
{
    int* heap = (int*) malloc(sizeof(int));

    std::cout << std::hex << (uint64_t) main << '\n';
    std::cout << std::hex << (uint64_t) &global << '\n';
    std::cout << std::hex << (uint64_t) heap << '\n';
    std::cout << std::hex << (uint64_t) &heap << '\n';

    char c;
    std::cin >> c;
    return 0;
}
```

```
g++ -O0 mem.cpp -o mem --std=c++11
./mem
```

```
400986
6022b4
18adc20
7ffd5591e7d0
```

/proc/.../maps

```
ps ax | grep mem
```

```
00400000-00401000 r-xp 00000000 08:01 2362492
        /home/mt/work/tmp/mem
00601000-00602000 r--p 00001000 08:01 2362492
        /home/mt/work/tmp/mem
00602000-00603000 rw-p 00002000 08:01 2362492
        /home/mt/work/tmp/mem
0189c000-018ce000 rw-p 00000000 00:00 0
        [heap]
7f66aaa53000-7f66aabc5000 r-xp 00000000 08:01 6826866
        /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21
7f66aadc5000-7f66aadcf000 r--p 00172000 08:01 6826866
        /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21
7f66aadcf000-7f66aadd1000 rw-p 0017c000 08:01 6826866
        /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21
7ffd55900000-7ffd55921000 rw-p 00000000 00:00 0
        [stack]
7ffd55952000-7ffd55954000 r--p 00000000 00:00 0
        [vvar]
7ffd55954000-7ffd55956000 r-xp 00000000 00:00 0
        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
        [vsyscall]
```

Память разбита на сегменты:

- кода (CS)
- данных (DS)
- стека (SS)

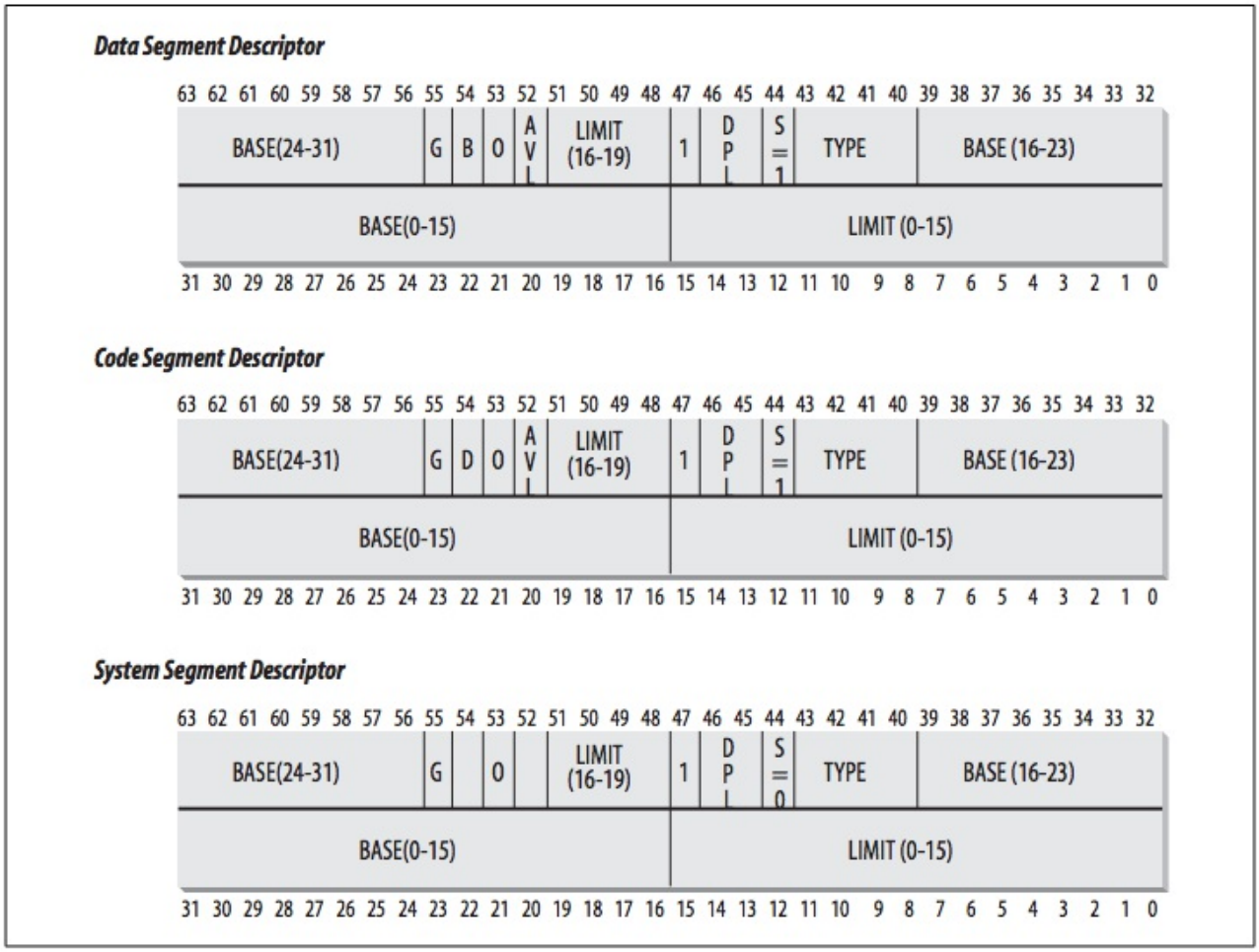
Регистр сегмента (CS, DS, SS) указывают на дескриптор.

Для инструкций и стека на смещение в сегменте указывает регистр:

- кода (EIP)
- стека (ESP)

Линейный адрес - это сумма базового адреса сегмента и смещения.

Дескриптор



Segment limit (20 bit) - размер сегмента, 55-й бит G определяет гранулярность размера:

- байты, если 0
- страницы, если 1 (размер страницы обычно 4Кб)

Бит 41-43:

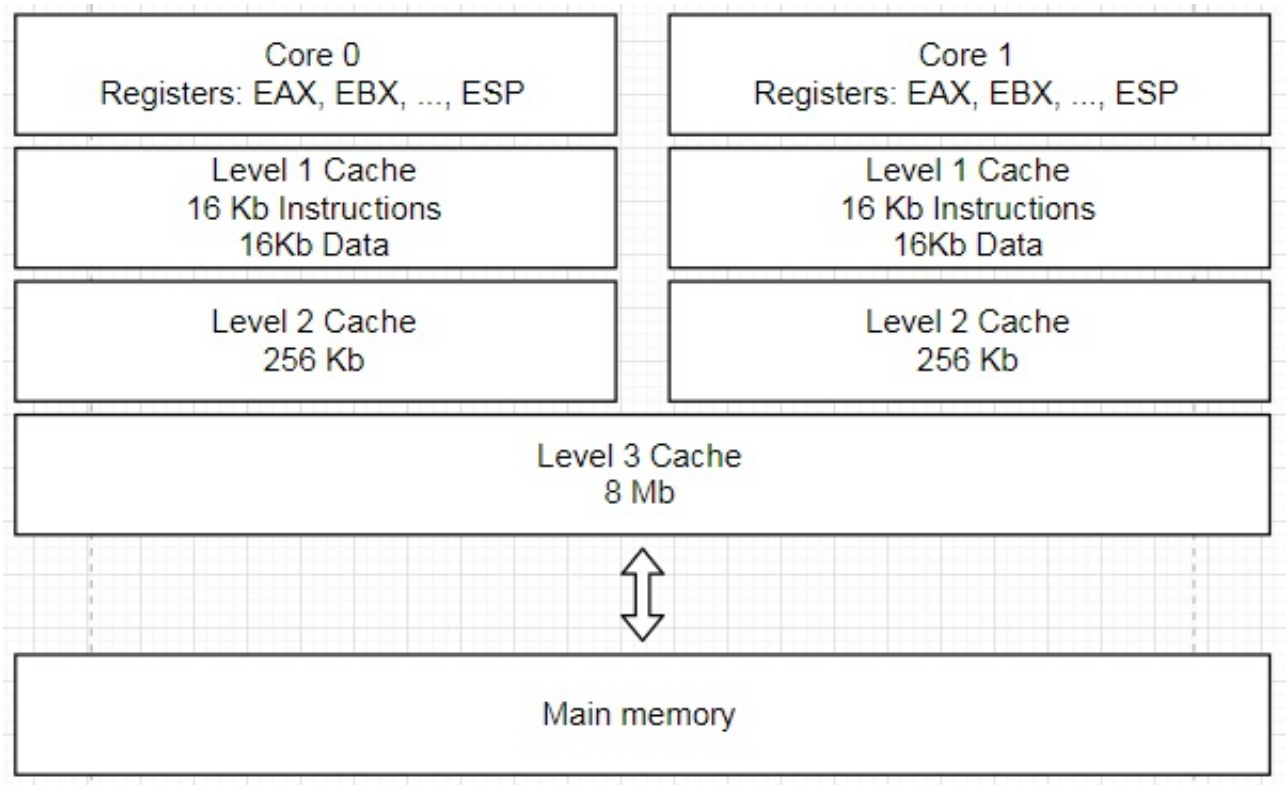
- 000 - сегмент данных, только чтение
- 001 - сегмент данных, чтение и запись
- 010 - сегмент стека, только чтение
- 011 - сегмент стека, чтение и запись
- 100 - сегмент кода, только выполнение
- 101- сегмент кода, чтение и выполнение

Виртуальная память

- Память делится на страницы
- Страница может находится в оперативной памяти или на внешнем носителе
- Трансляция из физического адреса в виртуальный и обратно выполняется через специальные таблицы: PGD (Page Global Directory), PMD (Page Middle Directory) и PTE (Page Table Entry). В PTE хранятся физические адреса страниц
- Для ускорения трансляции адресов процессор хранит в кеше таблицу TLB (Translation lookaside buffer)
- Если обращение к памяти не может быть оттранслировано через TLB, процессор обращается к таблицам страниц и пытается загрузить PTE оттуда в TLB. Если загрузка не удалась, процессор вызывает прерывание Page Fault

- Обработчик прерывания Page Fault находится в подсистеме виртуальной памяти ядра ОС и может загрузить требуемую страницу с внешнего носителя в оперативную память

Процессор



Важные константы

```
1 такт = 1 / частота процессора
1 / 3 GHz = 0.3 ns
                                          0.3 ns
L1 cache reference                        0.5 ns
Branch mispredict                         5   ns
```

> Неудачный if ()

```
L2 cache reference                        7   ns
Mutex lock/unlock                        25   ns
Main memory reference                   100   ns
```

> Кроме задержки (latency) есть понятие пропускной способности (throughput, bandwidth). В случае чтения из RAM - 10-50 Gb/sec

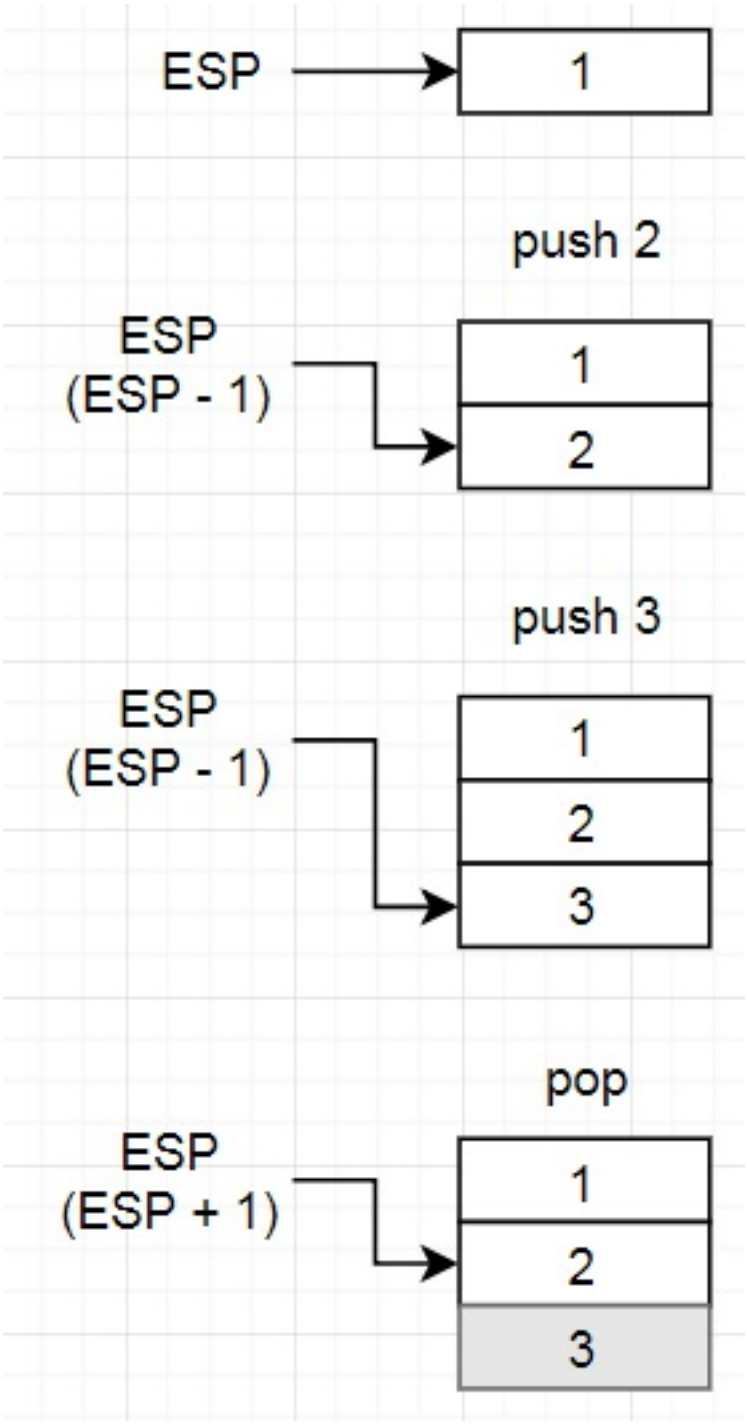
```
Compress 1K bytes with Zippy             3,000   ns
Send 1K bytes over 1 Gbps network       10,000   ns
Read 4K randomly from SSD              150,000   ns
Read 1 MB sequentially from memory     250,000   ns
Round trip within same datacenter      500,000   ns
Read 1 MB sequentially from SSD      1,000,000   ns
HDD seek                            10,000,000   ns
Read 1 MB sequentially from HDD     20,000,000   ns
Send packet CA->Netherlands->CA    150,000,000   ns
```

Источник: https://gist.github.com/jboner/2841832 (https://gist.github.com/jboner/2841832)

Выводы из таблицы

1. Стараться укладывать данные в кеш
2. Минимизировать скачки по памяти
3. В условиях основной веткой делать ветку которая выполняется с большей вероятностью

Стек



Классы управления памятью и областью видимости в C++

Характеризуются тремя понятиями:

1. **Время жизни**

Продолжительность хранения данных в памяти

2. **Область видимости**

Части кода из которых можно получить доступ к данным

3. **Связывание (linkage)**

> Если к данным можно обратиться из другой единицы трансляции — связывание внешнее (external), иначе связывание внутреннее (internal)

Автоматический/регистровый (register)

Время жизни	Область видимости	Связывание
Автоматическое (блок)	Блок	Отсутствует

```
{
    int i = 5;
}

if (true)
{
    register int j = 3;
}

for (int k = 0; k < 7; ++k)
{
}
```

Статический без связывания

Время жизни	Область видимости	Связывание
Статическое	Блок	Отсутствует

```
void foo()
{
    static int j = 3;
}
```

> Инициализируется при первом обращении

Статический с внутренним связыванием

Время жизни	Область видимости	Связывание
Статическое	Файл	Внутреннее

```
static int i = 5;
```

> Инициализируется до входа в main

Статический с внешним связыванием

Время жизни	Область видимости	Связывание

Статическое	Файл	Внешнее

```
// *.cpp
int i = 0;
```

```
// *.h
extern int i;
```

Типы памяти

Стек (Stack)

```
int i = 5;
std::string name;
char data[5];
```

> Выделение памяти на стеке очень быстрая, но стек не резиновый

Куча (Heap)

```
int* i = (int*) malloc(sizeof(int));
std::string* name = new std::string();
char* data = new char[5];
...
free(i);
delete(name);
delete[] data;
```

Память в куче выделяют new и malloc, есть сторонние менеджеры памяти.

Основное:

- new то же, что и malloc, только дополнительно вызывает конструкторы
- Внутри malloc есть буфер, если в буфере есть место, ваш вызов может выполниться быстро
- Если памяти в буфере нет, будет запрошена память у ОС (sbrk, VirtualAlloc) - это дорого
- ОС выделяет память страницами от 4Кб, а может быть и все 2Мб
- Стандартные аллокаторы универсальные, то есть должны быть потокобезопасны, быть одинаково эффективны для блоков разной длины, и 10 байт и 100Мб. Плата за универсальность - быстродействие

valgrind

```
#include <cstdlib>

int main()
{
    int* data = (int*) malloc(1024);
    return 0;
}
```

```
valgrind ./mem
```

```
==117392== Memcheck, a memory error detector
==117392== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==117392== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==117392== Command: ./mem
==117392==
==117392==
==117392== HEAP SUMMARY:
==117392==     in use at exit: 1,024 bytes in 1 blocks
==117392==   total heap usage: 1 allocs, 0 frees, 1,024 bytes allocated
==117392==
==117392== LEAK SUMMARY:
==117392==    definitely lost: 1,024 bytes in 1 blocks
==117392==    indirectly lost: 0 bytes in 0 blocks
==117392==      possibly lost: 0 bytes in 0 blocks
==117392==    still reachable: 0 bytes in 0 blocks
==117392==         suppressed: 0 bytes in 0 blocks
==117392== Rerun with --leak-check=full to see details of leaked memory
==117392==
==117392== For counts of detected and suppressed errors, rerun with: -v
==117392== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Глобальная память (data segment)

```
static const int i = 5;
static std::string name;
extern char data[5];
```

> Если не удастся разместить блок глобальной памяти, то программа даже не запустится

Массивы

```
T array[maxColumns];
T value = array[x];
```

> Значение в квадратных скобках должно быть известно на этапе компиляции, увы

```
int data[] = { 1, 2, 3 };
int i = data[2];
```

Фактически - это вычисление смещения:

```
ptr = data;
ptr = ptr + 2 * sizeof(int);
i = *ptr;
```

Массив - непрерывный блок байт в памяти, sizeof(data) вернет размер массива в байтах (не элементах!). Размер массива в элементах можно вычислить: sizeof(data) / sizeof(data[0])

```
int* data = new int[10];
int i = data[2];
delete[] data;
```

Массив <-> указатель

```
int i[] = { 1, 2, 3 };
int* j = i;
using array = int*;
array k = (array) j;
```

Двумерные массивы

```
T array[maxRows][maxColumns];
T value = array[y][x];
```

```
int data[][2] = { { 1, 2 },  { 3, 4 }, { 5, 6 } };
int i = data[2][1];
```

Фактически:

```
ptr = data;
ptr = ptr + maxColumns * sizeof(int) * 2 + 1;
i = *ptr;
```

Массив <-> указатель

```
int i[][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
int* j = (int*) i;
using matrix = int(*)[2];
matrix k = (matrix) j;
```

Измеряем скорость работы (benchmark)

1. Измерений должно быть много
2. Одному прогону верить нельзя
3. Компилятор оптимизирует, надо ему мешать
4. Перед тестами надо греться

Пример "вредной" оптимизации

```
int main()
{
    Timer t;
    for (int i = 0; i < 100 * 1000 * 1000; ++i)
        int a = i / 3;
    return 0;
}
```

Не даем компилятору оптимизировать

```
int main()
{
    Timer t;
    for (int i = 0; i < 100 * 1000 * 1000; ++i)
        volatile int a = i / 3;
    return 0;
}
```

Класс Timer

```
#include <chrono>
#include <iostream>

class Timer
{
    using clock_t = std::chrono::high_resolution_clock;
    using microseconds = std::chrono::microseconds;
public:
    Timer()
        : start_(clock_t::now())
    {
    }

    ~Timer()
    {
        const auto finish = clock_t::now();
        const auto us =
            std::chrono::duration_cast<microseconds>
                (finish - start_).count();
        std::cout << us << " us" << std::endl;
    }

private:
    const clock_t::time_point start_;
};
```

Практическая часть

Написать две программы, суммирующие элементы двумерного массива (матрицы) целых чисел. Одна программа суммирует по столбцам, вторая - по строкам. Измерить время работы в обоих случаях, сравнить результаты. Для замеров можно использовать класс Timer. В репозитории в директории homework создать директорию со своей фамилией, внутри этой директории создать директорию 01, файлы с решением положить внутрь. Программу компилировать с влюченной оптимизацией, например, так:

```
g++ sum_by_rows.cpp -o sum_by_rows --std=c++11 -O2
```

Оба решения можно запустить через valgrind и посмотреть количество промахов в кеш:

```
valgrind --tool=cachegrind your_program
```

Подумать.

EOF

Функции

```
int rollDice()
{
    return 4;
}

int square(int x)
{
    return x * x;
}
```

Конвенции вызова x32

cdecl

Исторически принятое соглашение для языка C.

Аргументы функций передаются через стек, справа налево. Аргументы, размер которых меньше 4-х байт, расширяются до 4-х байт. Очистку стека производит вызывающая программа. integer-like результат возвращается через регистр EAX.

Перед вызовом функции вставляется код, называемый прологом (prolog) и выполняющий следующие действия:

- сохранение значений регистров, используемых внутри функции
- запись в стек аргументов функции

После вызова функции вставляется код, называемый эпилогом (epilog) и выполняющий следующие действия:

- восстановление значений регистров, сохранённых кодом пролога
- очистка стека (от локальных переменных функции)

thiscall

Соглашение о вызовах, используемое компиляторами для языка C++ при вызове методов классов.

Отличается от **cdecl** соглашения только тем, что указатель на объект, для которого вызывается метод (указатель this), записывается в регистр ecx.

fastcall

Передача параметров через регистры: если для сохранения всех параметров и промежуточных результатов регистров не достаточно, используется стек (в gcc через регистры ecx и edx передаются первые 2 параметра).

Смотрим сгенерированный код

```
[[gnu::fastcall]]
void foo1(int x, int y, int z, int a)
{
}

void foo2(int x, int y, int z, int a)
{
}

void bar1()
{
    foo1(1, 2, 3, 4);
}

void bar2()
{
    foo2(5, 6, 7, 8);
}
```

```
g++ -c test.cpp -o test.o -O0 -m32
```

```
objdump -d test.o
```

```
000005c8 <_Z4bar1v>:
 5c8:   6a 04                   push   $0x4
 5ca:   6a 03                   push   $0x3
 5cc:   ba 02 00 00 00          mov    $0x2,%edx
 5d1:   b9 01 00 00 00          mov    $0x1,%ecx
 5d6:   e8 b5 ff ff ff          call   590 <_Z4foo1iiii>
 5dd:   c3                      ret

000005eb <_Z4bar2v>:
 5eb:   6a 08                   push   $0x8
 5ed:   6a 07                   push   $0x7
 5ef:   6a 06                   push   $0x6
 5f1:   6a 05                   push   $0x5
 5f3:   e8 b3 ff ff ff          call   5ab <_Z4foo2iiii>
 5fd:   c3                      ret
```

System V AMD64 ABI (Linux, MacOS, FreeBSD, Solaris)

- 6 регистров (RDI, RSI, RDX, RCX, R8, R9) для передачи integer-like аргументов
- 8 регистров (XMM0-XMM7) для передачи double/float
- если аргументов больше, они передаются через стек
- для возврата integer-like значений используются RAX и RDX (64 бит + 64 бит)

Встраивание функций (inline)

Иногда вызова функции не будет - оптимизирующий компилятор выполнит ее встраивание по месту вызова.

Можно подсказать компилятору выполнить встраивание:

```
inline void foo()
{
}
```

Но, компилятор умный и скорее всего проигнорирует inline, но можно попросить настойчивей:

```
// ms vc
__forceinline void foo()
{
}
```

```
// gcc
__attribute__((always_inline)) void foo()
{
}
```

Все равно нет гарантий.

Тот случай, когда макросы уместны

```
#ifdef __GNUC__
#define __forceinline __attribute__((always_inline))
#endif
```

Ссылки

Ссылка - псевдоним объекта.

Главное отличие от указателя - ссылка должна быть проинициализирована при объявлении и до конца своей жизни ссылается только на один объект.

```
int a = 1;
int b = 2;
int* ptr = nullptr;
ptr = &a;
ptr = &b;
int& ref; // Ошибка
int& ref = a; // ref ссылается на a
ref = 5; // Теперь a == 5
ref = b; // ref не стала указывать на b,
         // мы просто скопировали значение из b в a
ref = 7; // a == 7, b == 2
```

```
int a = 2;
int* ptr = nullptr;
int*& ptrRef = ptr; // ptrRef ссылается на ptr
ptrRef = &a; // теперь ptr хранит адрес a
```

Передача аргументов в функции

По значению

```
void foo(int x)
{
    x = 3;
}

int x = 1;
foo(x);
// x == 1

void bar(BigObject o) { ... }
```

- В функции окажется копия объекта, ее изменение не отразится на оригинальном объекте
- Копировать большие объекты может оказаться накладно

По ссылке

```
void foo(int& x)
{
    x = 3;
}

int x = 1;
foo(x);
// x == 3

void bar(BigObject& o) { ... }
```

- Копирования не происходит, все изменения объекта внутри функции отражаются на объекте
- Следует использовать, если надо изменить объект внутри функции

```
void swap(int& x, int& y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

По константной ссылке

```
void foo(const int& x)
{
    x = 3; // ошибка компиляции
}

void bar(const BigObject& o) { ... }
```

- Копирования не происходит, при попытке изменения объекта будет ошибка
- Большие объекты выгодней передавать по ссылке, маленькие - наоборот

По указателю

```
void foo(int* x)
{
    *x = 3;
}

void bar(BigObject* o) { ... }

void foo(const int* x)
{
    *x = 3; // ошибка компиляции
}

void bar(const BigObject* o) { ... }
```

- Копирования не происходит
- Если указатель на константный объект, то при попытке изменения объекта будет ошибка
- Есть дополнительный уровень косвенности, возможно придется что-то подгрузить в кеш из дальнего участка памяти
- Реализуется optional-концепция

```
int countObject(time_t* fromDate, time_t* toDate)
{
    const auto begin =
        fromDate == nullptr
            ? objects_.begin()
            : objects_.findFirst(fromDate);
}
```

По универсальной ссылке

```
void foo(int&& x) { ... }
void bar(BigObject&& o) { ... }
```

Поговорим в отдельной лекции.

Перегрузка функций

```
void print(int x) // 1
{
    std::cout << x << std::endl;
}

void print(bool x) // 2
{
    std::cout << (x ? "true" : "false") << std::endl;
}

print(10); // 1
print(true); // 2
```

Опасность перегрузки

```
void print(const std::string& x) // 3
{
    std::cout << "string" << std::endl;
}

print("hello!"); // 2 const char* приводится к bool
```

> Перегруженная функция - декорированная функция

Пространства имен

Проблема:

```
// math.h
double cos(double x);
```

```
// ваш код
double cos(double x);
```

Решение в стиле C:

```
// ваш код
double fastCos(double x);
```

Решение:

```
namespace fast
{
    double cos(double x);
}

fast::cos(x);
cos(x); // вызов из math.h
```

Поиск имен

- Проверка в текущем namespace
- Если имени нет и текущий namespace глобальный - ошибка
- Рекурсивно поиск в родительском namespace

```
void foo() {} // ::foo

namespace A
{
    void foo() {} // A::foo

    namespace B
    {
        void bar() // A::B::foo
        {
            foo(); // A::foo
            ::foo(); // foo()
        }
    }
}
```

Ключевое слово using

Добавляет имена из указанного namespace в текущий namespace.

```
void foo()
{
    using namespace A;
    // видимо все из A
}
```

```
void foo()
{
    using namespace A::foo;
    // видима только A::foo()
}
```

```
void foo()
{
    namespace ab = A::B;
    ab::bar(); // A::B::bar()
}
```

using может приводить к проблемам

```
using namespace fast;

cos(x); // ???
cos(x); // ???
```

> Не используйте using namespace в заголовочных файлах!

Указатель на функцию

```
void foo(int x)
{
}

typedef void (*FooPtr)(int);

FooPtr ptr = foo;
ptr(5);
```

```
using FooPtr = void(*)(int);
```

Функции высшего порядка

Функция высшего порядка — функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

Сценарии использования указателей на функции

Настройка поведения

```
void sort(int* data, size_t size, bool (*compare)(int x, int y));

bool less(int x, int y)
{
    retunr x < y;
}

sort(data, 100, less);
```

Фунции обратного вызова (callback)

```
using OnMailReceived = void (*)(const Mail& newMail);

void addOnMailReceivedObserver(OnMailReceived handler);

void onMailReceivedHandler(const Mail& newMail)
{
    ...
}

addOnMailReceivedObserver(onMailReceivedHandler);
```

Конвееры

```
using MoveFunctionPtr = void (*)(int& x, int& y);

void moveLeft(int& x, int& y) { ... }
void moveRight(int& x, int& y) { ... }

std::vector<MoveFunctionPtr> trajectory =
    {
        moveLeft,
        moveLeft,
        moveRight,
    };

int x = 0;
int y = 0;
for (auto& func : trajectory)
{
    func(x, y);
}
```

Лямбда-функции

```
auto lessThen3 = [](int x) { return x < 3; };

if (lessThen3(x)) { ... }
```

Синтаксис

```
[список_захвата](список_параметров) { тело_функции }
```

```
[список_захвата](список_параметров) -> тип_возвращаемого_значения
{ тело_функции }
```

Захват переменных

```
int x = 5;
int y = 7;
auto foo = [x, &y]() { y = 2 * x };
foo();
```

Если не указать &, то будет захват по значению, то есть копирование объекта; если указать, то по ссылке (нет копирования, можификации внутри функции отразяться на оригинальном объекте).

```
// Захват всех переменных в области видимости по значению
auto foo = [=]() {};
```

```
// Захват всех переменных в области видимости по ссылке
auto foo = [&]() {};
```

> Использование переменных, определённых в той же области видимости, что и лямбда-функция, называют замыканием.

Примеры захвата

```
[] // без захвата переменных из внешней области видимости
[=] // все переменные захватываются по значению
[&] // все переменные захватываются по ссылке
[x, y] // захват x и y по значению
[&x, &y] // захват x и y по ссылке
[in, &out] // захват in по значению, а out — по ссылке
[=, &out1, &out2] // захват всех переменных по значению,
// кроме out1 и out2, которые захватываются по ссылке
[&, x, &y] // захват всех переменных по ссылке, кроме x,
// которая захватывается по значению
```

mutable

```
int x = 3;
auto foo = [x]() mutable
{
    x += 3;
    ...
}
```

std::function

```
#include <functional>

using MoveFunction =
    std::function<void (int& x, int& y)>;

MoveFunction getRandomDirection() { ... }

std::vector<MoveFunction> trajectory =
    {
        moveLeft,
        moveLeft,
        moveRight,
        [](int& x, int& y)
        {
            ...
        },
        moveLeft,
        getRandomDirection()
    };

int x = 0;
int y = 0;
for (auto& func : trajectory)
{
    func(x, y);
}
```

Как выделить память в C++

```
char* data = (char*) malloc(1024);
...
free(data);
```

```
std::unique_ptr<char[]> data(new char[1024]);
// или так
auto data = std::make_unique<char[]>(1024);
```

```
char* ptr = data.get() // Указатель
char x = data[100]; // Элемент 100
data.reset(); // Явное освобождение памяти
```

Требования к домашним заданиям

1. В вашем github должен быть репозиторий msu_cpp_autumn_2019
2. Внутри репозитория должны быть директории из двух цифр, вида: 01, 02 и т.д. - это номера домашних заданий
3. Внутри каждой директории могут быть любые файлы реализующие задачу. Обязательным является только файл Makefile
4. В Makefile обязательно должны быть цели test и run. run запускает бинарный файл с домашним заданием, test запускает тесты вашего решения
5. Собираться ваш код должен компилятором C++ поддерживающим 14 стандарт
6. Внешних зависимостей быть не должно

После выполнения домашнего задания присылайте Владиславу Смирнову (vladislav.smirnov@corp.mail.ru) письмо с темой made_2019_cpp_номердз

Практическая часть

Используя метод рекурсивного спуска, написать калькулятор. Поддерживаемые операции:

- умножение
- деление
- сложение
- вычитание
- унарный минус

Для вычислений использовать тип int, приоритет операций стандартный. Передача выражения осуществляется через аргумент командной строки, поступаемые числа целые, результат выводится в cout. Пример:

```
calc "2 + 3 * 4 - -2"
```

Вывод:

```
16
```

Должна быть обработка ошибок, в случае некорректного выражения выводить в консоль ошибку и возвращать код отличный от 0. Тесты обязательны.

EOF

Структуры и классы

Информация о пользователе:

1. Имя
2. email

```
std::string name;
std::string email;
```

Агрегируем данные

Для упрощения программы, логически связанные данные можно объединить.

```
struct User
{
    std::string name;
    std::string email;
};

const User user =
    { "Bob", "bob@mail.ru" };

std::cout << user.name;
```

> name, email - поля структуры

Много пользователей (array of structs)

```
User users[N];
```

Много пользователей (struct of arrays)

```
struct Users
{
    std::string name[N];
    std::string email[N];
};
```

Модификаторы доступа

```
struct A
{
public:
    int x; // Доступно всем
protected:
    int y; // Наследникам и объектам класса
private:
    int z; // Только объектам класса
};

A a;
a.x = 1; // ок
a.y = 1; // ошибка
a.z = 1; // ошибка
```

> Объект - сущность в адресном пространстве компьютера, появляющаяся при создании класса.

struct vs class

В C++ struct от class отличаются только модификатором доступа по умолчанию. По умолчанию содержимое struct доступно извне (public), а содержимое class - нет (private).

```
class A
{
    int x; // private
};

struct B
{
    int x; // public
}
```

Методы класса

```
struct User
{
    void serialize(Stream& out)
    {
        out.write(name);
        out.write(email);
    }

private:
    std::string name;
    std::string email;
};
```

> serialize - метод класса

> Методы класса, доступные для использования другими классами, представляют его интерфейс

Классы в C

```
struct File
{
    int descriptor;
    char buffer[BufferSize];
};

File* openFile(const char* fileName)
{
    File* file = (File*) malloc(sizeof(File));
    file->descriptor = open(fileName, O_CREAT);
    return file;
}

void write(File* file, const char* data, size_t size)
{
    ...
}

void close(File* file)
{
    close(file->descriptor);
    free(file);
}

File* file = openFile("some_file.dat");
write(file, data, size);
close(file);
```

```
class File
{
public:
    File(const char* fileName)
    {
        descriptor = open(fileName, O_CREAT);
    }

    void write(const char* data, size_t size)
    {
        ...
    }

    ~File()
    {
        close(descriptor);
    }

private:
    int descriptor;
    char buffer[BufferSize];
};

File file("some_file.dat");
file.write(data, size);
```

Декорирование методов класса

```
struct A
{
    void foo(); // _ZN1A3fooEv
};

void bar(); // _Z3barv
```

Указатель на экземпляр класса

```
void write([File* this], const char* data, size_t size)
{
    this->descriptor ...
}
```

> Метод класса - обычная функция, которая неявно получает указатель на объект класса (this)

```
struct A
{
    void foo() { std::cout << "ok"; }
    void bar() { std::cout << x; }

    int x;
};

A* a = nullptr;
a->foo(); // Ок
a->bar(); // Разыменование нулевого указателя
```

```
void foo([A* this])
{
    std::cout << "ok";
}

void bar([A* this])
{
    std::cout << [this->]x;
}
```

Конструктор (ctor)

Служит для инициализации объекта.

Если конструктор не написан явно, C++ гарантирует, что будет создан конструктор по умолчанию.

```
struct A
{
    A() {}
};
```

Конструктор вызывается автоматически при создании объекта

```
// Выделение памяти в куче + вызов конструктора
A* x = new A();

// Выделение памяти на стеке + вызов конструктора
A y;
```

Деструктор (dtor)

Если деструктор не написан явно, C++ гарантирует, что будет создан деструктор по умолчанию.

```
struct A
{
    ~A() {}
};
```

Служит для деинициализации объекта, **гарантированно вызывается при удалении объекта**.

```
{
    A* x = new A();
    A y;
} // Выход из области видимости:
  // вызов деструктора + освобождение
  // памяти на стеке
  // Для x это означает, что
  // будет освобождена только память
  // занятая указателем, но та,
  // на которую он указывает
```

```
{
    A* x = new A();
    A y;
    delete x;
}
```

RAII (Resource Acquire Is Initialization)

Захват ресурса есть инициализация.

В конструкторе объект получает доступ к какому либо ресурсу (например, открывается файл), а при вызове деструктура этот ресурс освобождается (закрывается файл).

```
class File
{
public:
    File(const char* fileName)
    {
        descriptor = open(fileName, O_CREAT);
    }

    ~File()
    {
        close(descriptor);
    }
};
```

> Можно использовать не только для управления ресурсами

```
struct Profiler
{
    Profiler() { // получаем текущее время }
    ~Profiler() { // сохраняем время между
        // выховами конструктора и деструктора }
};

void someFunction()
{
    Profiler p;
    if (...) return;
    ...
    if (...) return;
    ...
}
```

Константные методы

```
struct A
{
    int x;
};

A a;
a.x = 3; // Ок

const A b;
b.x = 3; // Ошибка, константный
         // объект нельзя изменять

const A* c = &a;
c->x = 3; // Ошибка, константный
         // объект нельзя изменять
```

Любые методы кроме конструктора и деструктора могут быть константными.

```
class User
{
    using Year = uint32_t;
    Year age;
public:
    void setAge(Year value)
    {
        age = value;
    }

    bool canBuyAlcohol() const
    {
        return age >= 21;
    }
};

class UserDb
{
public:
    const User* getReadOnlyUser(
        const std::string& name) const
    {
        return db.find(name);
    }
};

const User* user = userDb.getReadOnlyUser("Bob");
user->setAge(21); // Ошибка
if (user->canBuyAlcohol()) // Ок
```

```
void User_setAge([User* const this], Year value)
{
    [this->]age = value;
}

bool User_canBuyAlcohol([const User* const this]) const
{
    return [this->]age >= 21;
}
```

mutable

```
class Log
{
    void write(const std::string& text);
};

class UserDb
{
    mutable Log log;
public:
    const User& getReadOnlyUser(
        const std::string& name) const
    {
        log.write("...");
        return db.find(name);
    }
};
```

```
const User& UserDb_getReadOnlyUser(
    [const UserDb* const this],
    const std::string& name) const
{
    [this->]log.write("...");
    // Вызываем Log_write с const Log* const
}

void Log_write([Log* const this], const std::string& text)
{
    ...
}
```

Наследование

Возможность порождать класс на основе другого с сохранением всех свойств класса-предка.

Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс – потомком, наследником, дочерним или производным классом.

```
class Shape
{
protected:
    int x;
    int y;
};

class Circle
    : public Shape
{
    int radius;
};
```

> Наследование моделирует отношение «является».

> Требуется для создания иерархичности – свойства реального мира.

Представление в памяти при наследовании

Инструменты для исследования

- **sizeof(T)** - размер типа в байтах
- **offsetof(T, M)** - смещение поля M от начала типа T

```
struct A
{
    double x;
};

struct B
    : public A
{
    double y;
};

struct C
    : public B
{
    double z;
};

std::cout << sizeof(A) << std::endl; // 8
std::cout << sizeof(B) << std::endl; // 16
std::cout << sizeof(C) << std::endl; // 24

std::cout << offsetof(C, x) << std::endl; // 0
std::cout << offsetof(C, y) << std::endl; // 8
std::cout << offsetof(C, z) << std::endl; // 16
```

Поле	Смещение	Доступность в типах
x	0	A, B, C
y	8	B, C
z	16	C

```
C* c = new C();
c->x; // Ок
c->y; // Ок
c->z; // Ок

B* b = (B*) c;
b->x; // Ок
b->y; // Ок
b->z; // Ошибка компиляции

A* a = (A*) c;
a->x; // Ок
a->y; // Ошибка компиляции
a->z; // Ошибка компиляции
```

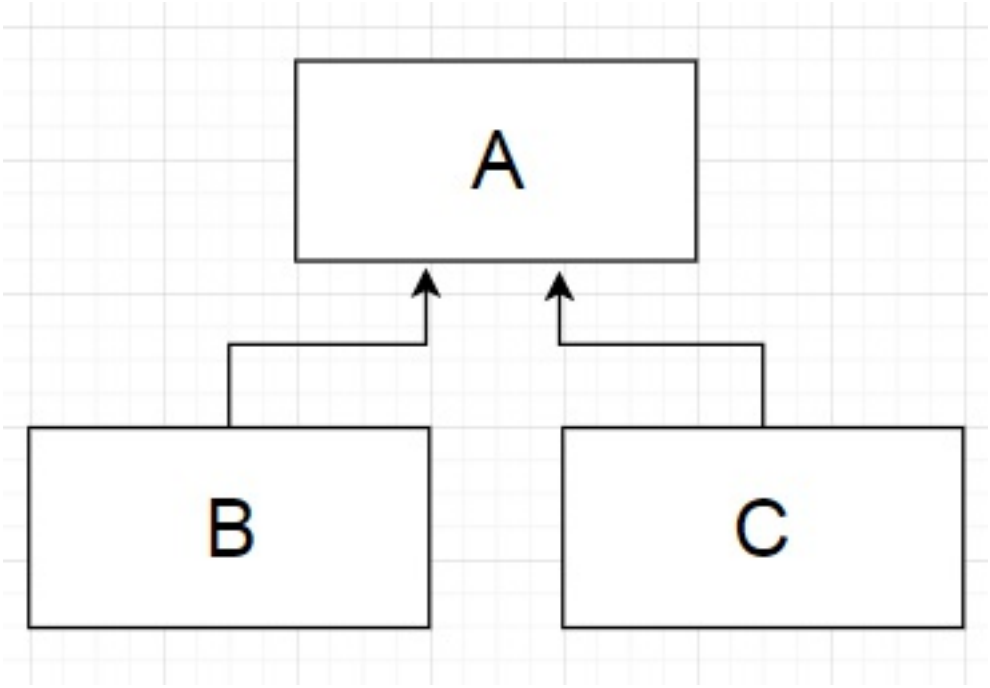
Приведение вверх и вниз по иерархии

Приведение вверх (к базовому классу) всегда безопасно.

```
void foo(A& a) {}

C c;
foo(c);
```

Приведение вниз может быть опасным

```
struct A {};
struct B : public A {};
struct C : public A {};
```



```
B* b = new B();
A* a = b;
C* c = a; // Ошибка компиляции
C* c = static_cast<C*>(b); // Ошибка компиляции
C* c = static_cast<C*>(a); // !!!
```

> Сохраняйте тип, пусть компилятор помогает писать корректный код!

> Общий базовый тип - плохая идея

Композиция

```
class Car
{
    Engine engine;
    Wheels wheels[4];
};
```

> Композиция моделирует отношение «содержит/является частью»

Агрегация

```
class Car
{
    Driver* driver_;
};
```

При агрегации класс не контролирует время жизни своей части.

Унифицированный язык моделирования (Unified Modeling Language, UML)

UML – это открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML используется для визуализации и документирования программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

UML редактор: https://www.draw.io/ (https://www.draw.io/)

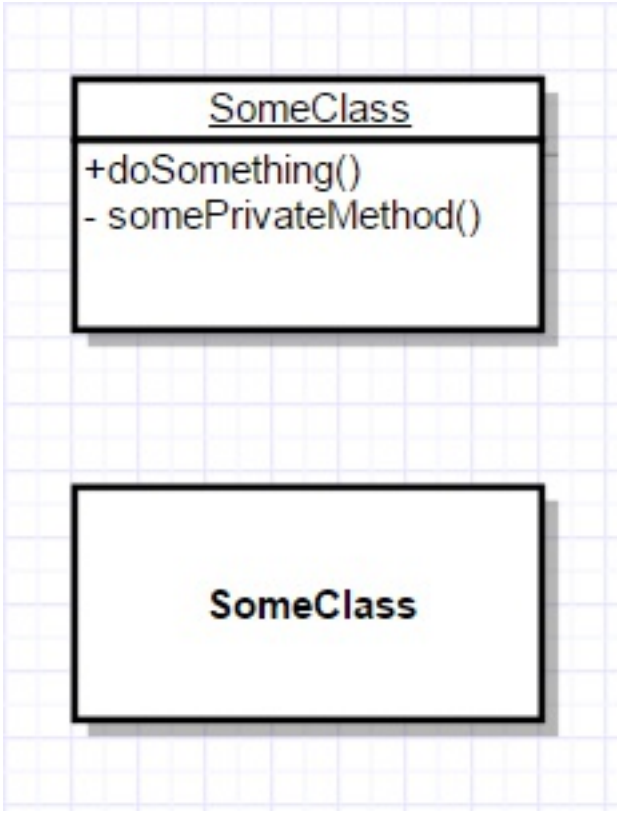
Диаграмма классов (Class diagram)

Статическая структурная диаграмма, описывающая структуру системы, демонстрирующая классы системы, их атрибуты, методы и зависимости между классами.

Классы

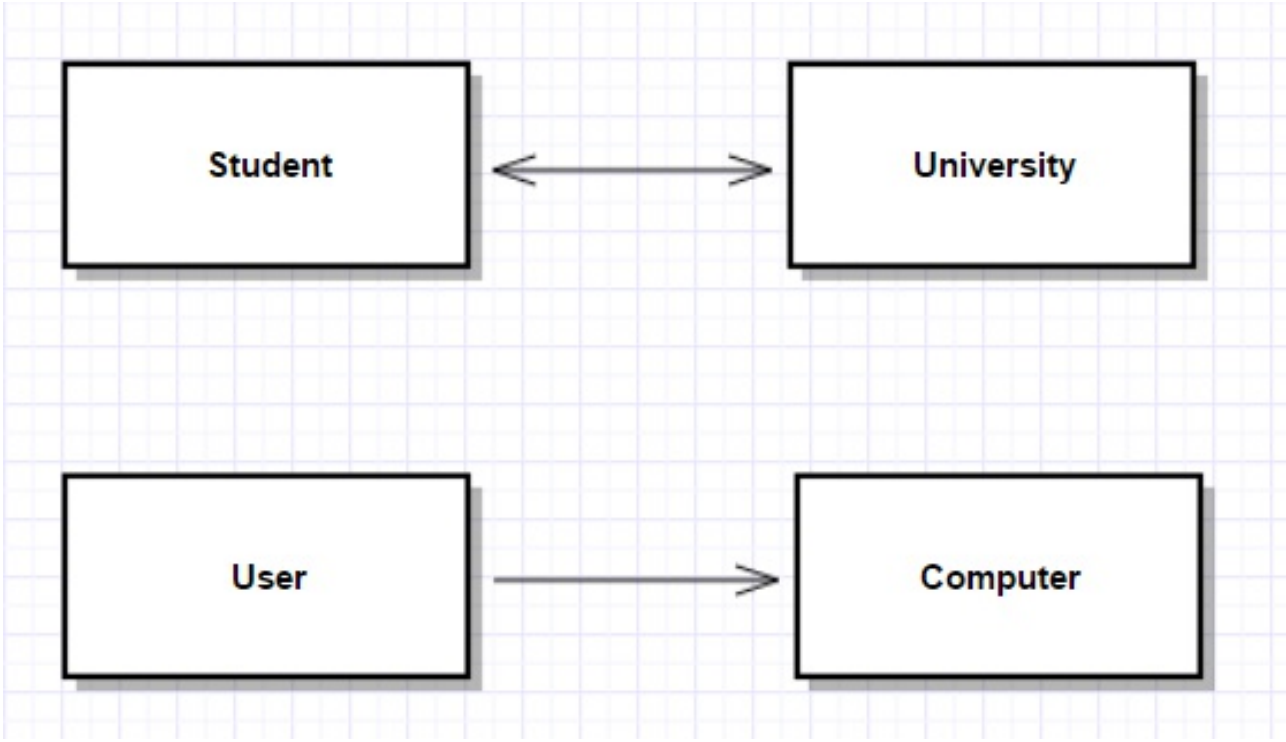
Видимость:

```
+ Публичный метод (public)
# Защищенный метод (protected)
- Приватный метод (private)
```



Ассоциация

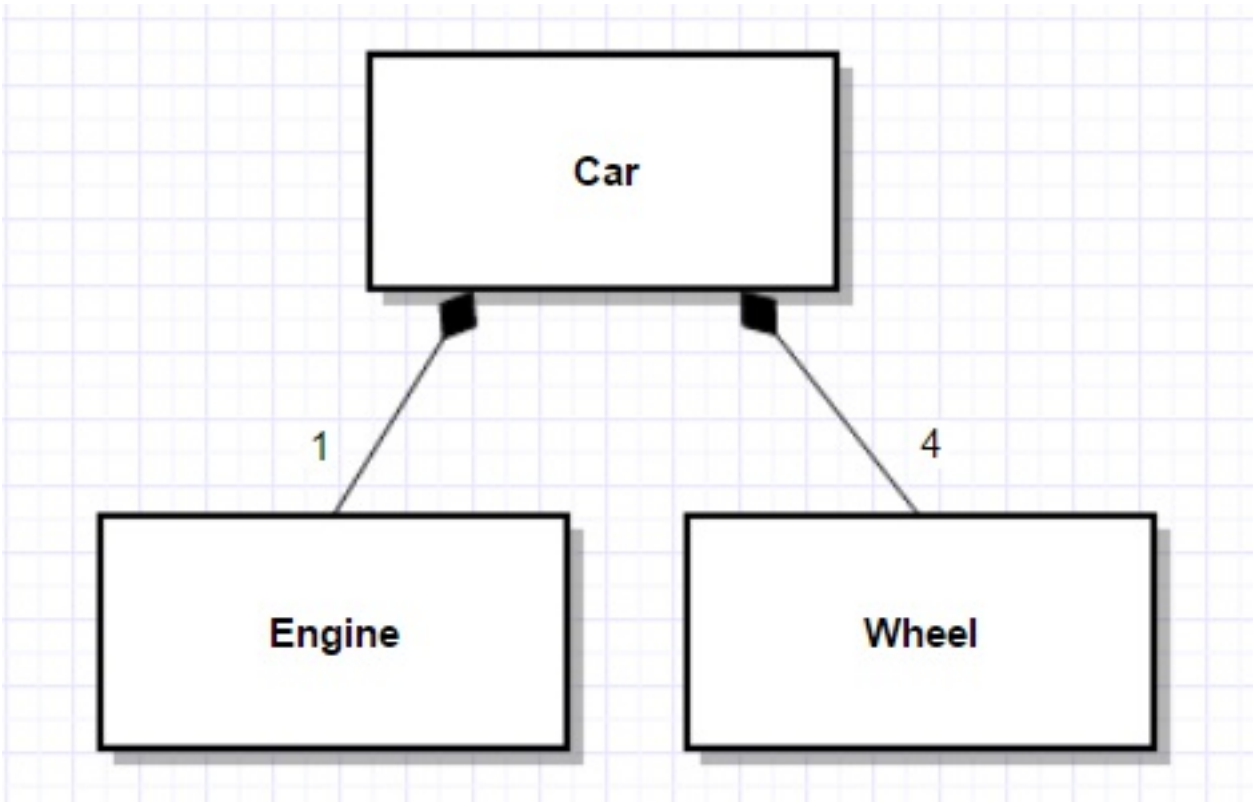
Показывает, что объекты связаны, бывает однонаправленной и двунаправленной.



Композиция

Моделирует отношение «содержит/является частью».

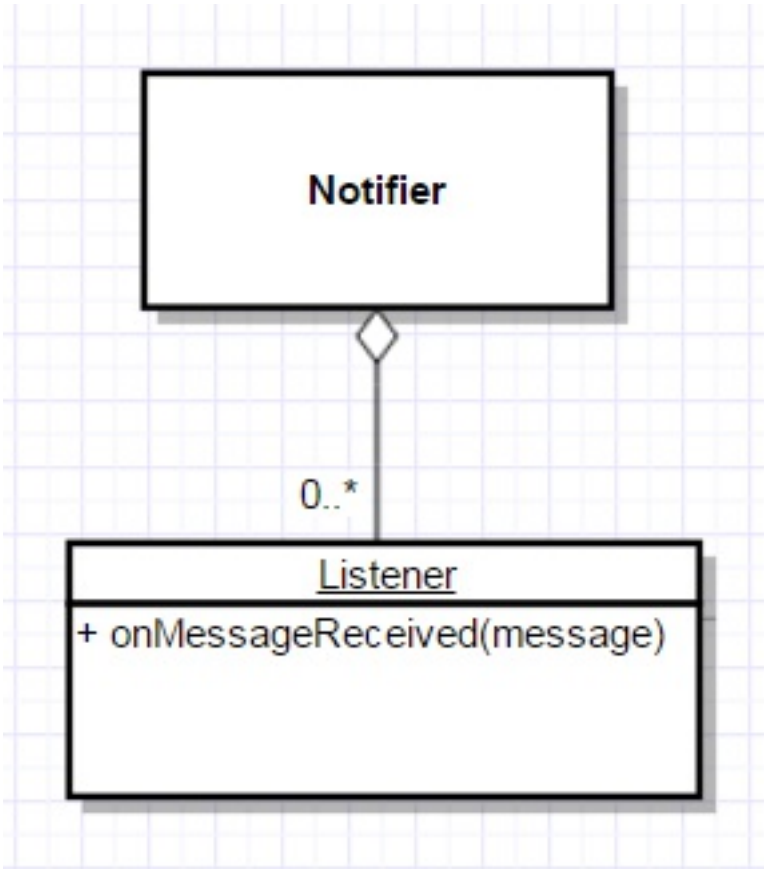
При композиции класс явно контролирует время жизни своей составной части.



Агрегация

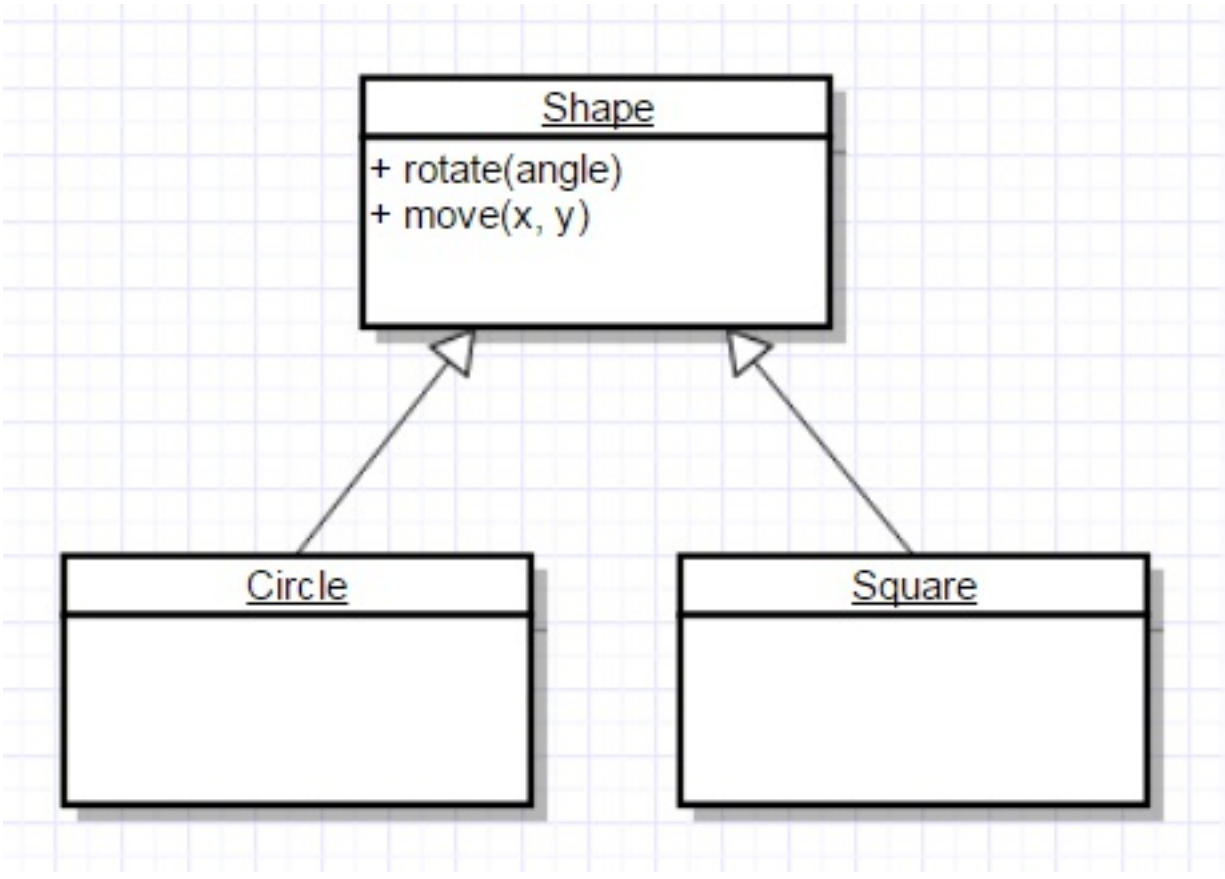
Моделирует отношение «содержит/является частью».

При агрегации класс не контролирует время жизни своей части.



Наследование

Моделирует отношение «является».



Конструирование объекта

Порядок конструирования:

1. Выделяется память под объект
2. Если есть базовые классы, то конструирование начинается с них в порядке их очередности в списке наследования
3. Инициализируются поля класса в том порядке, в котором они объявлены в классе
4. Происходит вызов конструктора

```
class A
{
public:
    A() {} // 3
    ~A() {}

private:
    int x; // 1
    int y; // 2
};

class B
    : public A
{
public:
    B() {} // 5
    ~B() {}

private:
    int z; // 4
};
```

Порядок уничтожения:

1. Происходит вызов деструктора
2. Вызываются деструкторы для полей класса в обратном порядке их объявления в классе
3. Уничтожаются базовые классы в порядке обратном списку наследования

```
class A
{
public:
    A() {}
    ~A() {} // 3

private:
    int x; // 5
    int y; // 4
};

class B
    : public A
{
public:
    B() {}
    ~B() {} // 1

private:
    int z; // 2
};
```

Списки инициализации

```
class A
{
    A()
        : x(5)
        , y(6)
    {
        z = 7;
    }

    int x;
    int y;
    int z;
};
```

Распространенная ошибка:

```
class A
{
    A()
        : y(5) // Инициализация в порядке объявления в классе!
        , x(y)
    {
    }

    int x;
    int y;
};
```

Инициализация в объявлении

```
class A
{
    int x = 3;
};
```

Выравнивание полей

В целях повышения быстродействия данные в памяти должны быть выровнены, то есть размещены определенным образом.

Предпочтительное выравнивание можно узнать:

```
std::cout << alignof(char) << std::endl; // 1
std::cout << alignof(double) << std::endl; // 8
```

Гранулярность памяти

Инструменты для исследования

- **sizeof(T)** - размер типа в байтах
- **offsetof(T, M)** - смещение поля M от начала типа T

```
struct S
{
    char m1;
    double m2;
};
```

```
sizeof(char) == 1
sizeof(double) == 8
sizeof(S) == 16
offsetof(S, m1) == 0
offsetof(S, m2) == 8
```

```
[         char         ][        double        ]
[c][.][.][.][.][.][.][.][d][d][d][d][d][d][d][d]
```

Выравниванием можно управлять:

```
#pragma pack(push, 1)
class S
{
public:
    char m1;
    double m2;
};
#pragma pack(pop)

offsetof(S, m1) == 0
offsetof(S, m2) == 1
sizeof(S) == 9
```

> Работать будет не всегда, компилятор может это проигнорировать, если посчитает, что сделать это нельзя

Оптимизация размера POD структур

```
struct POD
{
    int x;
    double y;
    int z;
};

std::cout << sizeof(POD) << std::endl; // 24
```

```
struct POD
{
    double y;
    int x;
    int z;
};

std::cout << sizeof(POD) << std::endl; // 16
```

Предсказуемое размещение в памяти

Порядок размещения полей класса/структуры в памяти в порядке объявления гарантирован только для простых типов (POD).

Простые типы (POD, Plain Old Data)

1. Скалярные типы (bool, числа, указатели, перечисления (enum), nullptr_t)
2. class или struct которые:
    - Имеют только тривиальные (сгенерированные компилятором) конструктор, деструктор, конструктор копирования
    - Нет виртуальных функций и базового класса
    - Все нестатические поля с модификатором доступа public
    - Не содержит статических полей не POD типа

Примеры

```
class NotPOD
{
public:
    NotPOD(int x)
    {
    }
};
```

```
class NotPOD
    : public Base
{
};
```

```
class NotPOD
{
    virtual void f()
    {
    }
};
```

```
class NotPOD
{
    int x;
};
```

```
class POD
{
public:
    NotPOD m1;
    int m2;
    static double m3;
private:
    void f() {}
};
```

> Копирование простого типа - memcpy

> Простые типы можно использовать для передачи из программы в программу, записи на диск и т.д. Но только на одной и той же платформе!

```
struct POD
{
    int x;
    double y;

    void serialize(File& file) const
    {
        file.write(this, sizeof(POD));
    }
};
```

Инициализация POD типов

```
struct POD
{
    int x;
    double y;
};
```

Инициализация нулем (zero-initialization):

```
POD p1 = POD();
POD p2 {};
POD* p3 = new POD();

// x == 0
// y == 0
```

Инициализация по умолчанию (default-initialization):

```
POD p1;
POD* p2 = new POD;

// x, y содержат мусор
```

Рекомендуемое разделение на заголовочные файлы и файлы с реализацией

a.h

```
#pragma once

struct A
{
    void foo();
};
```

a.cpp

```
#include "a.h"

void A::foo()
{
}
```

Защита от повторного включения

buffer.h

```
class Buffer
{
    ...
};
```

text_processor.h

```
#include "buffer.h"
...
```

main.cpp

```
#include "buffer.h"
#include "text_processor.h"
```

В одной единице трансляции два объявления класса `Buffer`, компилятор не знает какое использовать.

buffer.h

```
#ifndef BUFFER_H
#define BUFFER_H

class Buffer
{
    ...
};

#endif
```

> Или просто `#pragma once`

Циклическое включение

a.h

```
#include "b.h"

class A
{
    B* b;
};
```

b.h

```
#include "a.h"

class B
{
    A* a;
};
```

Предварительное объявление (forward declarations)

a.h

```
class B;

class A
{
    B* b;
};
```

a.cpp

```
#include "b.h"
#include "a.h"

...
```

b.h

```
class A;

class B
{
    A* a;
};
```

Практическая часть

Используя метод рекурсивного спуска, написать калькулятор. Поддерживаемые операции:

- умножение
- деление
- сложение
- вычитание

- унарный минус

Для вычислений использовать тип int64_t, приоритет операций стандартный. Передача выражения осуществляется через аргумент командной строки, поступаемые числа целые, результат выводится в cout. Пример:

```
calc "2 + 3 * 4 - -2"
```

Вывод:

```
16
```

Должна быть обработка ошибок, в случае некорректного выражения выводить в консоль `error`

EOF

Перегрузка методов

> Методы классов - это просто функции, в которые неявно передается указатель на сам класс

```
class Connection
{
public:
    void send(int value);
    void send(const std::string& value);
};
```

> Конструкторы - это тоже функции и их тоже можно перегружать.

```
class Connection
{
public:
    Connection(const std::string& address, uint16_t port);
    Connection(const IpV4& address, uint16_t port);
    Connection(const IpV6& address, uint16_t port);
};
```

> Деструкторы - тоже функции, но перегружать нельзя.

Параметры по умолчанию

```
class Connection
{
public:
    Connection(const std::string& address, uint16_t port = 8080);
};
```

```
class Connection
{
public:
    Connection(const std::string& address = "localhost", uint16_t port = 8080);
};
```

> Пропусков в параметрах по умолчанию быть не должно, начинаться они могут не с первого аргумента, но заканчиваться должны на последнем.

Явные приведения типов

static_cast

Явное приведение встроенных типов:

```
double x = 1.5;

// Убираем предупреждение компилятора,
// четко показываем свои намерения
int y = static_cast<int>(x);
```

Приведение указателя на void

```
int* data = static_cast<int*>(malloc(100 * sizeof(int)));
```

Приведение вверх и вниз по иерархии

```
struct A {};
struct B : public A{};
struct C : public A{};
struct D {};
A* a = new B();
// Ошибка компиляции
D* d = static_cast<D*>(a);
// Безопасно
B* b = static_cast<B*>(a);
// Неопределенное поведение на
// этапе выполнения
C* c = static_cast<C*>(a);
```

const_cast

Снятие или добавление константности.

```
int x = 5;
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

dynamic_cast

1. В базовом классе должна быть хотя бы одна виртуальная функция

2. Требуется RTTI (Runtime Type Information)

```
struct A {};
struct B : public A{};
struct C : public A{};
struct D {};
A* a = new B();
// Ok
B* b = dynamic_cast<B*>(a);
// nullptr
C* c = dynamic_cast<C*>(a);
// nullptr
D* d = dynamic_cast<D*>(a);
```

Если приводить не указатели, а ссылки, то в случае неудачного приведения будет выброшено исключение `std::bad_cast`

> dynamic_cast - признак плохого дизайна

> dynamic_cast - дорог

reinterpret_cast

Приведение одного типа к другому через указатель или ссылку не выполняя никаких проверок.

```
D* d = reinterpret_cast<D*>(a);
int x = reinterpret_cast<int>(a);
```

C-cast

```
B* b = (B*) a;
```

1. Компилятор попробует использовать static_cast
2. Если это не удалось, то reinterpret_cast
3. По необходимости будет добавлен const_cast

Неявные приведения типов

```
int x = 5;
double y = x;
```

```
struct A
{
    A(int x) {}
};

A a = 5;
```

```
struct A
{
    A(int x, int y = 3) {}
};

A a = 5;
```

```
class BigInt
{
public:
    BigInt(int64_t value) {}
};

BigInt x = 5;
```

```
struct A
{
    explicit A(int x) {}
};

A a = 5; // Ошибка
```

Операторы

Операторы сравнения

```
class BigInt
{
    static constexpr size_t Size = 256;
    uint8_t data_[Size];
public:
    bool operator==(const BigInt& other) const
    {
        if (this == &other)
            return true;

        for (size_t i = 0; i < Size; ++i)
            if (data_[i] != other.data_[i])
                return false;

        return true;
    }

    bool operator!=(const BigInt& other) const
    {
        return !(*this == other);
    }
};

BigInt x = 5;

if (c == 5)
    ...
```

Еще операторы сравнения:

- Меньше <
- Больше >
- Меньше или равно <=
- Больше или равно >=

Бинарные арифметические операторы

Попробуем написать метод осуществлющий сложение двух BigInt:

```
class BigInt
{
    const BigInt& operator+(BigInt& other)
    {
        ...
        return *this;
    }
};

BigInt x = 3;
BigInt y = 5;
BigInt z = x + y; // ок
BigInt z = x + y + x; // ошибка
```

```
// x + y -> const BigInt
tmp = x.operator+(y)
// tmp + z
tmp.operator+(x)
// opearator - не константный метод,
// а tmp - константный объект
```

```
class BigInt
{
    BigInt& operator+(BigInt& other)
    {
        ...
        return *this;
    }
};

BigInt x = 3;
BigInt y = 5;
BigInt z = x + y + x; // ок, но x изменился
```

```
class BigInt
{
    BigInt operator+(BigInt& other)
    {
        BigInt tmp;
        ...
        return tmp;
    }
};


BigInt x = 3;
BigInt y = 5;
BigInt z = x + y + x; // ок
```

```
BigInt x = 3;
const BigInt y = 5;
BigInt z = x + y + x; // передача константного
// объекта y по неконстантной ссылке
```

```
class BigInt
{
    BigInt operator+(const BigInt& other)
    {
        BigInt tmp;
        ...
        return tmp;
    }
};

BigInt x = 3;
const BigInt y = 5;
BigInt z = x + y + x; // ок
```

```
const BigInt x = 3;
const BigInt y = 5;
BigInt z = x + y + x; // передача константного
// объекта x по неконстантной ссылке
```

Финальный вариант:

```
class BigInt
{
    BigInt operator+(const BigInt& other) const
    {
        BigInt tmp;
        ...
        return tmp;
    }
};
```

Операторы могут не быть членами класса:

```
class Int128 {};
class BigInt
{
    BigInt operator+(const Int128& other) const
    {
        ...
    }
};
BigInt x = 3;
Int128 y = 5;
BigInt z = x + y; // ok
BigInt z = y + x; // у Int128 нет оператора
// сложения с BigInt
```

```
class BigInt
{
    friend BigInt operator+(const Int128& x, const BigInt& y);
};

BigInt operator+(const Int128& x, const BigInt& y)
{
    ...
}
```

Еще операторы:

- Вычитание -
- Деление /
- Умножение *
- Остаток от деления %

> Для операторов действует стандартный приоритет арифметических операторов

Унарные арифметические операторы

```
BigInt x = 3;
BigInt y = -x;

class BigInt
{
    bool isNegative_;
public:
    BigInt operator-() const
    {
        BigInt tmp(*this);
        tmp.isNegative_ = !isNegative_;
        return tmp;
    }
};
```

Для симметрии есть унарный плюс.

Операторы инкремента

```
Int x = 3;
++x;
x++;

class BigInt
{
    void increment()
    {
        ...
    }
public:
    // ++x
    BigInt& operator++()
    {
        increment();
        return *this;
    }
    // x++
    BigInt operator++(int)
    {
        BigInt tmp(*this);
        increment();
        return tmp;
    }
};
```

Операторы декремента аналогичны операторам инкремента.

Логические операторы

- Отрицание ! (унарный)
- И (логическое умножение) &&
- ИЛИ (логическое сложение) ||

Битовые операторы

- Инверсия ~
- И &
- ИЛИ |
- Исключающее ИЛИ (xor) ^
- Сдвиг влево <<
- Сдвиг вправо >>

Составное присваивание

Все арифметические, логические и побитовые операции только изменяющиие состояние объекта (с = в начале).

```
x += 3;
x *= 4;
```

```
class BigInt
{
    // не константная, так как объект изменяется
    const BigInt& operator+=(const BigInt& other)
    {
        ...
        return *this;
    }
};

BigInt x = 3;
(x += 5) + 7;

class BigInt
{
    BigInt operator+=(const BigInt& other)
    {
        BigInt tmp;
        ...
        return tmp;
    }
};
```

Оператор вывода в поток

Не метод класса.

```
std::ostream& operator<<(std::ostream& out, const BigInt& value)
{
    out << ...;
    return out;
}

BigInt x = 5;
std::cout << x;
```

Операторы доступа

Семантика доступа к массиву.

```
class Array
{
    uint8_t* data_;
public:
    const uint8_t& operator[](size_t i) const
    {
        return data_[i];
    }

    uint8_t& operator[](size_t i)
    {
        return data_[i];
    }
};

Array a;
a[3] = 4;

const Array b;
b[5] = 6; // Ошибка
auto x = b[1]; // Ok
```

Семантика указателя

```
class MyObject
{
public:
    void foo() {}
};

class MyObjectPtr
{
    MyObject* ptr_;
public:
    MyObjectPtr()
        : ptr_(new MyObject())
    {
    }

    ~MyObjectPtr()
    {
        delete ptr_;
    }

    MyObject& operator*()
    {
        return *ptr_;
    }

    const MyObject& operator*() const
    {
        return *ptr_;
    }

    MyObject* operator->()
    {
        return ptr_;
    }

    const MyObject* operator->() const
    {
        return ptr_;
    }
};

MyObjectPtr p;
p->foo();
(*p).foo();
```

Функтор

Позволяет работать с объектом как с функцией.

```
class Less
{
public:
    bool operator()(
        const BigInt& left, const BigInt& right) const
    {
        return left < right;
    }
};

Less less;
if (less(3, 5))
    ...
```

Другие операторы

- new
- delete
- ,

Сокрытие

```
struct A
{
    void foo() {} // 1
};

struct B
    : public A
{
    void foo() {} // 2
};

A a;
a.foo(); // Будет вызвана 1

B b;
b.foo(); // Будет вызвана 2

A* c = new B();
c->foo(); // Будет вызвана 1
```

Виртуальные функции

```
struct A
{
    virtual void foo() const {} // 1
};

struct B
    : public A
{
    void foo() const override {} // 2
};

A a;
a.foo(); // Будет вызвана 1

B b;
b.foo(); // Будет вызвана 2

A* c = new B();
c->foo(); // Будет вызвана 2

const A& d = B();
d.foo(); // Будет вызвана 2
```

В первых двух случаях используется раннее (статическое) связывание, еще на этапе компиляции компилятор знает какой метод вызвать.

В третьем случае используется позднее (динамическое) связывание, компилятор на этапе компиляции не знает какой метод вызвать, выбор нужного метода будет сделан во время выполнения.

Виртуальные функции в C

```cpp
#include <stdio.h>

struct Device
{
    virtual void write(const char* message) {}
};

class Console : public Device
{
    int id_;
public:
    Console(int id)
        : id_(id)
    {
    }

    void write(const char* message) override
    {
        printf("Console %d: %s\n", id_, message);
    }
};

class Socket : public Device
{
    const char* address_;
public:
    Socket(const char* address)
        : address_(address)
    {
    }

    void write(const char* message) override
    {
        printf("Send %s to %s\n", message, address_);
    }
};

int main()
{
    Device* devices[] = {
        new Console(10),
        new Socket("10.0.0.1") };

    Device* dev1 = devices[0];
    dev1->write("A");

    Device* dev2 = devices[1];
    dev2->write("B");

    return 0;
}
```

```
Console 10: A
Send B to 10.0.0.1
```

```
#include <stdio.h>
#include <stdlib.h>

struct Device;

struct DeviceVirtualFunctionTable
{
    void (*write)(Device* self, const char* message);
};

struct Device
{
    DeviceVirtualFunctionTable vft_;
};

void Device_write(Device* self, const char* message)
{
    self->vft_.write(self, message);
}

struct Console
{
    DeviceVirtualFunctionTable vft_;
    int id_;
};

void Console_write(Device* self, const char* message)
{
    Console* console = (Console*) self;
    printf("Console %d: %s\n", console->id_, message);
}

Device* Console_new(int id)
{
    Console* instance = (Console*) malloc(sizeof(Console));
    instance->vft_.write = Console_write;
    instance->id_ = id;
    return (Device*) instance;
}

struct Socket
{
    DeviceVirtualFunctionTable vft_;
    const char* address_;
};

void Socket_write(Device* self, const char* message)
{
    Socket* socket = (Socket*) self;
```

```
    printf("Send %s to %s\n", message, socket->address_);
}

Device* Socket_new(const char* address)
{
    Socket* instance = (Socket*) malloc(sizeof(Socket));
    instance->vft_.write = Socket_write;
    instance->address_ = address;
    return (Device*) instance;
}

int main()
{
    Device* devices[] = {
        Console_new(10),
        Socket_new("10.0.0.1") };

    Device* dev1 = devices[0];
    Device_write(dev1, "A");

    Device* dev2 = devices[1];
    Device_write(dev2, "B");

    return 0;
}
```

```
Console 10: A
Send B to 10.0.0.1
```

Таблица виртуальных функций

Если в классе или в каком-либо его базовом классе есть виртуальная функция, то каждый объект хранит указатель на таблицу виртуальных функций.

Таблица представляет собой массив из указателей на функции.

```
struct A
{
    void foo() {}
    int x;
};

struct B
{
    virtual void foo() {}
    int x;
};

std::cout << sizeof(A) << '\n';
std::cout << sizeof(B) << '\n';
```

```
4
16
```

Виртуальный деструктор

```
struct A
{
    ~A()
    {
        std::cout << "A";
    }
};

struct B
    : public A
{
    ~B()
    {
        std::cout << "B";
        delete object_;
    }

    SomeObject* object_;
};

A* a = new B();
delete a;
```

```
A
```

> Произошла утечка, так как не был вызван деструктор, в котором мы освобождали ресурс.

```
struct A
{
    virtual ~A()
    {
    }
};
```

> Используете наследование? Сделайте деструктор виртуальным.

Чисто виртуальные функции (pure virtual)

```
class Writer
{
public:
    virtual void ~Writer() {}

    virtual void write(const char* message) = 0;
};

class ConsoleWriter
    : public Writer
{
public:
    void write(const char* message) override
    {
        std::cout << message;
    }
}
```

Абстрактные классы

Классы имеющие хоть одну чисто виртуальную функцию - абстрактные. При попытке создать их компилятор выдаст ошибку. Если в производном классе не сделать реализацию чисто виртуальной функции, то он тоже становится абстрактным.

> Абстрактные классы в C++ - продвинутые интерфейсные классы в других языках.

Виртуальные функции и параметры по умолчанию

```
struct A
{
    virtual void foo(int i = 10)
    {
        std::cout << i; // 1
    }
};

struct B
    : public A
{
    virtual void foo(int i = 20)
    {
        std::cout << i; // 2
    }
};

A* a = new B();
a->foo(); // Будет вызвана 2, вывод 10

B* b = new B();
b->foo(); // Будет вызвана 2, вывод 20

A* a = new A();
a->foo(); // Будет вызвана 1, вывод 10
```

> Лучше избегать параметров по умолчанию для виртуальных функций

Модификаторы доступа при наследовании

```
class A
{
public:
    int x_;
protected:
    int y_;
private:
    int z_;
};
```

Псевдокод! Поля базового класса после наследования имеют такие модификаторы:

```
class B : public A
{
public:
    int x_;
protected:
    int y_;
};

A* a = new B(); // Ok
```

```
class B : protected A
{
protected:
    int x_;
    int y_;
};

A* a = new B(); // Ошибка
```

```
class  B : private A
{
private:
    int x_;
    int y_;
};

A* a = new B(); // Ошибка
```

public - классическое ООП наследование

```
class Device
{
};

class NetworkAdapter
    : public Device
{
};

class DeviceManager
{
    void addDevice(Device* dev)
    {
    }
}

devManager.addDevice(new NetworkAdapter());
```

private - наследование реализации

```
class NetworkAdapter
    : public Device
    , private Loggable
{
};

Loggable* l = new NetworkAdapter(); // Ошибка
```

final

```
struct A final
{
};

struct B : public A // Ошибка
```

Множественное наследование

```
struct A
{
    virtual ~A() {}
    double x;
    double y;
};

struct B : public A { };

struct C : public A { };

struct D
    : public B
    , public C
{
};
```


```
// 2 * 8(double) + 1 * 8(vtable)
sizeof(A) == 24
sizeof(B) == 24
sizeof(C) == 24
// (2 * 8(double) + 1 * 8(vtable)) + (2 * 8(double) + 1 * 8(vtable))
sizeof(D) == 48
```

```
[A][B][D]
         [A][C]
```

```
struct A
{
    A(double x)
        : x(x)
        , y(0)
    {
    }
    virtual ~A() {}
    double x;
    double y;
};

struct B : public A
{
    B(double x)
        : A(x)
    {
        y = x * 2;
    }
};

struct C : public A
{
    C(double x)
        : A(x)
    {
        y = x * 2;
    }
};

struct D
    : public B
    , public C
{
    D()
        : B(2)
        , C(3)
    {
        B::y = 1;
        C::y = 2;
    }
};
```

Ромбовидное наследование

```
struct A
{
    virtual ~A() {}
    double x;
    double y;
};

struct B : virtual public A { };

struct C : virtual public A { };

struct D
    : public B
    , public C
{
};
```


```
// 2 * 8(double) + 1 * 8(vtable)
sizeof(A) == 24
// 2 * 8(double) + 2 * 8(vtable)
sizeof(B) == 32
sizeof(C) == 32
// 2 * 8(double) + 3 * 8(vtable)
sizeof(D) == 40
```

```
[B][D]
      [C]
         [A]
```

Вложенные классы

```
class Vector
{
public:
    class Iterator
    {
    };

private:
    char* data_;
};

Vector::Iterator it = ...
```

> Имеют доступ к закрытой части внешнего класса

Практическая часть

Нужно написать класс-матрицу, тип элементов int. В конструкторе задается количество рядов и строк. Поддерживаются операции: получить количество строк(rows)/ столбцов(columns), получить конкретный элемент, умножить на число(*=), сравнение на равенство/неравенство. В случае ошибки выхода за границы бросать исключение:

```
throw std::out_of_range("")
```

Пример:

```
const size_t rows = 5;
const size_t cols = 3;

Matrix m(rows, cols);

assert(m.getRows() == 5);
assert(m.getColumns() == 3);

m[1][2] = 5; // строка 1, колонка 2
double x = m[4][1];

m *= 3; // умножение на число

Matrix m1(rows, cols);

if (m1 == m)
{
}
```

Для проверки на сайте класс должен быть оформлен как заголовочный файл, название класса - Matrix. Нужно скомпилировать файл `test.cpp`, в нем включается заголовочный файл `matrix.h` и тестируется. В случае успеха в выводе кроме строки done ничего больше не будет.

Подсказка

Чтобы реализовать семантику [][] понадобится прокси-класс. Оператор матрицы возращает другой класс, в котором тоже используется оператор [] и уже этот класс возвращает значение.

EOF

Argument-dependent name lookup (ADL)

Известен также, как Koenig lookup.

```
namespace X
{
    struct A { ... };

    std::ostream& operator<<(
        std::ostream& out, const A& value) { ... }

    void foo(const A& value) { ... }
}

X::A a;

std::cout << a;
foo(a);
```

Компилятор ищет функцию в текущем пространстве имен и если не находит, то в пространствах имен аргументов. Если находит подходящую функцию в двух местах, то возникает ошибка.

Методы генерируемые компилятором неявно

```
struct A
{
    X x;
    Y y;
    // Конструктор
    A()
        : x(X())
        , y(Y())
    {}
    // Деструктор
    ~A()
    {}
    // Копирующий конструктор
    // A a1;
    // A a2 = a1;
    A(const A& copied)
        : x(copied.x)
        , y(copied.y)
    {}
    // Оператор копирования
    // A a1;
    // A a2;
    // a2 = a1;
    A& operator=(const A& copied)
    {
        x = copied.x;
        y = copied.y;
        return *this;
    }
    // Перемещающий конструктор
    // A a1;
    // A a2 = std::move(a1);
    A(A&& moved)
        : x(std::move(moved.x))
        , y(std::move(moved.y))
    {}
    // Оператор перемещения
    // A a1;
    // A a2;
    // a2 = std::move(a1);
    A& operator=(A&& moved)
    {
        x = std::move(moved.x);
        y = std::move(moved.y);
        return *this;
    }
};
```

Правило тройки (пятерки)

Если явно объявить один из следующих методов:

- деструктор
- конструктор копирования
- оператор копирования

(после C++11, еще два)

- конструктор перемещения
- оператор перемещения

То компилятор не будет генерировать остальные автоматически, поэтому если они вам нужны, вы должны реализовать их самостоятельно.

rvalue-ссылка и lvalue-ссылка

До стандарта C++11 было два типа значений:

1. lvalue
2. rvalue

> "Объект - это некоторая **именованная область памяти**; lvalue - это выражение, обозначающее объект. Термин "lvalue" произошел от записи присваивания E1 = E2, в которой левый (left - левый(англ.), отсюда буква l, value - значение) операнд E1 должен быть выражением lvalue."

Керниган и Ритчи

```
int a = 1;
int b = 2;
int c = (a + b);
int foo() { return 3; }
int d = foo();

1 = a; // left operand must be l-value
foo() = 2; // left operand must be l-value
(a + b) = 3; // left operand must be l-value
```

1. Ссылается на объект - lvalue
2. Если можно взять адрес - lvalue
3. Все что не lvalue, то rvalue

```
int a = 3;
a; // lvalue
int& b = a;
b; // lvalue, ссылается на a
int* c = &a;
*c; // lvalue, ссылается на a
void foo(int val)
{
    val; // lvalue
}
void foo(int& val)
{
    val; // lvalue, ссылается на val
}
int& bar() { return a; }
bar(); // lvalue, ссылается на a
```

```
3; // rvalue
(a + b); // rvalue
int bar() { return 1; }
bar(); // rvalue
```

lvalue-ссылка

Ссылка на lvalue.

```
int a = 3;
int& b = a;
```

```
int& a = 3; // ошибка
const int& a = 3; // ок
a; // const lvalue
```

Объект жив до тех пор, пока жива ссылающаяся на него константная ссылка.

rvalue-ссылка

```cpp
#include <iostream>

int x = 0;

int val() { return 0; }
int& ref() { return x; }

void test(int&)
{
    std::cout << "lvalue\n";
}

void test(int&&)
{
    std::cout << "rvalue\n";
}

int main()
{
    test(0); // rvalue
    test(x); // lvalue
    test(val()); // rvalue
    test(ref()); // lvalue
    test(std::move(x)); // rvalue
    return 0;
}
```

> `std::move` приводит lvalue к rvalue

Копирование

Семантика: в результате копирования должна появится точная копия объекта.

```cpp
int x = 3;
int y = x;
// x == y

String a;
String b = a;
String c;
c = a;
// a == b == c
```

```
class String
{
    size_t size_;
    char* data_;
public:
    ~String()
    {
        delete[] data_;
    }

    // String b1;
    // String b2 = b1;
    String(const String& copied)
        : data_(new char[copied.size_])
        , size_(copied.size_)
    {
        std::copy(copied.data_, copied.data_ + size_, data_);
    }

    // String b1;
    // String b2;
    // b2 = b1;
    String& operator=(const String& copied)
    {
        // Плохо
        delete[] data_;
        data_ = new char[copied.size_];
        size_ = copied.size_;
        std::copy(copied.data_, copied.data_ + size_, data_);
        return *this;
    }
};
```

```
String b1;
b1 = b1;

std::vector<String> words;
...
words[to] = words[from];
```

Проверяйте на присваивание самому себе.

```
String& operator=(const String& copied)
{
    if (this == &copied)
        return *this;
    // Плохо
    delete[] data_;
    data_ = new char[copied.size_];
    size_ = copied.size_;
    std::copy(copied.data_, copied.data_ + size_, data_);
    return *this;
}
```

Финальный вариант:

```
String& operator=(const String& copied)
{
    if (this == &copied)
        return *this;
    char* ptr = new char[copied.size_];
    delete[] data_;
    data_ = ptr;
    size_ = copied.size_;
    std::copy(copied.data_, copied.data_ + size_, data_);
    return *this;
}
```

> Подумайте, а стоит ли писать конструктор/оператор копирования самостоятельно?

Копирование и наследование

```
struct A
{
    A() {}
    A(const A& a) {}
    virtual A& operator=(const A& copied)
        { return *this; }
};

class B
    : public A
{
public:
    B() {}

    B(B& b)
        : A(b)
    {
    }

    A& operator=(const A& copied) override
    {
        A::operator=(copied);
        return *this;
    }
};
```

Срезка

```
void foo(A a)
{
    // Срезанный до A объект
}

B a;
foo(a);
```

Нежелательное копирование

```
void send(std::vector<char> data)
{
    ...
}
```

```
void print(const std::vector<char>& data)
```

Используйте передачу по ссылке!

Явный запрет копирования

До C++11:

```
class Noncopyable
{
    Noncopyable(const Noncopyable&);
    Noncopyable& operator=(const Noncopyable&);
};

class Buffer
    : private Noncopyable
{
};
```

boost::noncopyable устроен именно так.

C++11:

```
class Buffer
{
    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;
};
```

Явное указание компилятору сгенерировать конструктор и оператор копирования

```
class Buffer
{
public:
    Buffer(const Buffer&) = default;
    Buffer& operator=(const Buffer&) = default;
};
```

Перемещение

Семантика: в результате перемещения в объекте, куда происходит перемещение, должна появиться точная копия перемещаемого объекта, оригинальный объект после этого остается в неопределенном, но корректном состоянии.

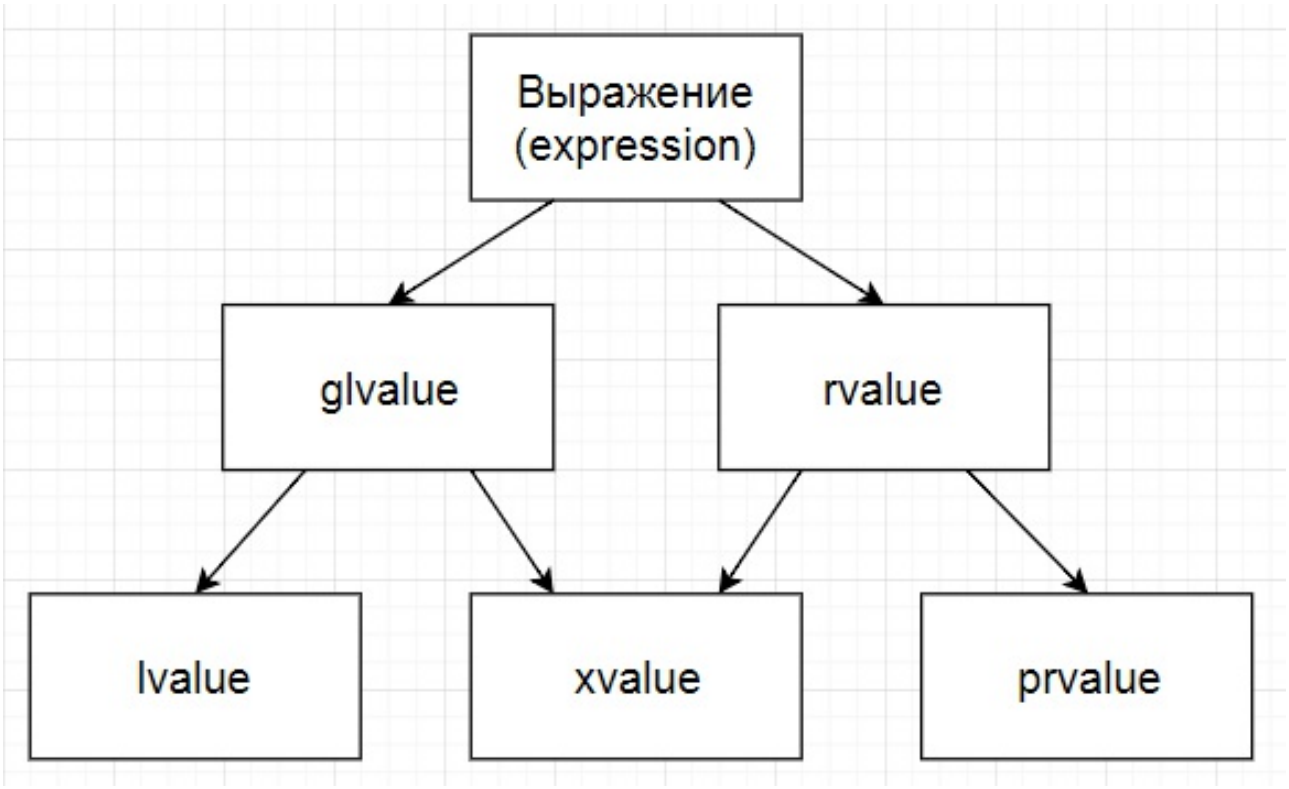
Передача владения

```
class unique_ptr
{
    T* data_;
};
```

Производительность

```
class Buffer
{
    char* data_;
    size_t size_;
};
```

lvalue и rvalue начиная с C++11



glvalue ("generalized" lvalue)

Выражение, чьё вычисление определяет сущность объекта.

prvalue ("pure" rvalue)

Выражение, чьё вычисление инициализирует объект или вычисляет значение операнда оператора, с соответствии с контекстом использования.

xvalue ("eXpiring" value)

Это glvalue, которое обозначает объект, чьи ресурсы могут быть повторно использованы (обычно потому, что они находятся около конца своего времени жизни).

lvalue

Это glvalue, которое не является xvalue.

rvalue

Это prvalue или xvalue.

Пример

lvalue

Выражение является lvalue, если ссылается на объект уже имеющий имя доступное вне выражения.

```
int a = 3;
a; // lvalue
int& b = a;
b; // lvalue
int* c = &a;
*c; // lvalue

int& foo() { return a; }
foo(); // lvalue
```

xvalue

1. Результат вызова функции возвращающей rvalue-ссылку

```
int&& foo() { return 3; }
foo(); // xvalue
```

2. Явное приведение к rvalue

```
static_cast<int&&>(5); // xvalue
std::move(5); // эквивалентно static_cast<int&&>
```

3. Результат доступа к нестатическому члену, объекта xvalue значения

```
struct A
{
    int i;
};

A&& foo() { return A(); }

foo().i; // xvalue
```

prvalue

Не принадлежит ни к lvalue, ни к xvalue.

```
int foo() { return 3; }
foo(); // prvalue
```

rvalue

Все что принадлежит к xvalue или prvalue.

glvalue

Все что принадлежит к xvalue или lrvalue.

Практическое правило (Скотт Мейерс)

1. Можно взять адрес - lvalue
2. Ссылается на lvalue (T&, const T&) - lvalue
3. Иначе rvalue

> Как правило rvalue соответствует временным объектам, например, возвращаемым из функций или создаваемых в результате неявных приведений типов. Также это большинство литералов.

Классификация

Есть имя	Может быть перемещено	Тип
да	нет	glvalue, lvalue
да	да	rvalue, xvalue, glvalue
нет	да	rvalue, prvalue

Еще примеры

```
void foo(int) {}
void foo(int&) {}
void foo(int&&) {}
```

```
void foo(int) {} // <-- этот?
void foo(int&) {} //  // <-- или этот?
void foo(int&&) {}

int x = 1;
foo(x); // lvalue
```

```
int x = 1;
int& y = x;
foo(y); // lvalue

void foo(int) {} // <-- этот?
void foo(int&) {} // <-- или этот?
void foo(int&&) {}
```

```
foo(1); // rvalue

void foo(int) {} // <-- этот?
void foo(int&) {}
void foo(int&&) {} // <-- или этот?
```

```
int bar() { return 1; }
foo(bar()); // rvalue

void foo(int) {} // <-- этот?
void foo(int&) {}
void foo(int&&) {} // <-- или этот?
```

```
foo(1 + 2); // rvalue

void foo(int) {} // <-- этот?
void foo(int&) {}
void foo(int&&) {} // <-- или этот?
```

Конструктор/оператор перемещения

```
class Buffer
{
    size_t size_;
    char* data_;
public:
    ~Buffer()
    {
        delete[] data_;
    }

    // Buffer b1;
    // Buffer b2 = std::move(b1);
    Buffer(Buffer&& moved)
        : data_(moved.data_)
        , size_(moved.size_)
    {
        moved.data_ = nullptr;
        moved.size_ = 0;
    }

    // Buffer b1;
    // Buffer b2;
    // b2 = std::move(b1);
    Buffer& operator=(Buffer&& moved)
    {
        if (this == &moved)
            return *this;
        delete[] data_;
        data_ = moved.data_;
        size_ = moved.size_;
        moved.data_ = nullptr;
        moved.size_ = 0;
        return *this;
    }
};
```

Явное указание компилятору сгенерировать конструктор и оператор перемещения

```
class Buffer
{
public:
    Buffer(Buffer&&) = default;
    Buffer& operator=(Buffer&&) = default;
};
```

Явное указание компилятору запретить перемещение

```
class Buffer
{
public:
    Buffer(Buffer&&) = delete;
    Buffer& operator=(Buffer&&) = delete;
};
```

Perfect forwarding

Задача: передать аргумент не создавая временных копий и не изменяя типа передаваемого аргумента.

```
void bar(T&) {}
void bar(T&&) {}
```

```
void foo(T x)
{
    // копия
    bar(x);
}
```

```
void foo(T& x)
{
    // Может приводить к ошибкам
    // компиляции, если x - rvalue:
    // foo(T());
    bar(x);
}

void foo(T&& x)
{
    // ок, но пришлось написать перегрузку
    bar(std::move(x));
}
```

Решение:

```
void foo(T&& x)
{
    bar(std::forward<T>(x));
}
```

Return value optimization (RVO)

Позволяет сконструировать возвращаемый объект в точке вызова.

```
Server makeServer(uint16_t port)
{
    Server server(port);
    server.setup(...);
    return server;
}

Server s = makeServer(8080);
```

Не мешайте компилятору:

```
Server&& makeServer(uint16_t port)
{
    Server server(port);
    server.setup(...);
    return std::move(server); // так не надо
}
```

Copy elision

Оптимизация компилятора разрешающая избегать лишнего вызова копирующего конструктора.

```
struct A
{
    explicit A(int) {}
    A(const A&) {}
};

A y = A(5); // Копирующий конструктор вызван не будет
```

> В копирующих конструкторах должна быть логика отвечающая только за копирование.

Шаблоны

Шаблоны классов

```
class Matrix
{
    double* data_;
};
```

```
class MatrixDouble
{
    double* data_;
};

class MatrixInt
{
    int* data_;
};
```

```
template <class T>
class Matrix
{
    T* data_;
};
```

```
Matrix<double> m;
Matrix<int> m;
```

Шаблоны функций

```
template <class T>
void printLine(const T& value)
{
    std::cout << value << '\n';
}
```

```
printLine<int>(5);
```

Компилятор может самостоятельно вывести тип шаблона в зависимости от аргументов вызова.

```
printLine(5);
```

class или typename

```
template <class T>
void printLine(const T& value)
{
}
```

```
template <typename T>
void printLine(const T& value)
{
}
```

Никакой разницы.

Практическая часть

Написать класс для работы с большими целыми числами. Размер числа ограничен только размером памяти. Нужно поддержать семантику работы с обычным int:

```
BigInt a = 1;
BigInt b = a;
BigInt c = a + b + 2;
```

Реализовать оператор вывода в поток, сложение, вычитание, унарный минус, все операции сравнения.

std::vector и другие контейнеры использовать нельзя - управляйте памятью сами.

EOF

Инстанцирование шаблона

Инстанцирование шаблона – это генерация кода функции или класса по шаблону для конкретных параметров.

```
template <class T>
bool lessThan7(T value) { return value < 7; }
```

```
lessThan7(5); //  Инстанцирование
// bool print(int value) { return value < 7; }

lessThan7(5.0); //  Инстанцирование
// bool print(double value)  { return value < 7; }
```

Явное указание типа

```
lessThan7<double>(5); //  Инстанцирование
// bool print(double value) { return value < 7; }
```

Константы как аргументы шаблона

```
template <class T, size_t Size>
class Array
{
    T data_[Size];
};
```

```
Array<int, 5> a;
```

Ограничения на параметры шаблона не являющиеся типами

Так можно:

```
template <int N>
int foo()
{
    return N * 2;
}
```

A double нельзя:

```
template <double N> // Ошибка
void foo()
{
}
```

float тоже нельзя.

> Причины исторические, почему не исправлено до сих пор не знаю.

Параметры шаблона должны быть известны на этапе компиляции.

```
template <int N>
void foo() { }

int x = 3;
foo<x>(); // Ошибка
```

Константы на литералы можно:

```
template <int N>
void foo() {}

const int x = 3;
foo<x>(); // Ok
```

А с обычной константой нельзя:

```
int bar() { return 0; }

template <int N>
void foo() { }

const int x = bar();
foo<x>(); // Ошибка
```

Но если вычислять значение во время компиляции, то можно:

```
constexpr int bar() { return 0; }

template <int N>
void foo() {}

const int x = bar();
foo<x>(); // Ok
```

> constexpr говорит компилятору, что надо стараться вычислить значение на этапе компиляции

Нельзя использовать объекты класса:

```
struct A {};

template <A a> // Ошибка
void foo()
{
}
```

Можно указатель на const char:

```
template <const char* s>
void foo()
{
}
```

И это даже можно инстанцировать nullptr или 0:

```
foo<nullptr>();
foo<0>();
```

Но нельзя литералом:

```
foo<"some text">(); // Ошибка
```

Параметры шаблона по умолчанию

```
template <class X, class Y = int>
void foo()
{
}

foo<char>();
```

```
template <class T, class ContainerT = std::vector<T>>
class Queue
{
    ContainerT data_;
};

Queue<int> queue;
```

Специализация шаблона

```
template <class T>
class Vector
{
    ...
}

template <>
class Vector<bool>
{
    ...
};
```

Суммирование последовательности от n, до 0:

```
#include <iostream>

template <int n>
int sum();

template <>
int sum<0>() { return 0; }

// template <>
// int sum<1>() { return 1; }
// template <>
// int sum<2>() { return 2 + 1; }
// ...

template <int n>
int sum()
{
    return n + sum<n - 1>();
}

int main()
{
    std::cout << sum<3>() << '\n';
    return 0;
}
```

```
int sum<0>():
        mov     eax, 0
        ret
main:
        call    int sum<3>()
        call    operator<<(int)
        ret
int sum<3>():
        call    int sum<2>()
        add     eax, 3
        ret
int sum<2>():
        call    int sum<1>()
        add     eax, 2
        ret
int sum<1>():
        call    int sum<0>()
        add     eax, 1
        ret
```

Разбухание кода

Необдуманное использование шаблонов может привести к разбуханию кода, кода становится много, он перестает помещаться в кеш, что ведет к существенным издержкам.

Еще реализация суммирования

```
template <int n>
struct sum;

template <>
struct sum<0>
{
    static constexpr int value = 0;
};

template <int n>
struct sum
{
    static constexpr int value = n + sum<n - 1>::value;
};

int main()
{
    std::cout << sum<3>::value << '\n';
    return 0;
}
```

```
main:
        mov     esi, 6
        call    operator<<(int)
        ret
```

Вычисления времени компиляции

А стоит ли?

1. Сложно для понимания и поддержки
2. Замедляет компиляцию

> Вероятно лучшей альтернативой будет скрипт делающий вычисления и генерирующий C++ код из констант с рассчитаными значениями.

Псевдонимы типов

Старый способ:

```
typedef int Seconds;
typedef Queue<int> IntegerQueue;

Seconds i = 5;
IntegerQueue j;
```

Новый (рекомендуемый) способ:

```
using Seconds = int;
using IntegerQueue = Queue<int>;

Seconds i = 5;
IntegerQueue j;
```

Псевдонимы типов для шаблонов:

```
template <class T>
using MyQueue = Queue<T, std::deque<T>>;

MyQueue<int> y;
```

Новый синтаксис функций

```
auto foo() -> void
{
}
```

auto

Позволяет статически определить тип по типу выражения.

```
auto i = 5;
auto j = foo();
```

range-based for и auto

```
for (auto i : { 1, 2, 3 })
    std::cout << i;
```

```
for (auto& i : data)
    i.foo();
```

decltype

Позволяет статически определить тип по типу выражения.

```
int foo() { return 0; }

decltype(foo()) x = 5;
// decltype(foo()) -> int
// int x = 5;
```

```
void foo(decltype(bar()) i)
{
}
```

Определение типа аргументов шаблона функций

```
template <typename T>
T min(T x, T y)
{
    return x < y ? x : y;
}

min(1, 2); // ok
min(0.5, 2); // error
min<double>(0.5, 2); // ok
```

```
template <typename X, typename Y>
X min(X x, Y y)
{
    return x < y ? x : y;
}

min(1.5, 2); // ok
min(1, 0.5); // ok?
```

```
template <typename X, typename Y>
auto min(X x, Y y) -> decltype(x + y)
{
  return x < y ? x : y;
}

min(1.5, 2); // ok
min(1, 0.5); // ok
```

typename

```
struct String
{
    using Char = wchar_t;
};

template <class T>
class Parser
{
    T::Char buffer[]; // Ошибка
};
```

Если компилятор встречая идентификатор в шаблоне, может его трактовать как тип или что-то иное (например, как статическую переменную), то он выбирает иное.

```
struct String
{
    using Char = wchar_t;
};

template <class T>
class Parser
{
    typename T::Char buffer[]; // Ok
};
```

Ошибка инстанцирования шаблона

```
template <class T>
bool lessThan7(T value) { return value < 7; }
```

```
struct A {};
A a;
lessThan7(a); //  Инстанцирование
// bool print(A value)  { return value < 7; }
// Ошибка инстанцирования, тип a не имеет operator<(int)
```

SFINAE (Substitution Failure Is Not An Error)

При определении перегрузок функции ошибочные инстанциации шаблонов не вызывают ошибку компиляции, а отбрасываются из списка кандидатов на наиболее подходящую перегрузку.

Неудачное инстанцирование шаблона - это не ошибка.

Например, позволяет на этапе компиляции выбрать нужную функцию:

```
// C++11

template<typename T>
void clear(T& t,
    typename std::enable_if<std::is_pod<T>::value>::type* = nullptr)
{
    std::memset(&t, 0, sizeof(t));
}

// Для не-POD типов
template<typename T>
void clear(T& t,
    typename std::enable_if<!std::is_pod<T>::value>::type* = nullptr)
{
    t = T{};
}
```

is_pod

```cpp
template <class T>
struct is_pod
{
    static constexpr bool value = false;
};
```

```cpp
template <>
struct is_pod<int>
{
    static constexpr bool value = true;
};
```

enable_if

```cpp
template<bool, typename T = void>
struct enable_if
{
};

// Частичная специализация для true
template<typename T>
struct enable_if<true, T>
{
    using type = T;
};

enable_if<false, int>::type // Ошибка, нет type
enable_if<true, int>::type // Ок, type == int
```

```cpp
// C++14

template<typename T>
void clear(T& t, std::enable_if_t<std::is_pod<T>::value>* = nullptr)
{
    std::memset(&t, 0, sizeof(t));
}

// Для не-POD типов
template<typename T>
void clear(T& t, std::enable_if_t<!std::is_pod<T>::value>* = nullptr)
{
    t = T{};
}
```

Можно получить на этапе компиляции информацию о типе, например, проверим есть ли у класса некий метод:

```
struct A
{
    void foo() {}
};

struct B
{
};

template<typename T>
struct HasFoo
{
    static constexpr bool value = true;
};

int main()
{
    std::cout << hasFoo<A>::value << '\n';
    std::cout << hasFoo<B>::value << '\n';
    return 0;
}
```

```
template<typename T>
struct HasFoo
{
    static constexpr bool value = ???;
};
```

Нам нужно будет 2 функции: одна принимает класс с нужным нам методом, другая принимает все остальное:

```
template<typename T>
struct HasFoo
{
    // Принимает все
    static int check(...);

    // Принимает нужный нам класс,
    // где есть какая-то foo()
    template <class U>
    static auto check(U* u) -> decltype(u->foo());
};
```

По возвращаемому функцией типу мы поймем, какая из перегрузок была использована, если тип совпадет, то это то, что нам нужно.

Проверка совпадения типов:

```
template <class T1, class T2>
struct IsSame
{
    static constexpr bool value = false;
};

template <class T>
struct IsSame<T, T>
{
    static constexpr bool value = true;
};
```

Финальный вариант:

```
template<typename T>
struct HasFoo
{
private:
    static int check(...);

    template <class U>
    static auto check(U* u) -> decltype(u->foo());

public:
    static constexpr bool value =
        IsSame
        <
            void,
            decltype(HasFoo<T>::check((T*) nullptr))
        >::value;
};
```

```
hasFoo<A>::value == true;
hasFoo<B>::value == false;
```

type_traits

В стандартной библиотеки функции определения свойств типов is_* находятся в заголовочном файле type_traits

Примеры:

```
is_integral  // Является ли тип целочисленным
is_floating_point // Является ли тип типом с плавающей точкой
is_array // Является ли тип типом массива
is_const  // Содержит ли тип в себе квалификатор const
is_pod // Является ли тип POD-типом
has_virtual_destructor // Имеет ли виртуальный деструктор

// И так далее
```

Шаблоны свойств (traits)

```
template <typename T>
struct NumericTraits
{
};

template <> // Специализация
struct NumericTraits<char>
{
    static constexpr int64_t min = -128;
    static constexpr int64_t max = 127;
};
```

```
template <typename T>
class Calculator
{
    T getNumber(const std::string& text)
    {
        const int64_t value = toNumber(text);
        if (value < NumericTraits<T>::min
            || value > NumericTraits<T>::max)
        {
            // range error
        }
    }
};
```

> Смотрите заголовочный файл numeric_limits

Не только значения, но и типы:

```
template <class T>
class BasicStream
{
public:
    using Char = T;
};

using Utf8Stream = BasicStream<char>;

Utf8Stream::Char c;
```

Классы стратегий

Класс стратегий - интерфейс для применения стратегий в алгоритме.

```
class Json
{
public:
    void encode(const char* data, size_t size) {}
};

class Xml
{
public:
    void encode(const char* data, size_t size) {}
};

template <class T, class Format>
class Connector
{
    Format format_;
public:
    void connect()
    {
        auto packet = makeConnectPacket();
        auto encodedPacket = format_.encode(
            packet.data, packet.size);
        send(encodedPacket);
    }
};

template<class T>
using JsonConnector = Connector<T, Json>;
```

Отличия между свойствами и стратегиями

Свойство - отличительная особенность характеризующая сущность.

Стратегия - образ действия сущности.

Сравнение динамического и статического полиморфизма

```
class Device
{
public:
    virtual ~Device() {}

    virtual void write(const char* data, size_t size) = 0;
};

class File final
    : public Device
{
public:
    void write(const char* data, size_t size) override {}
};

class Stream
{
    Device* device_;
public:
    explicit Stream(Device* device)
        : device_(device)
    {
    }

    void putChar(char c)
    {
        device_->write(&c, 1);
    }
};

auto stream = Stream(new File("file.txt"));
```

```
class File
{
public:
    explicit File(const char* name) {}
    void write(const char* data, size_t size) {}
};

template <class Device>
class Stream
{
    Device device_;
public:
    explicit Stream(Device&& device)
        : device_(std::move(device))
    {
    }

    void putChar(char c)
    {
        device_.write(&c, 1);
    }
};

using FileStream = BasicStream<File>;

FileStream stream(File("data"));
```

1. Динамический полиморфизм более гибок и позволяет настраивать поведение во время выполнения, но имеет накладные расходы на вызов виртуальных методов
2. Статический полиморфизм не имеет накладных расходов, но менее гибок

Шаблоны с произвольным количеством аргументов (variadic templates)

```
print(1, "abc", 2.5);
```

```
template <class T>
void print(T&& val)
{
    std::cout << val << '\n';
}

template <class T, class... Args>
void print(T&& val, Args&&... args)
{
    std::cout << val << '\n';
    print(std::forward<Args>(args)...);
}
```

Практическая часть

Простой сериализатор поддерживающий два типа: uint64_t и bool.

```
struct Data
{
    uint64_t a;
    bool b;
    uint64_t c;
};

Data x { 1, true, 2 };

std::stringstream stream;

Serializer serializer(stream);
serializer.save(x);

Data y { 0, false, 0 };

Deserializer deserializer(stream);
const Error err = deserializer.load(y);

assert(err == Error::NoError);

assert(x.a == y.a);
assert(x.b == y.b);
assert(x.c == y.c);
```

Сериализовать в текстовый вид с разделением пробелом, bool сериализуется как true и false

Подсказки по реализации

```
struct Data
{
    uint64_t a;
    bool b;
    uint64_t c;

    template <class Serializer>
    Error serialize(Serializer& serializer)
    {
        return serializer(a, b, c);
    }
};
```

```
// serializer.h
#pragma once

enum class Error
{
    NoError,
    CorruptedArchive
};

class Serializer
{
    static constexpr char Separator = ' ';
public:
    explicit Serializer(std::ostream& out)
        : out_(out)
    {
    }

    template <class T>
    Error save(T& object)
    {
        return object.serialize(*this);
    }

    template <class... ArgsT>
    Error operator()(ArgsT... args)
    {
        return process(args...);
    }

private:
    // process использует variadic templates
};
```

Deserializer реализуется аналогично Serializer, только принимает std::istream, а не std::ostream

Пример десериализации bool:

```
Error load(bool& value)
{
    std::string text;
    in_ >> text;

    if (text == "true")
        value = true;
    else if (text == "false")
        value = false;
    else
        return Error::CorruptedArchive;

    return Error::NoError;
}
```

EOF

Обработка ошибок

1. Возврат кода ошибки
2. Исключения

Возврат кода ошибки

```
enum class Error
{
    Success,
    Failure
};

Error doSomething()
{
    return Error::Success;
}

if (doSomething() != Error::Success)
{
    showError();
}
```

+ Простота

- Ошибку можно проигнорировать

- Делает код громозким

```
auto data = readData("data.json");

Json data;
auto error = readData(data, "data.json");
if (error != Success)
{
    ...
}
```

Поддержка со стороны C++

```
#include <system_error>
```

```
enum class HttpError
{
    NoError = 200,
    NotFound = 404
};

class HttpCategory:
    public std::error_category
{
public:
    const char* name() const noexcept override
    {
        return "http";
    }

    std::string message(int code) const override
    {
        switch (code)
        {
        case 200: return "ok";
        case 404: return "not found";
        }
        assert(!"invalid error code");
    }
};

std::error_code make_error_code(HttpError error)
{
    static const HttpCategory instance;
    return std::error_code(
        static_cast<int>(error),
        instance);
}
```

```
std::error_code download(const std::string& url)
{
    return make_error_code(HttpError::NotFound);
}

const auto error = download("http://1.1.1.1");
if (error)
{
    std::cerr << error << '\n';
    std::cerr << error.message() << '\n';
}
```

```
http:404
not found
```

Исключения

- Вопросы производительности

- При неправильном использовании могут усложнить программу

+ Нельзя проигнорировать

```
struct Error
{
    std::string message_;
    const char* fileName_;
    int line_;
    Error(const std::string& message,
        const char* fileName, int line)
        : message_(message)
        , fileName_(fileName)
        , line_(line)
    {
    }
};

void doSomething()
{
    throw Error(
        "doSomething error", __FILE__, __LINE__);
}

try
{
    doSomething();
}
catch (const Error& error)
{
    showError();
}
```

Что такое исключительная ситуация?

Ошибка которую нельзя обработать на данном уровне и игнорирование которой делает дальнейшую работу программы бессмысленной.

Гарантии безопасности исключений (exception safety)

1. Гарантировано искючений нет (No-throw guarantee)

Операции всегда завершаются успешно, если исключительная ситуация возникла она обрабатывается внутри операции.

2. Строгая гарантия (Strong exception safety)

Также известна как коммит ролбек семантика (commit/rollback semantics). Операции могут завершиться неудачей, но неудачные операции гарантированно не имеют побочных эффектов, поэтому все данные сохраняют свои исходные значения.

```
std::vector<int> source = ...;
try
{
    std::vector<int> tmp = source;
    tmp.push_back(getNumber());
    tmp.push_back(getNumber()); <-- Исключение
    tmp.push_back(getNumber());
    source.swap(tmp);
}
catch (...)
{
    return;
}
```

3. Базовая гарантия (Basic exception safety)

Выполнение неудачных операций может вызвать побочные эффекты, но все инварианты сохраняются и нет утечек ресурсов (включая утечку памяти). Любые сохраненные данные будут содержать допустимые значения, даже если они отличаются от того, что они были до исключения.

```
source.push_back(getNumber());
source.push_back(getNumber()); <-- Исключение
source.push_back(getNumber());
```

4. Никаких гарантий (No exception safety)

Поиск подходящего обработчика

```
class Error {};

class ArgumentError : public Error
{
    std::string message_;
public:
    ArgumentError(std::string&& message);
    const std::string& getMessage() const;
};

File openFile(const std::string& name)
{
    if (name.empty())
        throw ArgumentError("empty file name");
}


try
{
    auto file = openFile("data.json");
    auto json = file.readAll();
}
catch (const ArgumentError& error)
{
    std::cerr << error.getMessage();
}
catch (const Error& error)
{
}
catch (...)
{
}
```

1. Поиск подходящего обработчика идет в порядке следования обработчиков в коде
2. Полного соответствия типа не требуется, будет выбран первый подходящий обработчик
3. Если перехватывать исключение по значению, то возможна срезка до базового класса
4. Если наиболее общий обработчик идет раньше, то более специализированный обработчик никогда не будет вызван
5. Три точки - перехват любого исключения

> Исключения ОС - не исключения C++, например, деление на ноль. Для их обработки нужно использовать средства предоставляемые конкретной платформой

Раскрутка стека

```
struct A {};
struct Error {};
struct FileError : public Error {};

void foo()
{
    A a1;
    throw Error();
}

void bar()
{
    A a2;
    try
    {
        A a3;
        foo();
    }
    catch (const FileError&)
    {
    }
}

bar();
```

Поиск подходящего обработчика вниз по стеку вызовов с вызовом деструкторов локальных объектов - раскрутка стека.

Если подходящий обработчик не был найден вызывается стандартная функция terminate.

terminate

Вызывает стандартную функцию C - abort.

abort - аварийное завершение программы, деструкторы объектов вызваны не будут.

Поведение terimnate можно изменить установив свой обработчик функцией set_terminate.

Где уместен catch (...)?

Только в main, для того, чтобы поймать необработанное исключение, чтобы избежать вызов terminate и таким образом завершить работу с вызовом деструкторов.

```
int main()
{
    try
    {
        ...
    }
    catch (...)
    {
        std::cerr << "unknown error";
    }
}
```

Перезапуск исключения

```
try
{
    foo();
}
catch (...)
{
    std::cerr << "something wrong";
    throw;
}
```

noexcept

```
void foo() noexcept
{
}
```

noexcept говорит компилятору, что функция не выбрасывает исключений - это позволяет компилятору генерировать более компактный код, но если фактически исключение было выброшено, то будет вызвана функция terminate.

Исключения в деструкторе

Исключение покинувшее деструктор во время раскрутки стека или у глобального/статического объекта приведет к вызову terminate.

Начиная с C++11 все деструкторы компилятором воспринимаются как помеченные noexcept - теперь исключения не должны покидать деструктора никогда.

Исключения в конструкторе

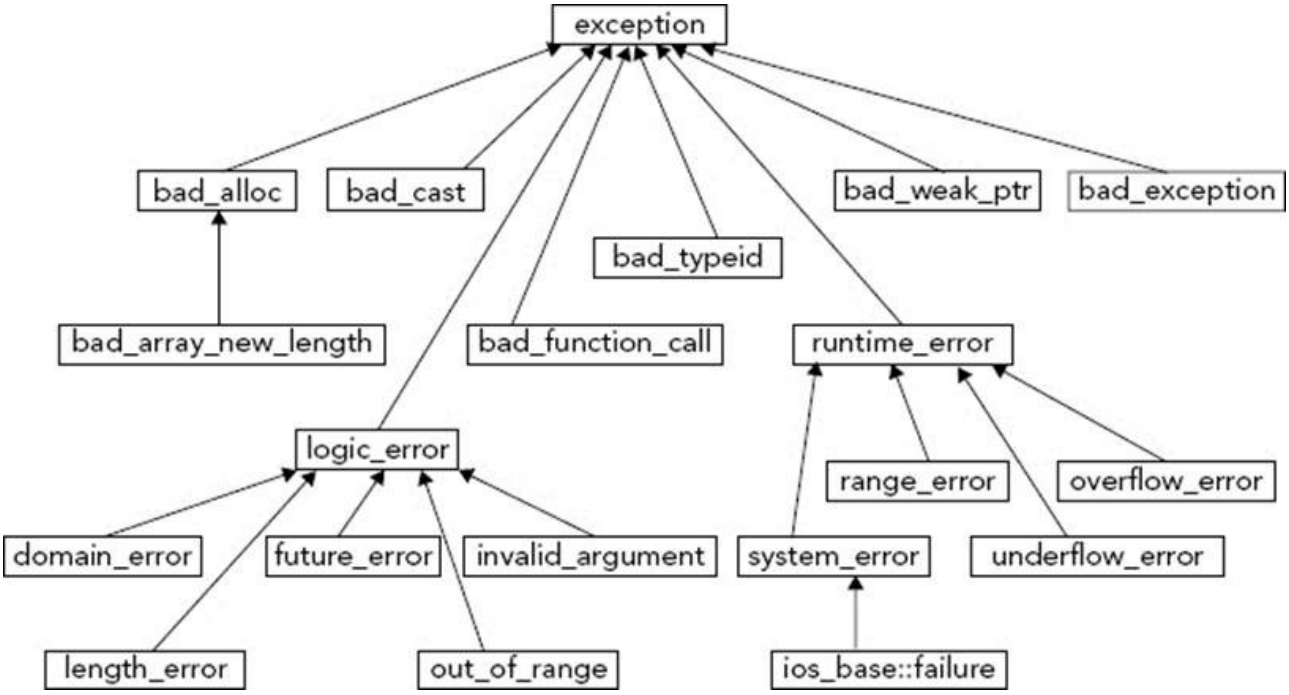
Клиент либо получает объект в консистентном состоянии, либо не получает ничего.

```
class Socket
{
    static constexpr size_t BufferSize = 2048;
    char* buffer_;
public:
    explicit Socket(const std::string& address)
        : data_(new char[BufferSize]) // <- утечка
    {
        if (address.empty())
            throw ArgumentError();
    }

    ~Socket()
    {
        delete[] buffer_; // Не будет вызван
    }
};
```

Для полностью сконструированных на момент выброса исключения объектов будут вызваны деструкторы, память выделенная под объект будет корректно освобождена, но поскольку объект не был полностью сконструирован, то деструктор вызван не будет.

Стандартные классы рекомендуемые для исключений

```
#include <stdexcept>
```




```
class exception
{
public:
    explicit exception(char const* const message);

    virtual char const* what() const;
```

Управление ресурсами

Используем идеому RAII (Resource Acquire Is Initialization):

```
struct Buffer
{
    explicit Buffer(size_t size)
        : data_(new char[size])
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }

    char* data_;
};
```

```
class Socket
{
    static constexpr size_t BufferSize = 2048;
    Buffer buffer_;
public:
    explicit Socket(const std::string& address)
        : buffer_(BufferSize)
    {
        if (address.empty())
            throw ArgumentError();
    }
};
```

Исключения под капотом

```
struct A
{
    A() {}
    ~A() {}
};

void bar() noexcept
{
}

void foo()
{
    A a;
    bar();
}
```

```
A::A() [base object constructor]:
        ret
A::~A() [base object destructor]:
        ret
bar():
        ret
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        lea     rdi, [rbp - 8]
        call    A::A() [base object constructor]
        call    bar()
        lea     rdi, [rbp - 8]
        call    A::~A() [base object destructor]
        add     rsp, 16
        pop     rbp
        ret
```

Убираем noexcept

```
struct A
{
    A() {}
    ~A() {}
};

void bar() {}

void foo()
{
    A a;
    bar();
}
```

```
A::A() [base object constructor]:
        ret
A::~A() [base object destructor]:
        ret
bar():
        ret
foo():
        call    A::A() [base object constructor]
        call    bar()
        jmp     .LBB1_1
.LBB1_1:
        call    A::~A() [base object destructor]
        ret
.LBB1_2: # landing pad
        call    A::~A() [base object destructor]
        call    _Unwind_Resume
```

Появился специальный блок (landing pad) используемый при раскрутке стека.

Добавляем блок catch

```
struct A
{
    A() {}
    ~A() {}
};

void bar() {}

void baz() noexcept {}

void foo()
{
    A a;
    try
    {
        bar();
    }
    catch (...)
    {
        baz();
    }
}
```

```
foo():
        call    A::A() [base object constructor]
        call    bar()
        jmp     .LBB2_1
.LBB2_1:
        jmp     .LBB2_5
.LBB2_2:
        call    __cxa_begin_catch
        call    baz()
        call    __cxa_end_catch
        jmp     .LBB2_4
.LBB2_4:
        jmp     .LBB2_5
.LBB2_5:
        call    A::~A() [base object destructor]
        ret
.LBB2_6:
        call    A::~A() [base object destructor]
        call    _Unwind_Resume
```

Выбрасываем исключение

```
struct A
{
    A() {}
    ~A() {}
};

void bar()
{
    throw A();
}

void baz() noexcept
{
}

void foo()
{
    A a;
    try
    {
        bar();
    }
    catch (...)
    {
        baz();
    }
}
```

```
bar():
        call    __cxa_allocate_exception
        call    A::A() [base object constructor]
        jmp     .LBB0_1
.LBB0_1:
        call    __cxa_throw
.LBB0_2: # landing pad
        call    __cxa_free_exception
        call    _Unwind_Resume
foo():
        call    A::A() [base object constructor]
        call    bar()
        jmp     .LBB4_1
.LBB4_1:
        jmp     .LBB4_5
.LBB4_2:
        call    __cxa_begin_catch
        call    baz()
        call    __cxa_end_catch
        jmp     .LBB4_4
.LBB4_4:
        jmp     .LBB4_5
.LBB4_5:
        call    A::~A() [base object destructor]
        ret
.LBB4_6:
        call    A::~A() [base object destructor]
        call    _Unwind_Resume
typeinfo name for A:
        .asciz  "1A"
typeinfo for A:
        .quad   vtable for __cxxabiv1::__class_type_info+16
        .quad   typeinfo name for A
```

Компиляция с включенной оптимизацией

```
struct A
{
    A() {}
    ~A() {}
};

void bar(int x)
{
    if (x == 1)
        throw A();
}

void baz() noexcept
{
}

void foo(int x)
{
    A a;
    try
    {
        bar(x);
    }
    catch (...)
    {
        baz();
    }
}
```

```
bar(int):
        cmp     edi, 1
        je      .LBB0_2
        ret
.LBB0_2:
        call    __cxa_allocate_exception
        call    __cxa_throw
foo(int):
        cmp     edi, 1
        je      .LBB3_1
        ret
.LBB3_1:
        call    __cxa_allocate_exception
        call    __cxa_throw
.LBB3_3:
        call    __cxa_begin_catch
        jmp     __cxa_end_catch         # TAILCALL
typeinfo name for A:
        .asciz  "1A"
typeinfo for A:
        .quad   vtable for __cxxabiv1::__class_type_info+16
        .quad   typeinfo name for A
```

Управление памятью

Стандартная библиотека предлагает два умных указателя для автоматического управления памятью:

1. unique_ptr
2. shared_ptr / weak_ptr

unique_ptr

- Монопольное владение памятью, в конструкторе захват, в деструкторе освобождение
- Копирование запрещено, перемещение разрешено

```
std::unique_ptr<MyClass> x(new MyClass());
auto y = std::make_unique<MyClass>(); // C++14

std::unique_ptr<char[]> z(new char[1024]);
```

shared_ptr

- Совместное владение памятью
- Копирование увеличивает счетчик ссылок
- В деструкторе счетчик уменьшается и если становится равным 0, то объект уничтожается

```
std::shared_ptr<MyClass> x(new MyClass());
auto y = std::make_shared<MyClass>();
```

Точки следования (sequence points)

Точки следования - это точки в программе, где состояние реальной программы полностью соответствует состоянию следуемого из исходного кода.

Точки следования необходимы для того, чтобы компилятор мог делать оптимизацию кода.

```
// Может быть утечка
foo(
    std::shared_ptr<MyClass>(new MyClass()),
    bar());
```

Компилятор может заменить это выражение на следующее:

```
auto tmp1 = new MyClass();
auto tmp2 = bar();
auto tmp3 = std::shared_ptr<MyClass>(tmp1);
foo(tmp1, tmp3);
```

Если из bar вылетит исключение, то объект на который указывает tmp1 будет некому удалить.

Решение 1:

```
std::shared_ptr<MyClass> x(new MyClass());
foo(x, bar()); // ok
```

Решение 2:

```
foo(std::make_shared<MyClass>(), bar()); // ok
```

Местонахождение точек:

1. В конце каждого полного выражения - ;
2. В точке вызова функции после вычисления всех аргументов
3. Сразу после возврата функции, перед тем как любой другой код из вызвавшей функции начал выполняться
4. После первого выражения (a) в следующих конструкциях:

```
    a || b
    a && b
    a, b
    a ? b : c
```

Если программа пытается модифицировать одну переменную дважды не пересекая точку следования, то это ведет к неопределенному поведению (undefined behavior):

```
int x = 0;
x = x++; // <-- UB

int i = 0;
i = i++ + ++i; // <-- UB
```

Схематичное устройство shared_ptr

```
#include <cassert>
#include <iostream>

template <class T>
class SharedPtr
{
    struct Data
    {
        T* object_;
        int counter_;
    };

    Data* data_;

    void release()
    {
        --data_->counter_;
        if (data_->counter_ == 0)
        {
            delete data_->object_;
            delete data_;
        }
```

```
    }

public:
    SharedPtr(T* object = nullptr)
        : data_(new Data { object, 1 })
    {
    }

    ~SharedPtr()
    {
        release();
    }

    SharedPtr(const SharedPtr<T>& copied)
        : data_(copied.data_)
    {
        ++data_->counter_;
    }

    SharedPtr& operator=(const SharedPtr<T>& copied)
    {
        if (data_ == copied.data_)
            return *this;

        release();

        data_ = copied.data_;
        ++data_->counter_;
        return *this;
    }

    T& operator*()
    {
        return *data_->object_;
    }

    const T& operator*() const
    {
        return *data_->object_;
    }

    T* operator->()
    {
        return data_->object_;
    }

    const T* operator->() const
    {
        return data_->object_;
    }
};
```

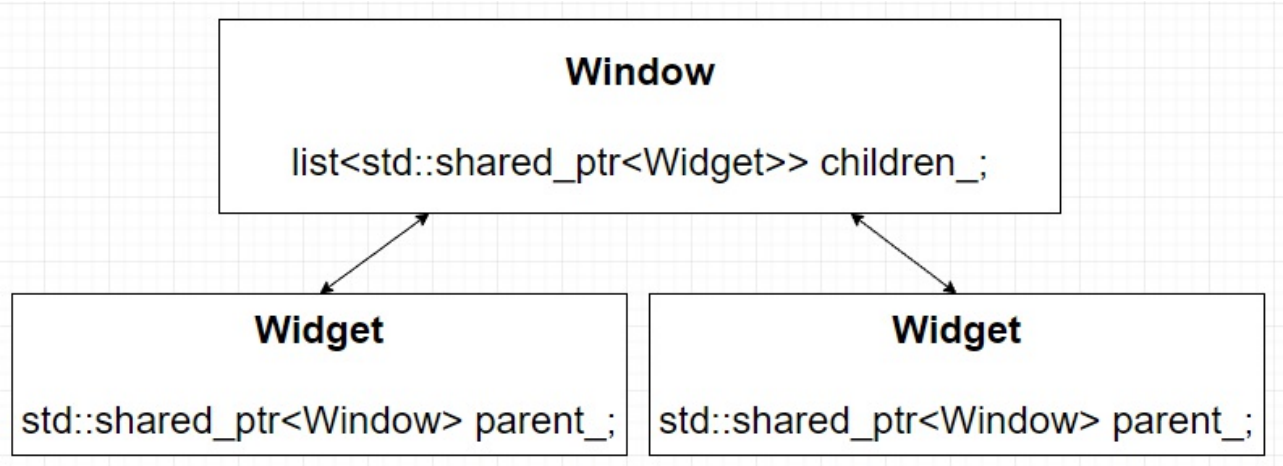
```
struct A
{
    A() { std::cout << "A" << std::endl; }
    ~A() { std::cout << "~A" << std::endl; }
    void foo() { std::cout << this << std::endl; }
};

SharedPtr<A> foo(SharedPtr<A> x)
{
    return x;
}

int main()
{
    auto x = foo(new A());
    auto y = x;
    y->foo();
    (*x).foo();
    y = nullptr;
    return 0;
}
```

Предпочитайте make_shared

```
auto x = std::shared_ptr<MyClass>(new MyClass());
auto x = std::make_shared<MyClass>();
```

1. Нет дублирования (MyClass два раза)
2. Безопасно в вызове функций
3. Оптимально - 1 вызов new вместо 2

Проблема циклических ссылок



```
class Widget;

class Window
{
    std::vector<std::shared_ptr<Widget>> children_;
};

class Widget
{
    std::shared_ptr<Window> parent_;
};
```

Winwow не может быть удален, так как в Widget жив shared_ptr на него, а Widget в свою очередь не может быть удален, так как жив Window.

> Ключевой вопрос C++ - кто кем владеет

weak_ptr

```
class Widget;

class Window
{
    std::vector<std::shared_ptr<Widget>> children_;
};

class Widget
{
    std::weak_ptr<Window> parent_;
};
```

weak_ptr не принимает владение объектом, но располагая weak_ptr всегда можно узнать жив ли объект и если жив, то получить на него shared_ptr.

```
std::shared_ptr<A> x;
std::weak_ptr<A> weak = x;
std::shared_ptr<A> y = weak.lock();
if (y)
{
    ...
}
```

enable_shared_from_this

Иногда нужно получить shared_ptr от самого себя, например, очень актуально при асинхронном взаимодействии, когда время жизни объекта не определено.

```
class A
{
    std::shared_ptr<A> getSharedPtr()
    {
        // Приведет к многократному удалению
        return std::shared_ptr<A>(this);
    }
};
```

Решение:

```
class A
    : public std::enable_shared_from_this<A>
{
    std::shared_ptr<A> getSharedPtr()
    {
        return shared_from_this(); // Ok
    }
};
```

Ограничения enable_shared_from_this

```
class A
    : public std::enable_shared_from_this<A>
{
    A()
    {
        shared_from_this(); // throw std::bad_weak_ptr
    }

    ~A()
    {
        shared_from_this(); // throw std::bad_weak_ptr
    }
};
```

Также перед использованием shared_from_this на объект уже должен ссылаться shared_ptr:

```
auto a = std::make_shared<A>();
auto b = a->getSharedPtr();
```

Практическая часть

Написать функцию для форматирования строки, поддерживаться должен любой тип, который может быть выведен в поток вывода. Формат строки форматирования:

```
"{0} any text {1} {0}"
```

Номер в фигурных скобках - номер аргумента. Если аргументов меньше, чем число в скобках, и в случае прочих ошибок выбрасывать исключение std::runtime_error

Пример:

```
auto text = format("{1}+{1} = {0}", 2, "one");
assert(text == "one+one = 2");
```

Фигурные скобки - зарезервированный символ, если встречаются вне контекста {n} выбрасывать исключение std::runtime_error

EOF### Функтор (функциональный объект)

Объект ведущий себя подобно функции.

```
template <class T>
class Less
{
    const T& x_;
private:
    Less(const T& x)
        : x_(x)
    {
    }

    bool operator()(const T& y) const
    {
        return x_ < y;
    }
};

Less lessThen3(3);

bool result = lessThen3(5); // false
```

Лямбда-функция

Краткая форма записи анонимных функторов.

```
auto lessThen3 = [](int y) { return 3 < y; };

bool result = lessThen3(5); // false
```

Лямбда - краткая форма анонимного функтора

```
int x = 3;
auto add3 = [x](int y) { return x + y; };
auto s = add3(5); // 8
```

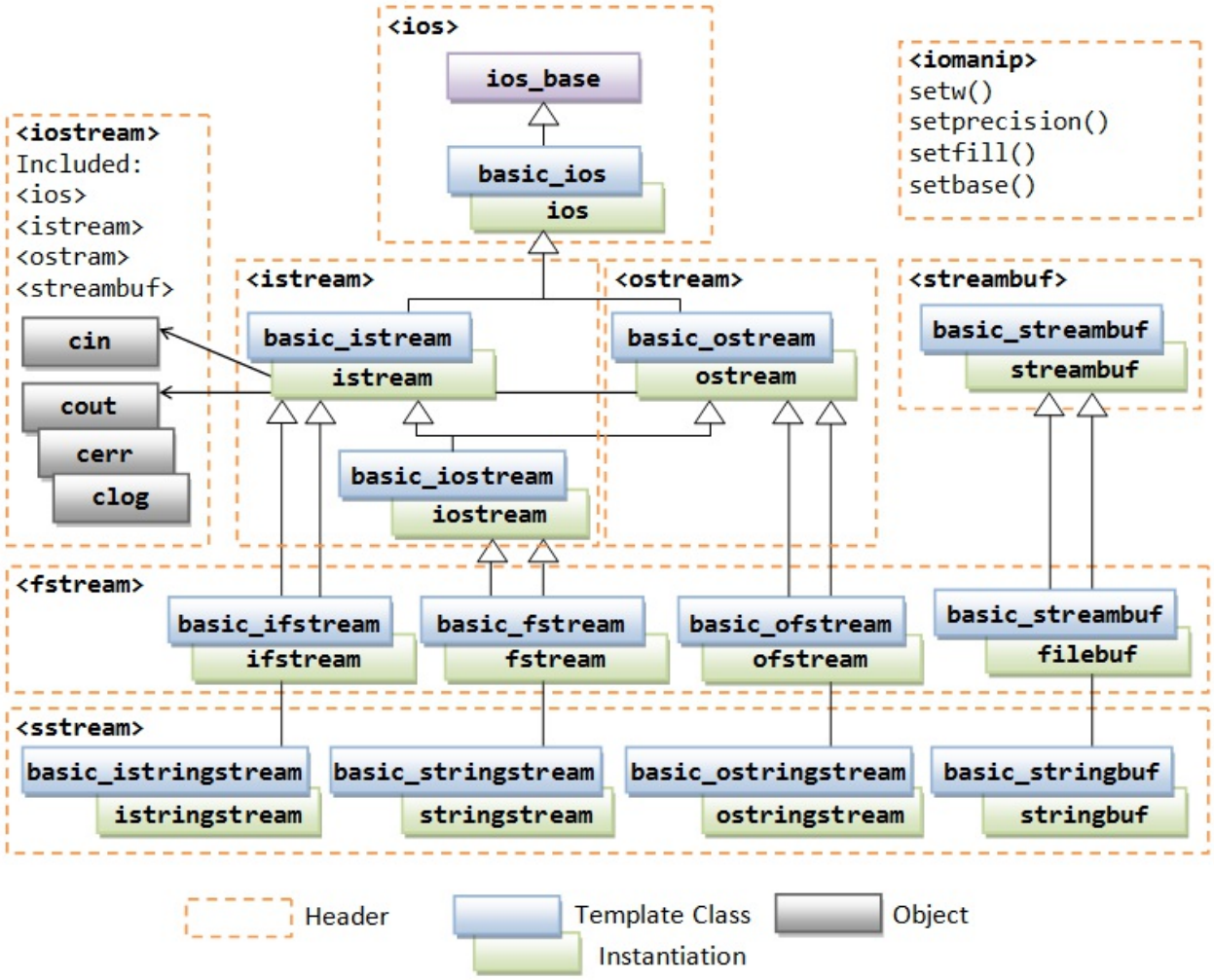
```
class lambda__a123 // Сгенерированное имя
{
    int x_;
public:
    explicit sum(int x)
        : x_(x)
    {
    }

    int operator()(int y) const
    {
        return x_ + y;
    }
};
auto add3 = lambda__a123(3);
auto s = add3(5); // 8
```

Стандартная библиотека C++

1. Ввод-вывод
2. Многопоточность
3. Регулярные выражения
4. Библиотека C
5. Библиотека шаблонов STL
6. Прочее (дата и время, обработка ошибок, поддержка локализации и т.д.)

> Документация: https://en.cppreference.com/w/ (https://en.cppreference.com/w/)

Потоки ввода-вывода



Файловый ввод-вывод

```
#include <fstream>
```

std::ifstream

Чтение из файла.

```
std::ifstream file("/tmp/file.txt");
if (!file)
{
    std::cout << "can't open file" ;
    return;
}

while (file.good())
{
    std::string s;
    file >> s;
}
```

```
const int size = 1024;
char buf[size];

std::ifstream file("/tmp/file.data", std::ios::binary);
file.read(buf, size);
const auto readed = file.gcount();
```

std::ofstream

Запись в файл.

```
std::ofstream file("/tmp/file.txt");
if (!file)
{
    std::cout << "can't open file" ;
    return;
}

file << "abc" << 123;
```

```
const int size = 1024;
char buf[size];

std::ofstream file("/tmp/file.data", std::ios::binary);
file.write(buf, size);
```

Вспомогательные классы

std::pair

Тип позволяющий упаковать два значения в один объект.

```
#include <utility>

auto p1 = std::pair<int, double>(1, 2.0);
auto p2 = std::make_pair(1, 2.0);

auto x = p1.first; // int == 1
auto y = p1.second; // double == 2
```

> pair имеет операторы сравнения позволяющие сделать лексикографическое сравнение элементов.

std::tuple

Тип позволяющий упаковать несколько значений в один объект.

```cpp
#include <tuple>

auto t = std::make_tuple(1, 2.0, "abc");
int a = std::get<0>(t);
double b = std::get<1>(t);
std::string c = std::get<2>(t);
```

> Соответствие типов проверяется на этапе компиляции.

> Как и pair имеет лексикографические операторы сравнения.

std::tie

tie, как и make_tuple создает tuple, но не объектов, а ссылок на них.

Использование tie для написания операторов сравнения

```cpp
struct MyClass
{
    int x_;
    std::string y_;
    double z_;

    bool operator<(const MyClass& o) const
    {
        return std::tie(x_, y_, z_) < std::tie(o.x_, o.y_, o.z_);
    }
};
```

```cpp
bool operator<(const MyClass& o) const
{
    if (x_ != o.x_)
        return x_ < o.x_;
    if (y_ != o.y_)
        return y_ < o.y_;
    return z_ < o.z_;
}
```

Библиотека шаблонов STL (Standard Template Library)

1. Контейнеры (containers) – хранение набора объектов в памяти
2. Итераторы (iterators) – средства для доступа к источнику данных (контейнер, поток)
3. Алгоритмы (algorithms) – типовые операции с данными
4. Адаптеры (adaptors) – обеспечение требуемого интерфейса
5. Функциональные объекты (functors) – функция как объект для использования другими компонентами

О большое

«О» большое – математическое обозначение для сравнения асимптотического поведения алгоритма.

Фраза «сложность алгоритма есть O(f(n))» означает, что с ростом параметра n время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на f(n).

Типичные значения:

1. Время выполнения константно: O(1)
2. Линейное время: O(n)
3. Логарифмическое время: O(log n)
4. Время выполнения «n логарифмов n»: O(n log n)

5. Квадратичное время: O(n²)

Контейнеры

1. Последовательные (Sequence containers)
2. Ассоциативные (Associative containers)
3. Неупорядоченные ассоциативные (Unordered associative containers)
4. Контейнеры-адаптеры (Container adaptors)

Последовательные контейнеры

std::array

```
#include <array>

template <class T, std::size_t N>
class array
{
    T data_[N];
    size_t size_;
public:
    using size_type = size_t;
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;

    constexpr size_type size() const noexcept
    {
        return size_;
    }

    constexpr bool empty() const noexcept
    {
        return false;
    }

    reference at(size_type pos)
    {
        if (size_ <= pos)
            throw std::out_of_range(std::to_string(pos));
        return data_[pos];
    }

    constexpr const_reference at(size_type pos) const;

    reference operator[](size_type pos)
    {
        return data_[pos];
    }

    constexpr const_reference operator[](size_type pos) const;
```

```
    reference front()
    {
        return data_[0];
    }

    constexpr const_reference front() const;

    reference back()
    {
        return data_[size_ - 1];
    }

    constexpr const_reference back() const;

    T* data() noexcept
    {
        return data_;
    }

    const T* data() const noexcept;

    void swap(array<T, N>& other);
};
```

Вставка Удаление Поиск Доступ

- - O(n) O(1)

```
std::array<int, 5> a = { 1, 2, 3, 4, 5 };
auto x = a[2];
a[2] = x * 2;
```

std::initializer_list

```
template <class T>
class initializer_list
{
public:
    size_type size() const noexcept;
    const T* begin() const noexcept;
    const T* end() const noexcept;
};
```

```
Array<int, 3> a = { 1, 2, 3 };
```

```
template <class T, size_t N>
class Array
{
public:
    Array(std::initializer_list<T> init)
    {
        size_t i = 0;
        auto current = init.begin();
        const auto end = init.end();
        while (current != end)
        {
            data_[i++] = *current++;
        }
    }
};
```

std::vector

```cpp
template<class T,
    class Alloc = std::allocator<T>>
class vector
{
public:
    using size_type = size_t;
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
    using allocator_type = Alloc;

    explicit vector(size_type count);
    vector(size_type count, const value_type& defaultValue);
    vector(initializer_list<value_type> init);

    iterator begin() noexcept;
    reverse_iterator rbegin() noexcept;
    const_iterator cbegin() const noexcept;
    const_reverse_iterator crbegin() const noexcept;

    iterator end() noexcept;
    reverse_iterator rend() noexcept;
    const_iterator cend() const noexcept;
    const_reverse_iterator crend() const noexcept;

    void push_back(value_type&& value);
    void push_back(const value_type& value);

    template<class... VT>
    void emplace_back(VT&&... values);

    iterator insert(const_iterator where, T&& value);
    iterator insert(const_iterator where, const T& value);

    template<class... VT>
    iterator emplace(const_iterator where, VT&&... values);

    void reserve(size_type count); // Выделяет память
    size_type capacity() const noexcept;

    void resize(size_type newSize); // Изменяет размер
    void resize(size_type newsize, const value_type& defaultValue);

    iterator erase(const_iterator where);

    // [from, to)
    iterator erase(const_iterator from, const_iterator to);

    void clear() noexcept;
};
```

emplace_back vs push_back(&&)

```
class A
{
    A(int, int) {}
    A(A&&) {}
};

A a(1, 2);

vec.push_back(std::move(a));
vec.emplace(1, 2);
```

Вектор - динамический массив, при добавлении элементов может изменять размер.

Вставка	Удаление	Поиск	Доступ
O(n)	O(n)	O(n)	O(1)
В конце O(1) или O(n)	В конце O(1)	В отсортированном O(log n)	

Трюки с вектором

Быстрое удаление O(1)

Если порядок элементов не важен, то меняем удаляемый элемент с последним местами и удаляем последний (pop_back).

Изменение размера вектора перед вставкой

```
const auto size = file.size();
std::vector<char> data(size);
for (size_t i = 0; i < size; ++i)
    data[i] = file.read();
```

Позволяет сократить количество переаллокаций и существенно ускорить код.

Очистка вектора

```
std::vector<int> data;
for (int i = 0; i < 100500; ++i)
    data.push_back(i);
data.clear();
std::cout << data.capacity() << std::endl; // >= 100500
data.swap(std::vector<int>());
std::cout << data.capacity() << std::endl; // 0
```

```
data.shrink_to_fit(); // C++11
```

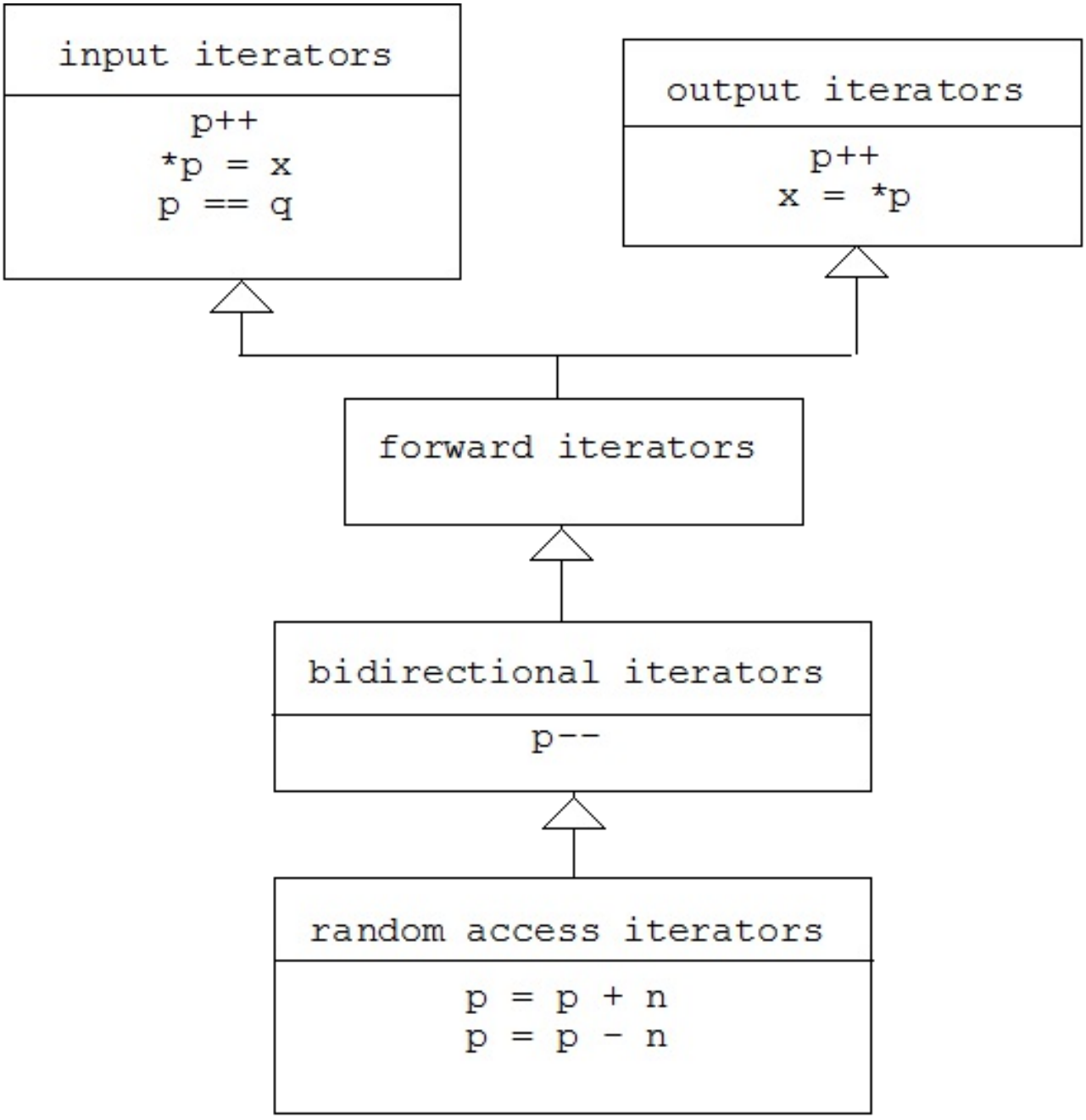
Итераторы (iterators)

Объект предоставляющий доступ к элементам коллекции и осуществляющий навигацию по ним.

Позволяет реализовать универсальные алгоритмы работы с контейнерами.

Классификация итераторов:

- Ввода (Input Iterator)
- Однонаправленные (Forward Iterator)
- Двунаправленные (Bidirectional Iterator)
- Произвольного доступа (Random Access Iterator)
- Вывода (Output Iterator)




```
template <class T, size_t N>
class Array
{
    T data_[N];
};
```

```
template<
    typename _Category,
    typename _Tp,
    typename _Distance = ptrdiff_t,
    typename _Pointer = _Tp*,
    typename _Reference = _Tp&>
struct iterator
{
    /// One of the @link iterator_tags tag types@endlink.
    typedef _Category  iterator_category;
    /// The type "pointed to" by the iterator.
    typedef _Tp        value_type;
    /// Distance between iterators is represented as this type.
    typedef _Distance  difference_type;
    /// This type represents a pointer-to-value_type.
    typedef _Pointer   pointer;
    /// This type represents a reference-to-value_type.
    typedef _Reference reference;
};
```

```
template <class T>
class Iterator
    : public std::iterator<std::forward_iterator_tag, T>
{
    T* ptr_;
public:
    using reference = T&;

    explicit Iterator(T* ptr)
        : ptr_(ptr)
    {
    }

    bool operator==(const Iterator<T>& other) const
    {
        return ptr_ == other.ptr_;
    }

    bool operator!=(const Iterator<T>& other) const
    {
        return !(*this == other);
    }

    reference operator*() const
    {
        return *ptr_;
    }

    Iterator& operator++()
    {
        ++ptr_;
        return *this;
    }
};
```

```
template <class T, size_t N>
class Array
{
    T data_[N];
public:
    using iterator = Iterator<T>;

    iterator begin() noexcept
    {
        return iterator(data_);
    }

    iterator end() noexcept
    {
        return iterator(data_ + N);
    }
};
```

```
Array<int, 5> arr;
for (auto i : arr)
    std::cout << i;

Array<int, 5>::iterator it = arr.begin();
while (it != arr.end())
    ++it;
```

Адаптеры

```
#include <iterator>
```

reverse_iterator

```
template <class T>
using reverse_iterator = reverse_iterator<Iterator<T>>;

reverse_iterator rbegin() const noexcept
{
    return reverse_iterator(end());
}

reverse_iterator rend() const noexcept
{
    return reverse_iterator(begin());
}
```

back_insert_iterator

Вставляет элемент в конец контейнера (push_back).

```
std::vector<int> v;
std::back_insert_iterator<std::vector<int>> it = std::back_inserter(v);
*it = 5;
++it;
*it = 7;
// v == { 5, 7 }
```

front_insert_iterator

Вставляет элемент в начало контейнера (push_front).

insert_iterator

Вставляет элемент в указанное место (insert).

```
std::set<int> s;
std::insert_iterator<std::set<int>> it = std::inserter(s, s.end());
*s = 3;
```

Операции с итераторами

advance

Переместить итератор на n

```
std::advance(it, 4);
```

distance

Расстояние между двумя итераторами

```
auto n = std::distance(it1, it2);
```

Потоковые итераторы

Позволяют работать с потоком через интерфейс итератора.

ostream_iterator

```
auto it = std::ostream_iterator<int>(std::cout, " ");
*it = 3;
```

istream_iterator

```
auto it = std::istream_iterator<int>(std::cin);
int x = *it;
```

Аллокаторы

Назначение аллокатора - выделять и освобождать память.

malloc и new - аллокаторы.

```
template<class T,
    class Alloc = std::allocator<T>>
class vector
{
};
```

```
template<class T>
class allocator
{
public:
    using value_type = T;
    using pointer = T*;
    using size_type = size_t;

    pointer allocate(size_type count);
    void deallocate(pointer ptr, size_type count);

    size_t max_size() const noexcept;
};
```

std::deque

Интерфейс повторяет интерфейс std::vector, отличие в размещении в памяти - std::vector хранит данные в одном непрерывном куске памяти, std::deque хранит данные в связанных блоках по n элементов.

```
std::vector
[ ][ ][ ][ ][ ][ ][ ]

std::deque
[ ][ ][ ]   [ ][ ][ ]
```

Вставка	Удаление	Поиск	Доступ
O(n)	O(n)	O(n)	O(1)
В конце и начале O(1)	В конце и начале O(1)	В отсортированном O(log n)	

std::forward_list

Связный список, элементы которого храняться в произвольных участках памяти.

```
template <class T>
struct Node
{
    T value_;
    Node<T>* next_;
};

template <class T>
class List
{
    Node<T>* root_;
};
```

```
auto node = root_;
while (node != nullptr)
{
    node = node->next_;
}
```

Вставка	Удаление	Поиск	Доступ
O(1)	O(1)	O(n)	O(n)

Итератор списка не поддерживает произвольный доступ, следовательно алгоритмы STL, которые требуют random access iterator работать со списком не будут, например, std::sort

Нахождение петли в списке

Берем 2 итератора. Первый увеличиваем каждую итерацию на 1, второй на 2. Если итераторы на какой-либо итерации встретились - петля есть, если дошли до конца - петли нет.

std::list

Отличие от односвязного списка - возможность перемещаться в обратном направлении.

```
template <class T>
struct Node
{
    T value_;
    Node<T>* prev_;
    Node<T>* next_;
};
```

Разворот списка

Идем по списку и меняем местами значения prev и next.

Ассоциативные контейнеры

Контейнер позволяющий хранить пары вида (ключ, значение) и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Элементы отсортированы по ключу:

- `set<Key, Compare, Allocator>`
- `map<Key, T, Compare, Allocator>`
- `multiset<Key, Compare, Allocator>`
- `multimap<Key, T, Compare, Allocator>`

Элементы не отсортированы:

- `unordered_set<Key, Hash, KeyEqual, Allocator>`
- `unordered_map<Key, T, Hash, KeyEqual, Allocator>`
- `unordered_multiset<Key, Hash, KeyEqual, Allocator>`
- `unordered_multimap<Key, T, Hash, KeyEqual, Allocator>`

set будем представлять как вырожденный случай map, где ключ равен значению.

В set и map ключи уникальны, в multi версиях контейнеров допускаются наличие значений с одинаковым ключом.

	Вставка	**Удаление**	**Поиск**	**Доступ**
set, map	O(log n)	O(log n)	O(log n)	O(log n)
unordered_set, unordered_map	O(1) или O(n)	O(1) или O(n)	O(1) или O(n)	O(1) или O(n)

```
#include <unordered_map>

std::unordered_map<std::string, size_t> frequencyDictionary;

std::string word;
while (getWord(word))
{
    auto it = frequencyDictionary.find(word);
    if (it == frequencyDictionary.end())
        frequencyDictionary[word] = 1;
    else
        it->second++;
}
```

Контейнеры-адаптеры

Являются обертками над другими контейнерами и предоставляют нужный интерфейс.

- `stack<T, Container = std::deque<T>>`
- `queue<T, Container>`
- `priority_queue<T, Container, Compare>`

std::stack

Реализует интерфейс стека - положить значение в стек, извлечь значение из стека, последний пришел первый вышел (LIFO).

```
#include <stack>

std::stack<int> s;
s.push(3);
s.push(5);
int x = s.top(); // 5
s.pop();
int y = s.top(); // 3
```

```
template<class T,
    class Container = std::deque<T> >
class stack
{
    Container data_;
public:
    using value_type = T;
    using size_type = typename Container::size_type;
    using reference = T&;
    using const_reference = const T&;

    void push(value_type&& value)
    {
        data_.push_back(std::move(value));
    }

    void push(const value_type& value)
    {
        data_.push_back(value);
    }

    template<class... VT>
    void emplace(VT&&... values)
    {
        data_.emplace_back(std::forward<VT>(values)...);
    }

    bool empty() const
    {
        return data_.empty();
```

```
    }

    size_type size() const
    {
        return data_.size();
    }

    reference top()
    {
        return data_.back();
    }

    const_reference top() const
    {
        return data_.back();
    }

    void pop()
    {
        data_.pop_back();
    }
};
```

std::queue

Реализует интерфейс очереди - положить значение в стек, извлечь первое значение из стека, первый пришел первый вышел (FIFO).

```
#include <queue>

template<
    class T,
    class Container = std::deque<T>>
class queue;
```

```
void push(const value_type& value);
void push(value_type&& value);
```

```
reference front();
const_reference front() const;
```

```
void pop();
```

std::priority_queue

Отличие от queue - за O(1) можно извлечь элемент наиболее полно удовлетворяющий условию.

```
#include <queue>

template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>>
class priority_queue;
```

```
struct Packet
{
    int priority_;
    std::string payload_;
};

auto PriorityComparator =
    [](const Packet& x, const Packet& y)
    {
        return x.priority_ > y.priority_;
    };

using PacketQueue = std::priority_queue
    <
        Packet,
        std::vector<Packet>,
        decltype(PriorityComparator)
    >;

PacketQueue incoming(PriorityComparator);
```

Библиотека алгоритмов STL

1. Не изменяющие последовательные алгоритмы
2. Изменяющие последовательные алгоритмы
3. Алгоритмы сортировки
4. Бинарные алгоритмы поиска
5. Алгоритмы слияния
6. Кучи
7. Операции отношений

```
#include <algorithm>
```

Не изменяющие последовательные алгоритмы

Не изменяют содержимое последовательности и решают задачи поиска, подсчета элементов, установления равенства последовательностей.

adjacent_find

Возвращает итератор, указывающий на первую пару одинаковых объектов, если такой пары нет, то итератор - end.

```
std::vector<int> v { 1, 2, 3, 3, 4 };
auto i = std::adjacent_find(v.begin(), v.end());
// *i == 3
```

all_of

Проверяет, что все элементы последовательности удовлетворяют предикату.

```
std::vector<int> v { 1, 2, 3, 4 };
if (std::all_of(v.begin(), v.end(), [](int x) { return x < 5; }))
    std::cout << "all elements are less than 5";
```

any_of

Проверяет, что хоть один элемент последовательности удовлетворяет предикату.

none_of

Проверяет, что все элементы последовательности не удовлетворяют предикату.

count, count_if

Возвращает количество элементов, значение которых равно value или удовлетворяет предикату.

```
std::vector<int> v { 3, 2, 3, 4 };
auto n = std::count(v.begin(), v.end(), 3);
// n == 2
```

equal

Проверяет, что две последовательности идентичны.

```
bool isPalindrome(const std::string& s)
{
    auto middle = s.begin() + s.size() / 2;
    return std::equal(s.begin(), mid, s.rbegin());
}

isPalindrome("level"); // true
```

Есть версия принимающая предикат.

find, find_if, find_if_not

Находит первый элемент последовательности удовлетворяющий условию.

find_end

Находит последний элемент последовательности удовлетворяющий условию.

find_first_of

Ищет в первой последовательности первое вхождение любого элемента из второй последовательности.

```
std::vector<int> v { 0, 2, 3, 25, 5 };
std::vector<int> t { 3, 19, 10, 2 };

auto result = std::find_first_of(
    v.begin(), v.end(),
    t.begin(), t.end());

if (result == v.end())
    std::cout << "no matches found\n";
else
    std::cout << "found a match at "
        << std::distance(v.begin(), result) << "\n";
}

// found a match at 1
```

for_each

Вызывает функцию с каждым элементом последовательности.

```
std::vector<int> v { 3, 2, 3, 4 };
auto print = [](int x) { std::cout << x; };
std::for_each(v.begin(), v.end(), print);
```

search

Ищет вхождение одной последовательности в другую последовательность.

search_n

Возвращает итератор на начало последовательности из n одинкаовых элементов или end.

```
auto it = search_n(data.begin(), data.end(), howMany, value);
```

mismatch

Возвращает пару итераторов на первое несовпадение элементов двух последовательностей.

```
std::vector<int> x { 1, 2 };
std::vector<int> y { 1, 2, 3, 4 };
auto pair = std::mismatch(x.begin(), x.end(), y.begin());
// pair.first == x.end()
// pair.second = y.begin() + 2
```

Модифицирующие последовательные алгоритмы

Изменяют содержимое последовательности, решают задачи копирования, замены, удаления, перестановки значений и т.д.

copy, copy_if, copy_n

Копируют диапазон последовательности в новое место.

```
std::vector<int> data { 1, 2, 3, 4 };
std::copy(data.begin(), data.end(),
    std::ostream_iterator<int>(std::cout, " "));
```

```
std::vector<int> data { 1, 2, 3, 4 };
std::vector<int> out;
std::copy(data.begin(), data.end(), std::back_inserter(out));
```

```
char* source = ...;
size_t size = 1024;
char* destination = ...;
std::copy(source, source + size, destination);
```

copy_backward

Аналогично copy, но в обратном порядке.

move, move_backward

Аналогично copy, но вместо копирования диапазона используется перемещение.

fill, fill_n

Заполнение диапазона значениями.

```
std::vector<int> data { 1, 2, 3, 4 };
std::fill(data.begin(), data.end(), 0);
```

generate, generate_n

Заполнение сгенерированными значениями.

```
std::vector<int> randomNumbers;
auto iter = std::back_inserter(randomNumbers);
std::generate_n(iter, 100, std::rand);
```

remove, remove_if

Удаляет элементы удовлетворяющие критерию. Если быть точным данные алгоритмы ничего не удаляют, просто изменяют последовательность так, чтобы удаляемые элементы были в конце и возвращают итератор на первый элемент.

```
std::string str = "Text\t with\t \ttabs";
auto from = std::remove_if(
    str.begin(), str.end(),
    [](char x) { return x == '\t'; })
// Text with tabs\t\t\t
str.erase(from, str.end());
// Text with tabs
```

remove_copy, remove_copy_if

То же, что и remove, но то, что не должно удаляться копируется в новое место.

```
std::string str = "Text with   spaces";
std::remove_copy(str.begin(), str.end(),
    std::ostream_iterator<char>(std::cout), ' ');
```

```
Textwithspaces
```

replace, replace_if

Заменяет элементы удовлетворяющие условию в последовательности.

```
std::string str = "Text\twith\ttabs";
std::replace_if(str.begin(), str.end(),
    [](char x) { return x == '\t'; }, ' ');
```

reverse

Поворачивает элементы последовательности задом наперед.

swap

Меняет два элемента местами.

```
int x = 3;
int y = 5;
std::swap(x, y);
```

iter_swap

Меняет два элемента на которые указывают итераторы местами.

swap_ranges

Меняет местами два диапазона последовательностей.

shuffle

Перемешивает диапазон последовательности.

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::random_device rd;
std::mt19937 gen(rd());

std::shuffle(v.begin(), v.end(), gen);
```

unique

Удаляет (аналогично remove) дубликаты в последовательности, последовательность должна быть отсортирована.

```
std::vector<int> v { 1, 1, 2, 3, 3 };
const auto from = std::unique(v.begin(), v.end());
// 1 2 3 1 3
v.erase(from, v.end());
// 1 2 3
```

Алгоритмы сортировки

is_sorted

Проверяет упорядочена ли последовательность.

```
std::vector<int> v = { 1, 2, 3 };
const bool isSoreted =
    std::is_sorted(v.begin(), v.end());
// true
```

sort

Сортирует последовательность.

```
std::vector<int> v = { 2, 3, 1 };
std::sort(v.begin(), v.end(),
    [](int x, int y) { return x > y; });
// 3 2 1
```

> Сложность O(n * log n)

partial_sort

Сортирует часть последовательности (TOP-N).

```
std::array<int, 10> s { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
std::partial_sort(s.begin(), s.begin() + 3, s.end());
```

```
0 1 2 7 8 6 5 9 4 3
```

> Сложность O((last-first) * log (middle-first))

stable_sort

Сортирует последовательность, если два объекта равны, их порядок не изменится.

> Сложность O(n * log² n)

nth_element

Помещает элемент в позицию n, которую он занимал бы после сортировки всего диапазона.

```
std::vector<int> v { 3, 1, 4, 5, 2 };
const auto medianIndex = v.size() / 2;
std::nth_element(v.begin(), v.begin() + medianIndex, v.end());
const auto median = v[medianIndex];
// 3
```

> Сложность O(n)

Алгоритмы бинарного поиска

Последовательности к которым применяются алгоритмы должны быть отсортированы.

binary_search

Поиск по отсортированной последовательности.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
bool has2 = std::binary_search(v.begin(), v.end(), 2);
// true
```

lower_bound

Возвращает итератор, указывающий на первый элемент, который не меньше, чем value.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
//                     ^
auto it = std::lower_bound(v.begin(), v.end(), 2);
```

upper_bound

Возвращает итератор, указывающий на первый элемент, который больше, чем value.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
//                        ^
auto it = std::upper_bound(v.begin(), v.end(), 2);
```

equal_range

Возвращает такую пару итераторов, что элемент на который указывает первый итератор не меньше value, а элемент на который указывает второй итератор больше value.

```
std::vector<int> v { 1, 2, 3, 4, 5 };
//                     ^  ^
auto pair = std::equal_range(v.begin(), v.end(), 2);
```

Практическая работа

Написать свой контейнер Vector аналогичный std::vector, аллокатор и итератор произвольного доступа для него. Из поддерживаемых методов достаточно operator[], push_back, pop_back, empty, size, clear, begin, end, rbegin, rend, resize, reserve.

Чтобы тесты проходили, классы должны иметь такие имена:

```
template <class T>
class Allocator
{
};

template <class T>
class Iterator
{
};

template <class T, class Alloc = Allocator<T>>
class Vector
{
public:
    using iterator = Iterator<T>;

private:
    Alloc alloc_;
};
```

Интерфейс аллокатора и итератора смотрите в документации.

Как безопасно увеличить буфер

```
template <class T>
class Buffer
{
public:
    explicit Buffer(size_t initialSize = 1024)
        : data_(new T[initialSize])
        , size_(initialSize)
    {
    }

    void resize(size_t newSize)
    {
        if (size_ < newSize)
        {
            auto newData = std::make_unique<T[]>(newSize);
            std::copy(data_.get(), data_.get() + size_, newData.get());
            data_.swap(newData);
            size_ = newSize;
            return;
        }

        assert(!"not implemented yet");
    }

private:
    std::unique_ptr<T[]> data_;
    size_t size_;
};
```

После лекции я с ребятами написал в качестве примера итератор, который итерирует последовательность по четным числам, вот код:

```
#include <iostream>
#include <set>
#include <vector>

template <class Iter>
class OddIterator
    : public std::iterator<std::forward_iterator_tag, typename
Iter::value_type>
{
    Iter current_;
    Iter end_;

    void findNext()
    {
        while (current_ != end_)
        {
            if (*current_ % 2 == 0)
                return;
            ++current_;
        }
    }

public:
    OddIterator(Iter&& begin, Iter&& end)
        : current_(std::move(begin))
        , end_(std::move(end))
    {
        findNext();
    }

    bool operator==(const OddIterator& other) const
    {
        return current_ == other.current_;
    }

    bool operator!=(const OddIterator& other) const
    {
        return !(*this == other);
    }

    void operator++()
    {
        if (current_ != end_)
        {
            ++current_;
            findNext();
        }
```

```
    }

    int operator*() const
    {
        return *current_;
    }
};

template <class Container>
OddIterator<typename Container::const_iterator> getBegin(const Container& data)
{
    return OddIterator<typename Container::const_iterator>(data.cbegin(),
data.cend());
}

template <class Container>
OddIterator<typename Container::const_iterator> getEnd(const Container& data)
{
    return OddIterator<typename Container::const_iterator>(data.cend(),
data.cend());
}

int main()
{
    std::vector<int> data1 = { 9, 8, 1, 3, 4, 5, 6 };
    std::for_each(getBegin(data1), getEnd(data1), [](int x) { std::cout << x <<
'\n'; });

    std::cout << '\n';

    std::set<int> data2(data1.begin(), data1.end());
    std::for_each(getBegin(data2), getEnd(data2), [](int x) { std::cout << x <<
'\n'; });

    return 0;
}
```

EOF

std::function

Обёртка функции общего назначения. Экземпляры std::function могут хранить и ссылаться на любой вызываемый объект - функцию, лямбда-выражение, привязку выражения или другой объект-функцию. Экземпляры std::function можно хранить в переменных, контейнерах, передавать в функции.

```cpp
#include <functional>

using Function = std::function<void (int)>;

void doSomething(Function f)
{
    f(10);
}

void foo(int x) {}

Function f1 = foo;
Function f2 = [](int x) {};

struct A
{
    void operator()(int x) {}
};

Function f3 = A();

struct B
{
    void bar(int x) {}
    static void foo(int x) {}
};

Function f4 = &B::foo;

B b;
Function f5 = std::bind(
    &B::bar, &b, std::placeholders::_1);

std::vector<Function> functions =
    { f1, f2, f3, f4, f5 };
for (auto& f : functions)
    doSomething(f);
```

std::bind

Позволяет получить функциональный объект с требуемым интерфейсом.

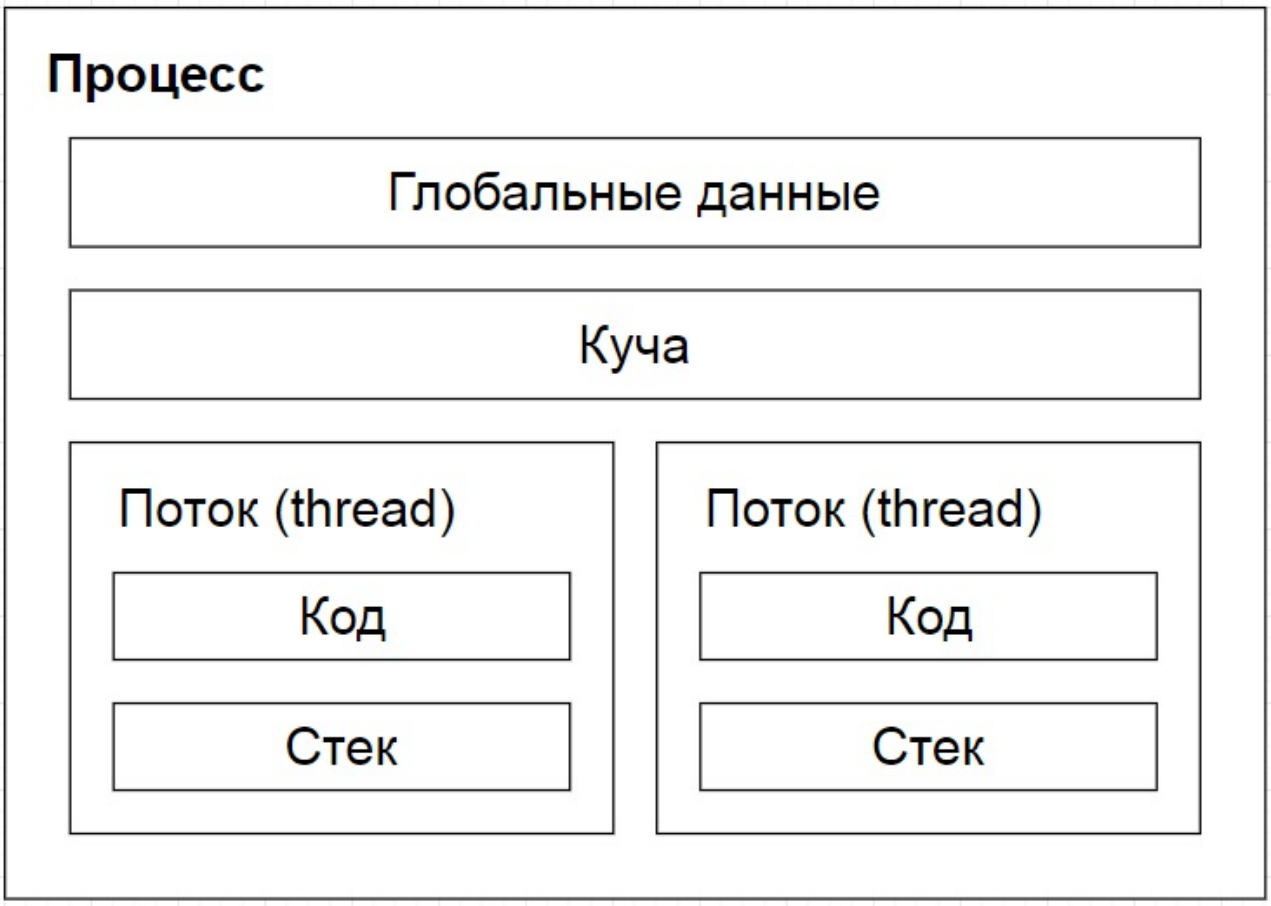
```
using Generator = std::function<int ()>;

void prepareData(Generator gen) { ... }

int monotonic(int initial) { ... }
int random(const std::string& device) { ... }

Generator gen1 = std::bind(monotonic, 100);
prepareData(gen1);

Generator gen2 = std::bind(random, "/dev/random");
prepareData(gen2);
```

```
if (std::all_of(v.begin(), v.end(),
    [](int x) { return x < 5; }))
{
    ...
}

bool lessThan(int v, int max)
{
    return v < max;
}

auto lessThan3 =
    std::bind(lessThan, std::placeholders::_1, 3);
if (std::all_of(v.begin(), v.end(), lessThan3))
{
    ...
}
```

```
struct Robot
{
    Robot() = default;
    Robot(const Robot&) = delete;
    Robot& operator=(const Robot&) = delete;
};

using Command = std::function<void ()>;

enum class Direction
{
    Left,
    Right,
    Up,
    Down
};

void move(Robot& robot, Direction dir) { ... }
void fire(Robot& robot) { ... }

Robot robot;

std::vector<Command> program;

program.push_back(
    std::bind(move, robot, Direction::Left)); // error
program.push_back(
    std::bind(fire, robot)); // error
```

```
program.push_back(
    std::bind(move, std::ref(robot), Direction::Left));
program.push_back(
    std::bind(fire, std::ref(robot)));
```

Многопоточность (multithreading)



Многозадачность – возможность параллельной (или псевдопараллельной) обработки нескольких задач.

1. Многозадачность основанная на прерываниях планировщика
2. Кооперативная многозадачность – выполняемый код должен уступать процессорное время для других

Современные компьютеры – сложные системы

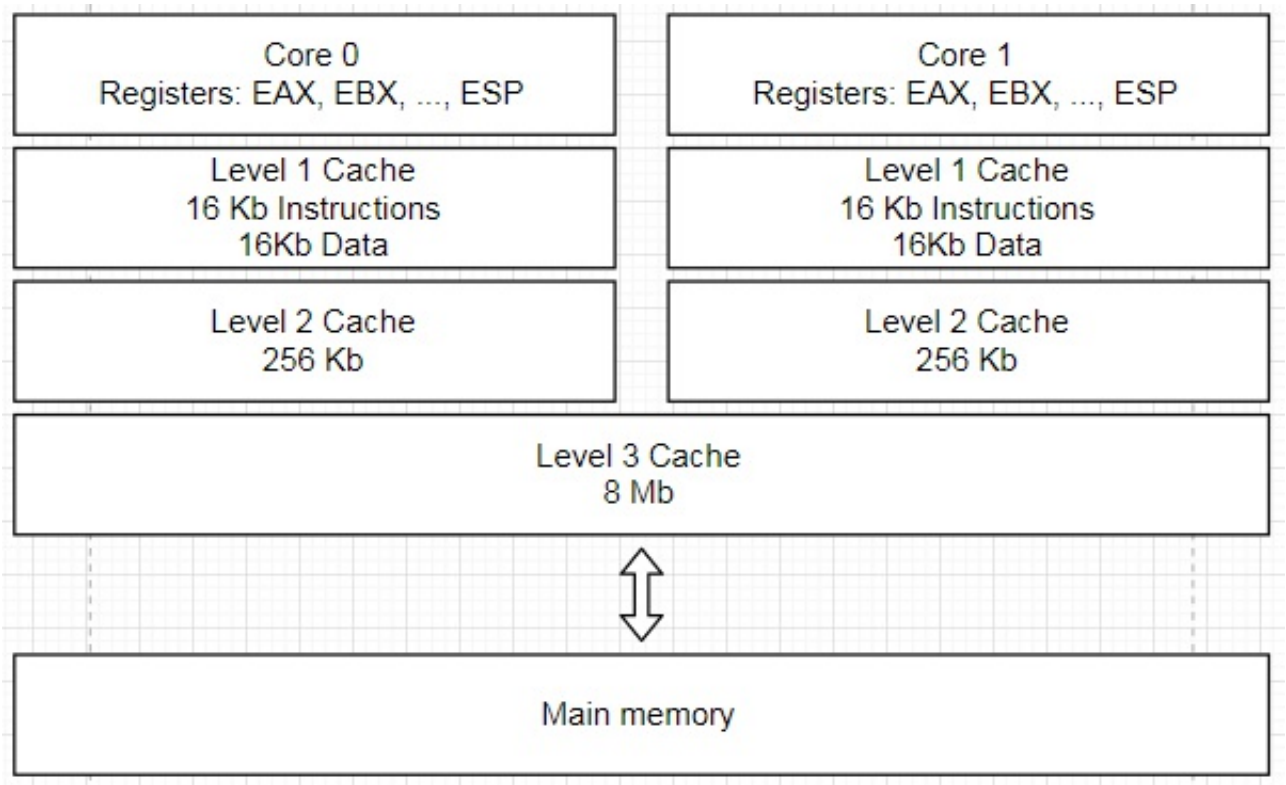
Компилятор

C++ почти ничего не знает о многопоточности и при оптимизизациях не учитывает фактор многопоточности.

```
bool shutdown = false;

void thread1()
{
    shutdown = false;
    while (!shutdown)
    {
        // Может выполняться вечно
    }
}

void thread2()
{
    shutdown = true;
}
```

```
volatile bool shutdown = false;
// Поможет лишь частично!
```

Процессор



```
bool ready = false;
int data = 0;

int foo() { return 5; }

void produce()
{
    data = foo();
    ready = true;
}

void consume()
{
    while (!ready) ;
    assert(data == 5); // не всегда
}
```

Вожможный пример выполнения кода процессором:

```
void produce()
{
    // data = foo();
    // Это долго, выполню пока это:
    ready = true;
    // А теперь остальное:
    data = foo();
}

void consume()
{
    while (!ready) // ждем
        ;
    assert(data == 5); // не всегда
}
```

Исправляем produce:

```
void produce()
{
    data = foo();
    // Инструкция запрещающая процессору
    // изменять порядок выполнения
    ------------------
    ready = true;
}

void consume()
{
    // Этих данных у меня еще нет
    // while (!ready)
    //    ;
    // Поэтому, поскольку данные не
    // взаимосвязаны, можно выполнить
    assert(data == 5); // не всегда
    // А теперь
    while (!ready) // ждем
        ;
}
```

Исправляем consume:

```
void consume()
{
    while (!ready) // ждем
        ;
    // Инструкция запрещающая процессору
    // изменять порядок выполнения
    ------------------
    assert(data == 5);
}
```

Барьер - инструкция состоящая из указания двух типов операций работы с памятью:

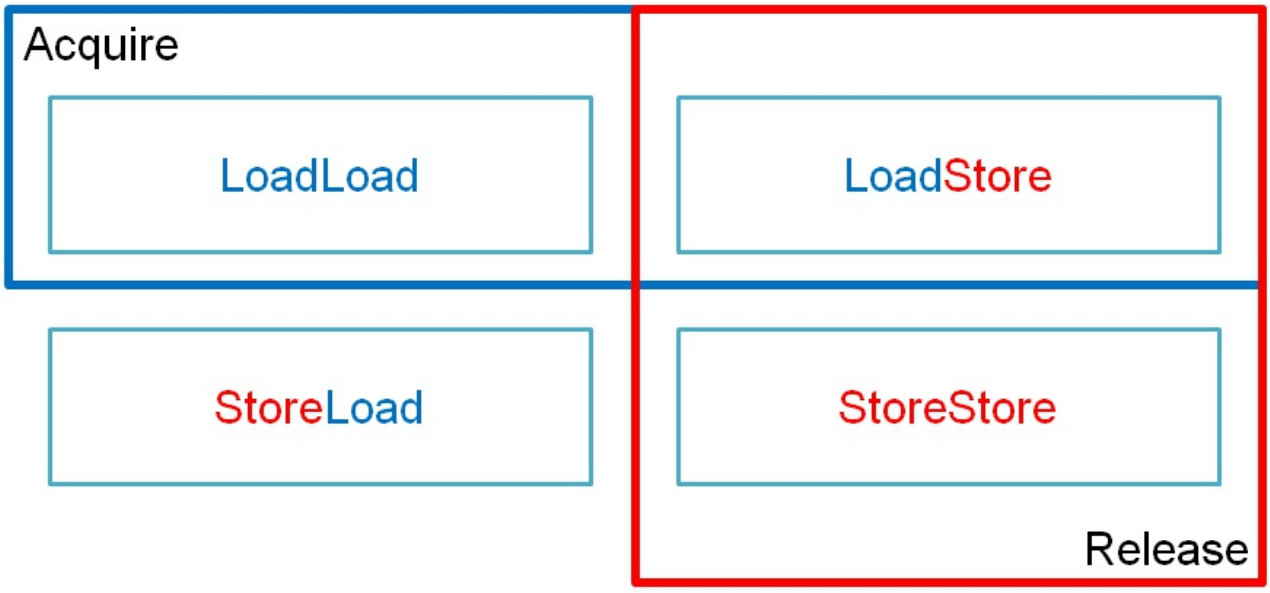
```
X_Y
```

Барьер гарантирует, что до барьера все операции работы с памятью типа X будут выполнены, а операции типа Y после барьера не начнут выполняться.

Операций работы с памятью две:

1. Чтение (Load)
2. Запись (Store)

Следовательно барьеров может быть 4:



Acquire

Acquire гарантирует, что все операции после барьера будут начаты после того, как будут выполнены все Load-операции до барьера.

Release

Release гарантирует, что все операции до барьера будут выполнены до того, как начнут выполняться Store-операции после барьера.

Барьеры памяти в C++

```
#include <atomic>

void atomic_thread_fence(std::memory_order order);

enum memory_order
{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire, // <-- acquire
    memory_order_release, // <-- release
    memory_order_acq_rel,
    memory_order_seq_cst  // <-- default
};
```

```
void produce()
{
    data = foo();
    // Перед тем, как делать Store-операции
    // завершить все операции до барьера
    atomic_thread_fence(std::memory_order_release);
    ready = true;
}
```

```
produce():
    call    foo()
    mov     DWORD PTR data[rip], eax
    mfence
    mov     BYTE PTR ready[rip], 1
    ret
```

```
void consume() noexcept
{
    while (!ready) ;
    // Не выполнять никаких инструкций, пока
    // не будут выполнены Load-инструкции
    atomic_thread_fence(std::memory_order_acquire);
    int k = data;
}
```

Инструкции amd64 реализующие барьеры:

- LFENCE (load fence)
- SFENCE (store fence)
- MFENCE (memory fence)

mfence выполняется до микросекунды и более!

Для контраста из второй лекции:

```
Compress 1K bytes with Zippy             3,000   ns
```

Атомарные значения

```
std::atomic<T> value;

T load(std::memory_order
    order = std::memory_order_seq_cst) const noexcept;

void store(T value, std::memory_order
    order = std::memory_order_seq_cst) noexcept;
```

```
std::atomic<int> i = 5;

i.store(3);
int j = i.load();

++i;
int k = i;
```

Основная рекомендация

Не разделять изменяемые данные между потоками!

Создание потока

```cpp
#include <thread>

void threadFunction()
{
    ...
}

std::thread t(threadFunction);

t.join(); или t.detach();
```

```cpp
{
    std::thread t(threadFunction);
} // <-- Здесь созданный на стеке t будет уничтожен
```

> Если на момент уничтожения объекта std::thread не был вызван join или detach, то будет вызван std::terminate

> У каждого потока свой стек

std::this_thread

```cpp
// идентификатор потока
const std::thread::id id =
    std::this_thread::get_id();

// указание планировщику снять
// поток с выполнения до следующего раза
std::this_thread::yield();

// усыпить поток на указанное время
std::this_thread::sleep_for(
    std::chrono::seconds(1))
```

> std::thread::id можно сравнить, можно вывести в поток вывода

std::async

```
#include <future>

// запуск в отдельном потоке
std::async(std::launc::async, []() { ... });

// запуск на усмотрение компилятора, может выполнится в том же потоке
std::async(std::launc::deferred, []() { ... });

void doSomething(int x, int y)
{
}

std::async(std::launch::async, doSomething, 5, 7);
```

std::future

Ожидание выполнения асинхронной задачи.

```
std::future<int> f =
    std::async(std::launch::async, []() { return 5 });
...
const int result = f.get();
```

```
auto f =
    std::async(std::launch::async, []() { return 5 });
...
f.wait();
```

```
auto f =
    std::async(std::launch::async, []() { return 5 });

auto status = f.wait_for(std::chrono::seconds(1));

if (status == std::future_status::deferred)
    std::cout << "задача еще не стартовала";
else if (status == std::future_status::timeout)
    std::cout << "результата не дождались";
else if (status == std::future_status::ready)
    std::cout << "все готово";
```

std::promise

Позволяет вернуть результат работы из потока.

```
#include <future>

std::future<int> runTask()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();

    auto task = [](std::promise<int>&& p)
    {
        p.set_value(1);
    };

    std::thread thread(task, std::move(promise));
    thread.detach();

    return future;
}

auto task = runTask();
task.get();
```

Исключения в потоке

```
void foo()
{
    throw std::runtime_error();
}

std::thread t1(foo);
t1.join();
```

В этом случае поток просто завершиться, об исключении мы не узнаем.

```
auto f = std::async(std::launch::async, foo);

try
{
    f.get();
}
catch (const std::runtime_error& error)
{
    // Получили ошибку
}
```

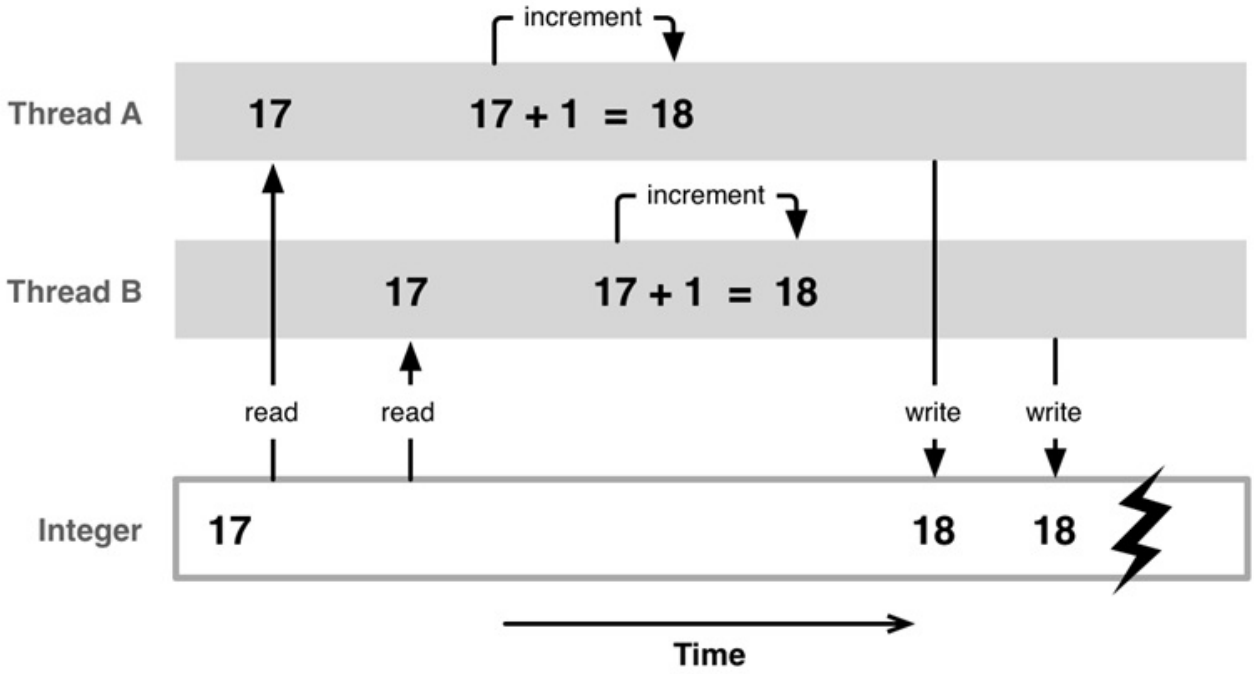
```
auto task = ([](std::promise<int>&& p)
{
    try
    {
        foo();
    }
    catch (...)
    {
        p.set_exception(std::current_exception());
    }
}
```

std::packaged_task

```
std::future<int> runTask()
{
    std::packaged_task<int()> task([]()
    {
        return 1;
    });

    auto future = task.get_future();

    std::thread thread(std::move(task));
    thread.detach();

    return future;
}

auto task = runTask();
task.get();
```

Гонки (race condition)

```
int i = 17;

void plus1()
{
    i += 1;
}

std::thread t1(plus1);
std::thread t2(plus1);

t1.join();
t2.join();

std::cout << i; // ???
```



Средства синхронизации

1. Атомарные операции
2. Спинлоки (spinlock)
3. Семафоры (semaphore)
4. Мютексы (mutex)
5. Условные переменные (condition variable)
6. Критические секции (critical section)
7. Высокоуровневые очереди и планировщики

Спинлоки (spinlock)

База - все блокировки в ядре ОС основаны на спинлоках, которые в свою очередь используют атомарные операции, без этого реализовать безопасное межпроцессорное взаимодействие невозможно.

```
int atomicExchange(int* old, int newValue);

// *lock == 0 - никем не захвачен
void spinlock(volatile int* lock)
{
    while (true)
    {
        if (*lock == 0)
        {
            const int old = atomicExchange(lock, 1);
            if (old == 0)
            {
                return;
            }
        }
    }
}
```

Семафоры (semaphore)

Семафор — это объект, над которым можно выполнить три операции:

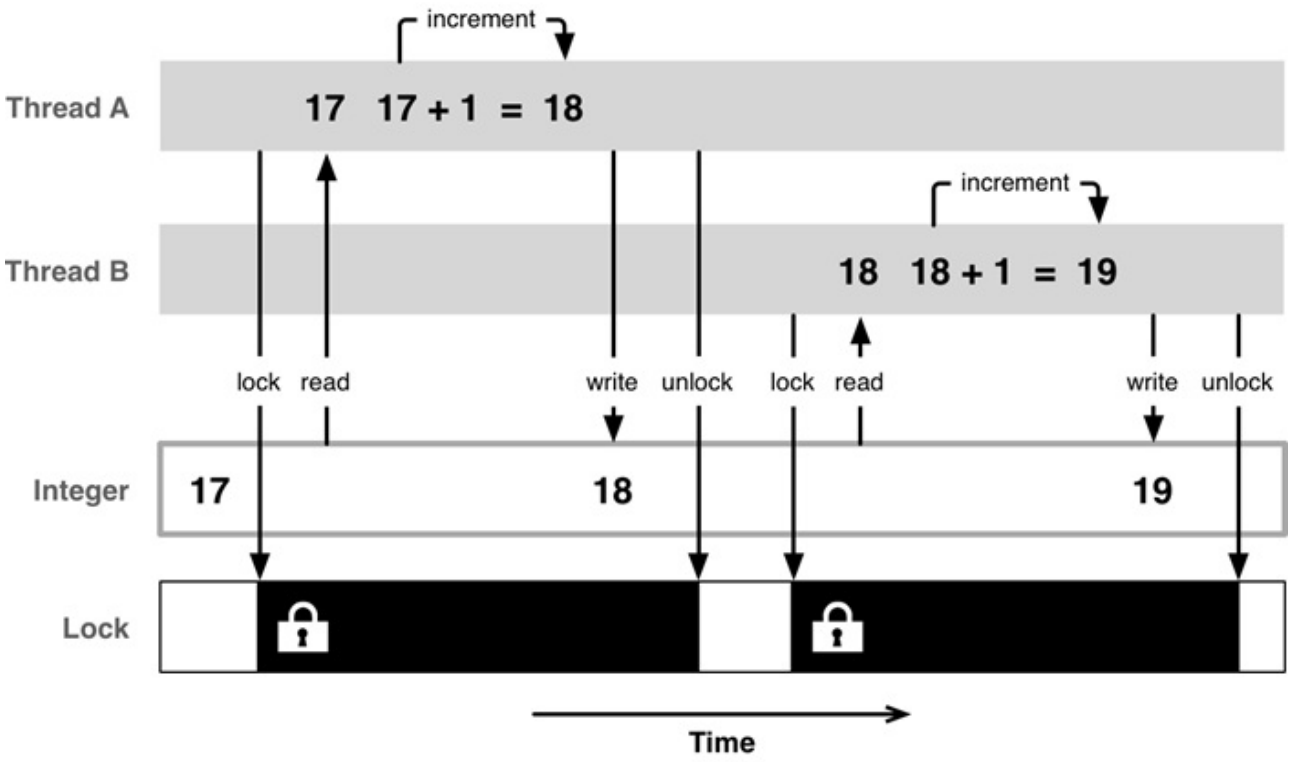
1. Инициализация семафора (задать начальное значение счётчика)
2. Захват семафора (ждать пока счётчик станет больше 0, после этого уменьшить счётчик на единицу)
3. Освобождение семафора (увеличить счётчик на единицу)

Реализуется ОС, описан в POSIX, на базе семафора можно реализовать остальные механизмы синхронизации.

Мютексы (mutex)

```
#include <mutex>

std::mutex m;

m.lock();
m.unlock();

if (m.try_lock())
    m.unlock();
```

```cpp
int i = 0;

std::mutex mutex;

void plus1()
{
    std::lock_guard<std::mutex> lock(mutex);
    i += 1;
}

std::thread t1(plus1);
std::thread t2(plus1);
```



recursive_mutex

```cpp
std::mutex m;
m.lock();
m.lock(); // Неопределенное поведение
```

```cpp
std::recursive_mutex m;
m.lock();
m.lock(); // Ок
```

> Количество lock и unlock должно совпадать

timed_mutex

```cpp
#include <mutex>

std::timed_mutex m;

m.lock();
m.unlock();

if (m.try_lock())
    m.unlock();

auto period = std::chrono::milliseconds(100);
if (m.try_lock_for(period))
    m.unlock();

auto now = std::chrono::steady_clock::now();
m.try_lock_until(now + std::chrono::seconds(1));
```

> steady_clock - monotonic clock

lock_guard<T>

Захват мютекса в конструкторе, освобождение в деструкторе.

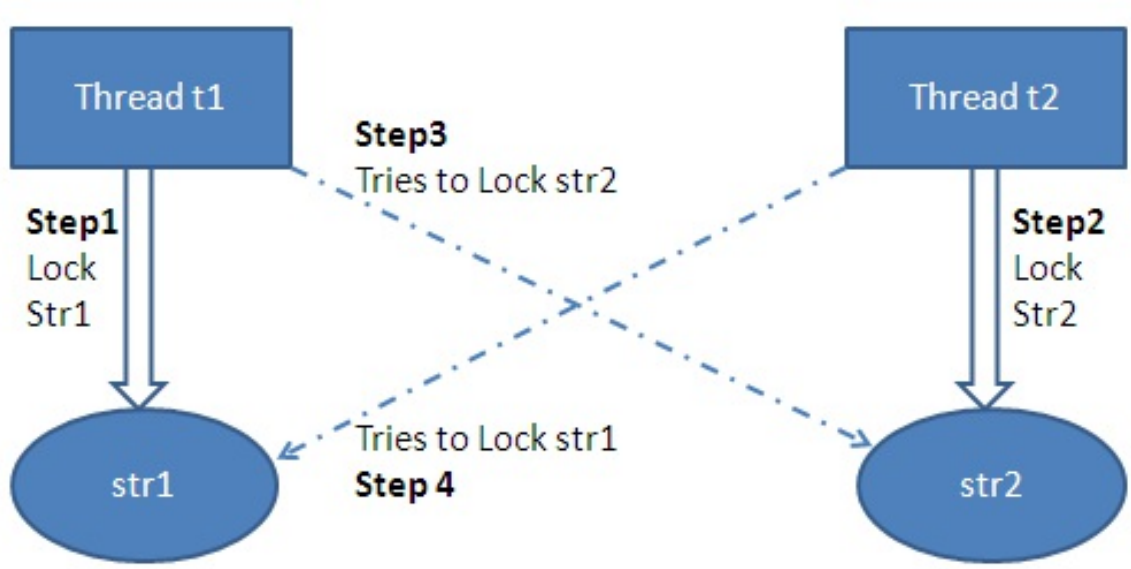
unique_lock<T>

Расширяет поведение lock_guard:

- lock
- try_lock
- try_lock_for
- try_lock_until
- unlock
- swap
- release

Взаимоблокировки (deadlock)

```cpp
std::mutex m1;
std::mutex m2;

void t1() // thread 1
{
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
}

void t2() // thread 2
{
    std::lock_guard<std::mutex> lock1(m2);
    std::lock_guard<std::mutex> lock2(m1);
}
```



Блокировка в одном и том же порядке

```
void t1() // thread 1
{
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
}

void t2() // thread 2
{
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
}
```

Одновременная блокировка

Иногда дать гарантию на блокировку в одном и том же порядке дать нельзя.

```
class Data
{
    std::mutex m_;
public:
    void apply(const Data& data)
    {
        std::lock_guard<std::mutex> lock1(m_);
        std::lock_guard<std::mutex> lock2(data.m_);
        ...
    }
};

Data d1;
Data d2;

d1.apply(d2); // thread 1
d2.apply(d1); // thread 2
```

```
void apply(const Data& data)
{
    using Lock = std::unique_lock<std::mutex>;
    Lock lock1(m_, std::defer_lock);
    Lock lock2(data.m_, std::defer_lock);
    std::lock(lock1, lock2);
    ...
}
```

Условные переменные (condition_variable)

Средство для обеспечения коммуникации потоков.

```
Data data;

std::mutex m;
std::condition_variable dataReady;

void consumer() // thread 1
{
    std::unique_lock<std::mutex> lock(m);
    while (!data.ready())
        dataReady.wait(lock);
}

void producer() // thread 2
{
    {
        std::lock_guard<std::mutex> lock(m);
        data.prepare();
    }
    dataReady.notify_one();
}
```

```cpp
#include <condition_variable>

std::mutex m;
std::unique_lock<std::mutex> lock(m);

std::condition_variable c;

c.wait(lock);

c.wait(lock, predicate);
// while (!predicate())
// {
//     wait(lock);
// }

// wait_for
// wait_until

// В другом потоке
c.notify_one();
c.notify_all();
```

Семафор на базе мютекса

```
class Semaphore
{
    std::mutex mutex_;
    std::condition_variable condition_;
    int counter_;
public:
    explicit Semaphore(int initialValue = 1)
        : counter_(initialValue)
    {
    }

    void enter()
    {
        std::unique_lock<std::mutex> lock(mutex_);
        condition_.wait(lock,
            [this]()
            {
                return counter_ > 0;
            });
        --counter_;
    }

    void leave()
    {
        std::unique_lock<std::mutex> lock(mutex_);
        ++counter_;
        condition_.notify_one();
    }
};
```

```
Semaphore s;

void t1() // thread 1
{
    s.enter();
    s.leave();
}
```

Практическая часть

Ping pong

Классическая задача.

Два потока по очереди выводят в консоль сообщение. Первый выводит ping, второй выводит pong. Количество ping и pong - по 500 000 каждый.

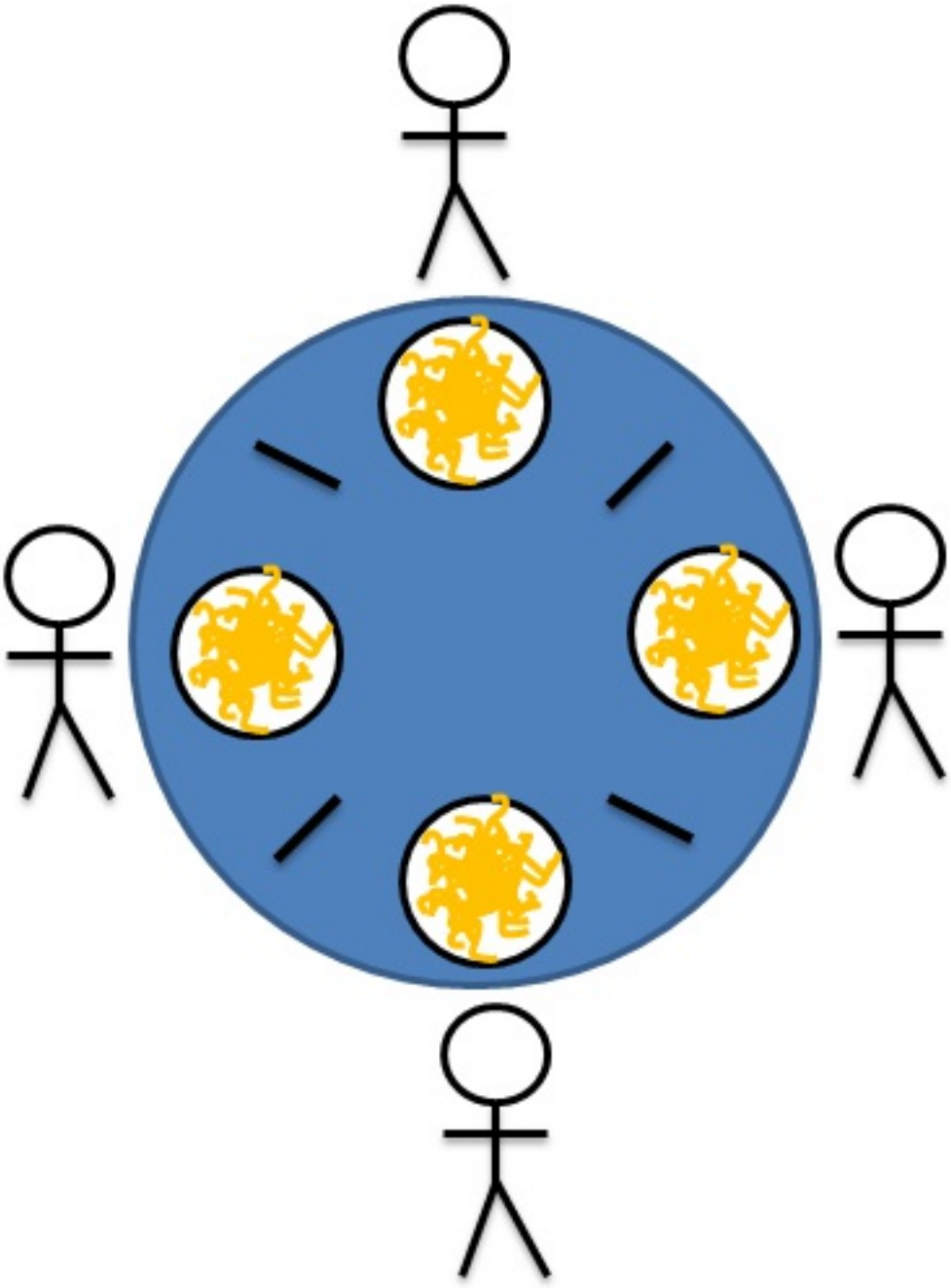
Вывод:

```
ping
pong
ping
pong
ping
pong
…
```

EOF

Задача об обедающих философах

n философов сидят за столом, перед каждым тарелка с едой, между тарелками лежит по 1 палочке. Чтобы поесть, надо взять 2 палочки, то есть философ должен взять палочку слева и справа от своей тарелки.



1. Философы не вежливые

Философ взял правую палочку и не отдает, левую палочку взял сосед и тоже не отдает - произошла взаимная блокировка (deadlock).

2. Философы вежливые

Философ взял правую палочку, посмотрел, что левая занята соседом, положил правую, подумал о вечном, повторил с начала. Остальные поступают аналогично. Если о вечном все думают одинаковое время, то никто из них не сможет поесть (livelock).

3. Философы слишком быстро едят

Поэтому палочка либо слева, либо справа все время занята и философ голодает (starvation).

Решение задачи

1. Иерархия ресурсов

Надо пронумеровать палочки, далее философы берут палочку с наименьшим номером, затем с наибольшим, возращают палочки в обратном порядке. Если n - 1 философов взяли палочке, то останется одна с наибольшим номером, поэтому ее последний философ взять не сможет. Один из взявших палочку берет палочку с наибольшим номером, затем возвращает ее, позволяя поесть следующему.

2. Официант

По запросу выдает палочки или предлагает подождать, если все палочки заняты.

Проблема спящего парикмахера

Цель - парикмахер должен работать когда клиенты есть и спать, когда их нет. Клиент прийдя в парикмахерскую, если парикмахер свободен стрижется, если занят, то идет в приемную и там садится на стул и ждет, если свободных стульев нет, то уходит.

Producer–consumer problem - частный случай этой задачи.

```
struct Client {};

std::mutex barbershop;
std::condition_variable hasClient;

const size_t ChairsNum = 5;

std::queue<Client> clients;

bool clientCame(Client c)
{
    {
        std::unique_lock<std::mutex> lock(barbershop);

        if (clients.size() == ChairsNum)
            return false;

        clients.push(c);
    }

    hasClient.notify_one();

    return true;
}

void barberThread()
{
    while (true)
    {
        Client c;
        {
            std::unique_lock<std::mutex> lock(barbershop);
            while (clients.empty())
            {
                hasClient.wait(lock);
            }
            c = clients.front();
            clients.pop();
        }
        trim(c);
    }
}
```

Задача о читателях-писателях

Есть область памяти, позволяющая чтение и запись. Несколько потоков имеют к ней доступ, при этом одновременно могут читать сколько угодно потоков, но писать — только один. Как обеспечить такой режим доступа?

1. Приоритет читателя

Пока память открыта на чтение, давать читателям беспрепятственный доступ. Писатели могут ждать сколько угодно.

2. Приоритет писателя

Как только появился хоть один писатель, читателей больше не пускать. При этом читатели могут простаивать.

3. Одинаковый приоритет

Независимо от действий других потоков, читатель или писатель должен пройти барьер за конечное время.

Стратегии 1 и 2 чреваты голоданием потоков.

Голодание (starvation) — это более высокоуровневая проблема, чем гонки и взаимоблокировки. Материал для любознательных: [стратегии планирования задач (http://en.wikipedia.org/wiki/Category:Scheduling_algorithms)](http://en.wikipedia.org/wiki/Category:Scheduling_algorithms)

shared_mutex (C++17, boost)

```
#include <boost/thread/shared_mutex.hpp>
#include <boost/thread/locks.hpp>

boost::shared_mutex mutex;

void reader()
{
    boost::shared_lock<boost::shared_mutex> lock(mutex);
    // блокируется если есть unique_lock
    // не блокируется, если есть другие shared_lock
}

void writer()
{
    boost::unique_lock<boost::shared_mutex> lock(mutex);
    // получить эксклюзивный доступ на общих условиях
    // голодания не будет
}

void conditionalWriter()
{
    boost::upgrade_lock<boost::shared_mutex> lock(mutex);
    // блокируется если есть unique_lock
    // не блокируется, если есть другие shared_lock
    if (need_to_write)
    {
        boost::upgrade_to_unique_lock<boost::shared_mutex>
            uniqueLock(lock);
        // получить эксклюзивный доступ
        // блокируется если есть другие shared_lock
        // при высокой конкуренции на чтение может
        // начаться голодание (starvation)
    }
}
```

std::call_once

Проинициализировать что-либо потокобезопасно один раз.

```
std::unique_ptr<Display> display;

void print(const std::string& message)
{
    if (!display)
        display.reset(new Display());
    display->print(message);
}
```

```
std::unique_ptr<Display> display;

static std::once_flag displayInitFlag;

void print(const std::string& message)
{
    std::call_once(displayInitFlag,
        []() { display.reset(new Display()); });
    display->print(message);
}
```

thread_local

Хранилище уровня потока (c++11).

- Создается когда запускается поток
- Уничтожается когда поток завершает работу
- Для каждого потока свое

```
static thread_local std::map<std::string, int> threadCache;
```

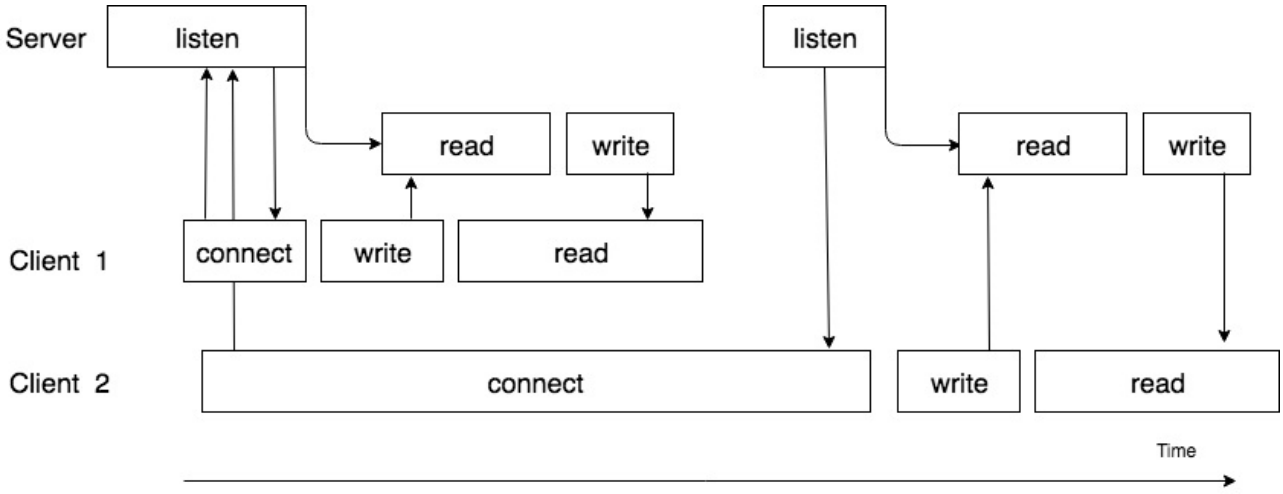
Потокобезопасные интерфейсы

```
template <class T>
class queue
{
    bool empty() const;
    T pop();
};

queue<Task> tasks;

if (!tasks.empty())
    process(tasks.pop());
```

```cpp
template <class T>
class ThreadSafeQueue
{
    bool tryPop(T& value)
    {
        std::lock_guard<std::mutex> lock(mutex);
        if (data_.empty())
            return false;
        value = data_.front();
        data_.pop();
        return true;
    }
};

ThreadSafeQueue<Task> tasks;

Task task;
if (tasks.tryPop(task))
    process(task);
```

Асинхронность

```cpp
Server srv("localhost", 8000);

while (true)
{
    auto connection = srv.listen();
    while (connection.good())
    {
        auto data = connection.read();
        connection.write(data);
    }
}
```



Асинхронность с колбеками

```cpp
#include <boost/asio.hpp>
```

```
class server
{
    boost::asio::ip::tcp::acceptor acceptor_;
public:
    server(
        boost::asio::io_context& io_context,
        short port)
        : acceptor_(
            io_context,
            boost::asio::ip::tcp::endpoint(
                boost::asio::ip::tcp::v4(), port))
    {
        do_accept();
    }

private:
    void do_accept()
    {
        acceptor_.async_accept(
            [this](
                boost::system::error_code ec,
                boost::asio::ip::tcp::socket socket)
        {
            if (!ec)
            {
                std::make_shared<session>(
                    std::move(socket))->start();
            }

            do_accept();
        });
    }
};
```

```
class session
    : public std::enable_shared_from_this<session>
{
    tcp::socket socket_;
    const size_t max_length = 1024;
    char data_[max_length];
public:
    session(boost::asio::ip::tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }

private:
    void do_read()
    {
        auto self(shared_from_this());
        socket_.async_read_some(
            boost::asio::buffer(data_, max_length),
            [this, self](
                boost::system::error_code ec,
                std::size_t length)
        {
            if (!ec)
            {
                do_write(length);
            }
        });
    }

    void do_write(std::size_t length)
    {
        auto self(shared_from_this());
        boost::asio::async_write(
            socket_,
            boost::asio::buffer(data_, length),
            [this, self](
                boost::system::error_code ec,
                std::size_t /*length*/)
        {
            if (!ec)
            {
                do_read();
            }
        });
    }
};
```

```
boost::asio::io_context io_context;
server s(io_context, 4000);
io_context.run();
```

Асинхронность с корутинами

```
namespace this_coro =
    boost::asio::experimental::this_coro;

template <typename T>
using awaitable = boost::asio::experimental::awaitable
    <
        T,
        boost::asio::io_context::executor_type
    >;

awaitable<void> listener()
{
    auto executor = co_await this_coro::executor();
    auto token = co_await this_coro::token();

    boost::asio::ip::tcp::acceptor acceptor(
        executor.context(), { tcp::v4(), 4000 });
    while (true)
    {
        tcp::socket socket =
            co_await acceptor.async_accept(token);
        boost::asio::experimental::co_spawn(executor,
            [socket = std::move(socket)]() mutable
            {
                return echo(std::move(socket));
            },
            boost::asio::experimental::detached);
    }
}
```

```
awaitable<void> echo(tcp::socket socket)
{
    auto token = co_await this_coro::token();

    char data[1024];
    while (true)
    {
        size_t n = co_await socket.async_read_some(
            boost::asio::buffer(data), token);

        co_await boost::asio::async_write(
            socket,
            boost::asio::buffer(data, n),
            token);
    }
}
```

```
boost::asio::io_context io_context;
boost::asio::experimental::co_spawn(
    io_context,
    listener,
    boost::asio::experimental::detached);
io_context.run();
```

Арифметика

Целые отрицательные числа

```
[n][6][5][4][3][2][1][0]
```

Диапазон

Верхняя граница:

2^7 = 128

[0, 127]

Нижняя граница:

-(2^7 + 1)

[-128, 0]

```
  -1 == 0b11111111
-128 == 0b10000000
```

Переполнение

```
uint8_t a = 255;
++a; // overflow
assert(a == 0);
--a; // underflow
assert(a == 255);
```

```
int8_t a = 127;
++a;
// ??? - UB
```

Иллюстрация неопределенного поведения:

```
#include <iostream>

int main()
{
    for(int i = 0; i < 10; ++i)
    {
        auto val = i * 1000000000;
        std::cout << i << ' ' << val << std::endl;
    }
}
```

```
0 0
1 1000000000
2 2000000000
3 -1294967296
4 -294967296
5 705032704
6 1705032704
7 -1589934592
8 -589934592
9 410065408
10 1410065408
11 -1884901888
12 -884901888
...
```

> Проверено на gcc -O2

Приведение знакового к беззнаковому

```
int8_t a = -1;
uint32_t b = a;
assert(b == 4294967295);
```

> Результат копирования знакового бита

Сравнение знакового и беззнакового

```
uint32_t a = 10;
int32_t b = -1;
assert(a < b); // !!!
```

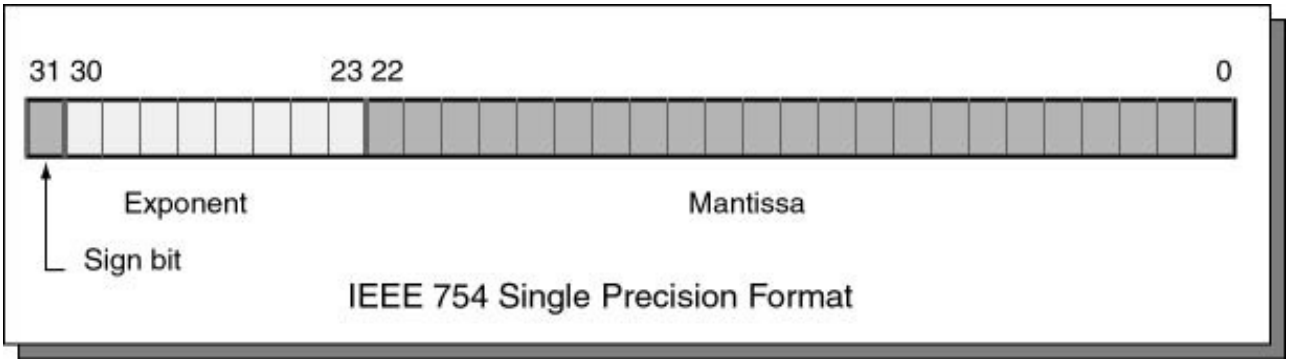
> Если один из аргументов беззнаковый, то он неявно приводится к знаковому.

Плавающая точка

```
float f = 1.234e6;
```

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{\overbrace{-4}^{\text{exponent}}}}_{\text{base}}$$

Стандарт IEEE 754



Нормализованная форма

Такая форма, в которой мантисса без учёта знака находится на интервале [1; 10)

```
−158,08 = −0,15808 * 10^3
0,00129 = −0,129 * 10^−2
0,15 = 0,15 * 10^0
```

float

Одинарная точность

```
32 бита
Порядок (exponent) 8 бит
Мантисса (significand) 23 бита
Можно представить целое до 2^24 без потери точности
```

double

Двойная точность

```
64 бита
Порядок (exponent) 11 бит
Мантисса (significand) 52 бита
Можно представить целое до 2^53 без потери точности
```

Специальные значения

Inf (infinity)

Возникает при переполнении или делении не нуля на ноль.

NaN (not a number)

К операциям, приводящим к появлению NaN в качестве ответа, относятся:

- все математические операции, содержащие NaN в качестве одного из операндов;
- деление нуля на ноль;
- деление бесконечности на бесконечность;
- умножение нуля на бесконечность;
- сложение бесконечности с бесконечностью противоположного знака;

- вычисление квадратного корня отрицательного числа;
- логарифмирование отрицательного числа

> NaN не равен ни одному другому значению (даже самому себе); соответственно, самый простой метод проверки результата на NaN — это сравнение полученной величины с самой собой.

Ноль

Так как в нормализованной форме ноль невозможно представить единственным способом.

Сравнение чисел с плавающей точкой на равенство

```
double a = 10 / 3.;
double b = a;
const bool isEqual = a == b; // Ok
```

```
double a = ...;
double b = ...;
// const bool isEqual = a == b; ???
const bool isEqual = fabs(a - b) < 0.0001;
```

> Погрешность представления числа увеличивается с ростом самого этого числа

https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/ (https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/)

Арифметика с массивами чисел с плавющей точкой

> При сложении большого и малого чисел малое просто исчезнет

1. Сортируем массив по возрастанию
2. Получаем сумму от 0 до максимального положительного
3. Получаем сумму от 0 до минимального отрицательного
4. Складываем результаты 2 и 3

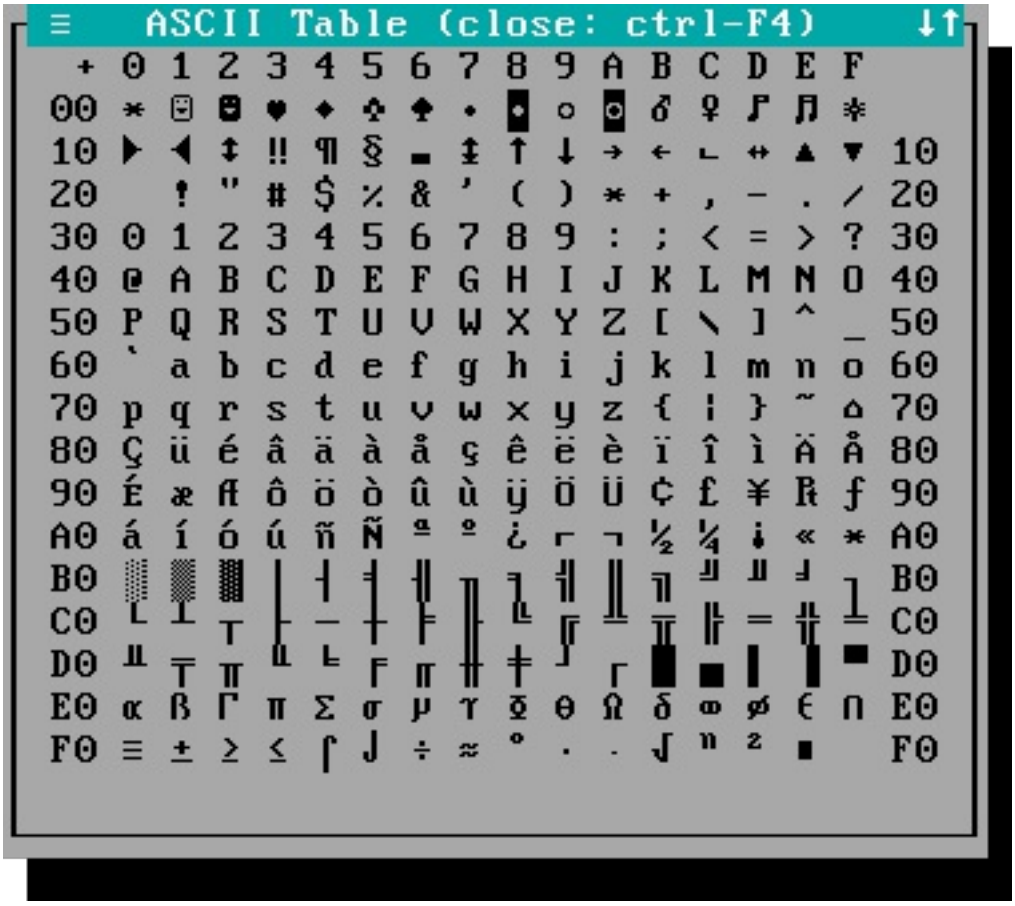
Unicode

ASCII

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

ASCII 8



KOI8-R

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	─	│	┌	┐	└	┘	├	┤	┬	┴	┼	▀	▄	▌	▐	▬
9	▒	▒	▓	⌠	■	·	√	≈	≤	≥		⌡	°	²	·	÷
A	═	‖	╒	ё	╓	╔	╕	╖	╗	╘	╙	╚	╛	╜	╝	╞
B	╟	╠	╡	Ё	╢	╣	╤	╥	╦	╧	╨	╩	╪	╫	╬	n
C	ю	а	б	ц	д	е	ф	г	х	и	й	к	л	м	н	о
D	п	я	р	с	т	у	ж	в	ь	ы	з	ш	э	щ	ч	ъ
E	Ю	А	Б	Ц	Д	Е	Ф	Г	Х	И	Й	К	Л	М	Н	О
F	П	Я	Р	С	Т	У	Ж	В	Ь	Ы	З	Ш	Э	Щ	Ч	Ъ

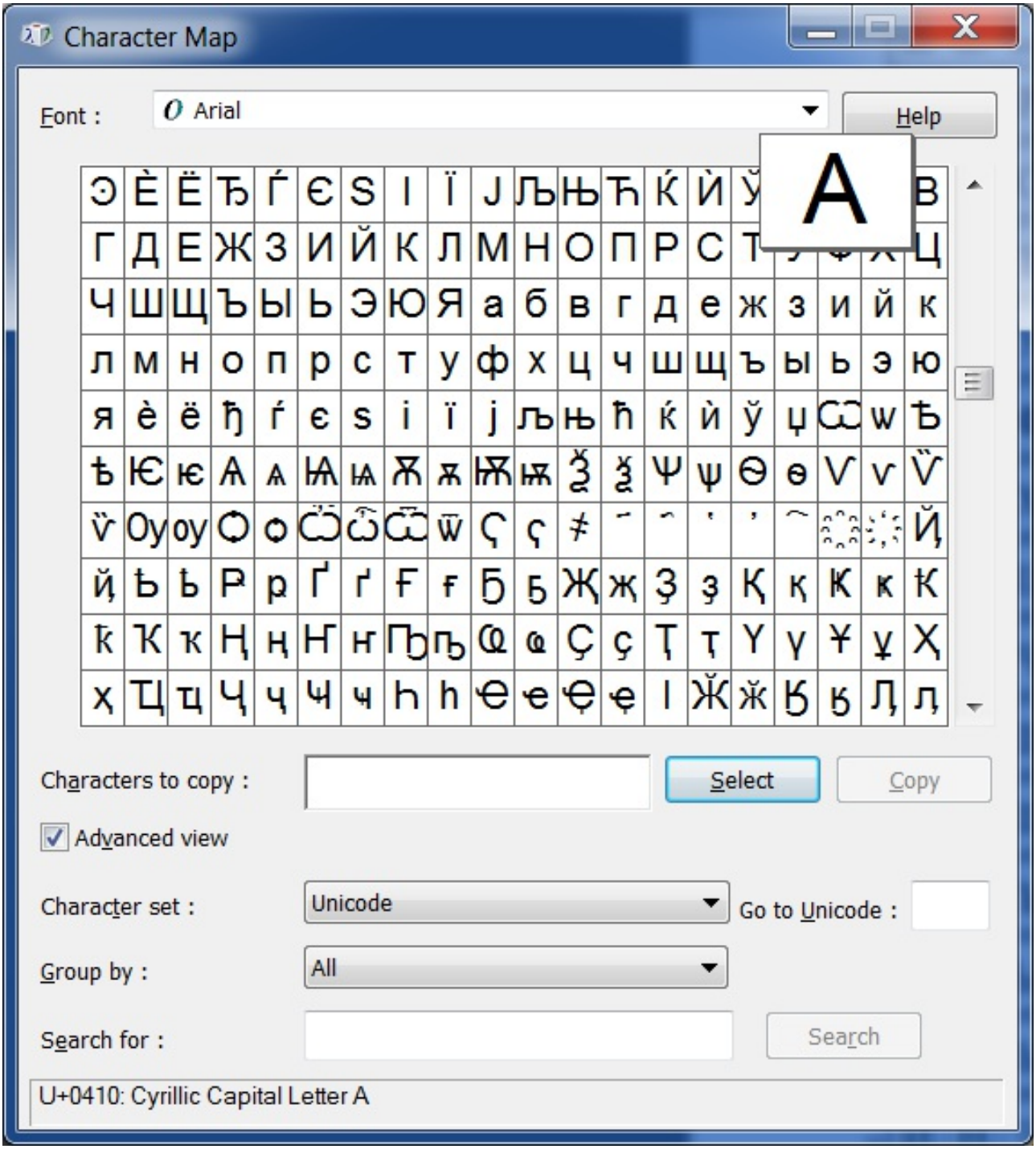
Множество кодировок

- Unicode – стандарт, а не кодировка
- CodePoint – символ в таблице Unicode (U+xxxx, где xxxx шестнадцетеричное

число, например, U+0410)

- На данный момент уже существует 9 версия стандарта
- Начиная с версии 2.0 кодовая область расширена за пределы 2^16
- В версии 6.0 описано 109 242 графических и 273 прочих символа

http://www.unicode.org/ (http://www.unicode.org/)



UCS-2

```
H       E       L       L       O
U+0048 U+0065 U+006C U+006C U+006F
```

```
00 48 00 65 00 6C 00 6C 00 6F (Big Endian)
    или
48 00 65 00 6C 00 6C 00 6F 00 (Little Endian)
```

BOM (byte order mark)

UTF-8

```
EF BB BF
```

UTF-16

```
FE FF (Big Endian)
FF FE (Little Endian)
```

UTF-32

```
00 00 FE FF (Big Endian)
FF FE 00 00 (Little Endian)
```

UTF-16

- Позволяет отобразить 2^20 + 2^16 − 2048 (1 112 064) символов
- Кодирует 0000..D7FF и E000..10FFFF (в виде суррогатных пар)
- Исключенный диапазон D800..DFFF используется как раз для кодирования суррогатных пар
- Суррогатная пара – символ кодируемый двумя 16 битными словами

UTF-8

Переменное количество байт от 1 до 6
Позволяет закодировать 2^31 CodePoints

Диапазон символов	Количество байт
00000000-0000007F	1
00000080-000007FF	2
00000800-0000FFFF	3
00010000-001FFFFF	4
00200000-03FFFFFF	5
04000000-7FFFFFFF	6

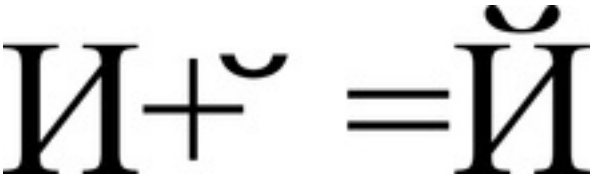
UTF-32, UCS-4

- Кодирует любой символ 4 байтами
- Позволяет индексировать 231 CodePoints
- Символы Юникод непосредственно индексируемы

Но и это еще не все

Модифицирующие символы

Графические символы в Юникоде подразделяются на протяжённые и непротяжённые (бесширинные). Непротяжённые символы при отображении не занимают места в строке. К ним относятся, в частности, знаки ударения и прочие диакритические знаки. Как протяжённые, так и непротяжённые символы имеют собственные коды. Протяжённые символы иначе называются базовыми (base characters), а непротяжённые — модифицирующими (combining characters); причём последние не могут встречаться самостоятельно. Например, символ «á» может быть представлен как последовательность базового символа «a» (U+0061) и модифицирующего символа «◌́» (U+0301) или как монолитный символ «á» (U+00E1).



Алгоритмы нормализации

Поскольку одни и те же символы можно представить различными кодами, сравнение строк байт за байтом становится невозможным. Алгоритмы нормализации (normalization forms) решают эту проблему, выполняя приведение текста к определённому стандартному виду. Приведение осуществляется путём замены символов на эквивалентные с использованием таблиц и правил. «Декомпозицией» называется замена (разложение) одного символа на несколько составляющих символов, а «композицией», наоборот, — замена (соединение) нескольких составляющих символов на один символ.

В стандарте Юникода определены 4 алгоритма нормализации текста: NFD, NFC, NFKD и NFKC.

Что еще?

Например, двунаправленное письмо.

Стандарт Юникод поддерживает письменности языков как с направлением написания слева направо (left-to-right, LTR), так и с написанием справа налево (right-to-left, RTL) – например, арабское и еврейское письмо. В обоих случаях символы хранятся в «естественном» порядке; их отображение с учётом нужного направления письма обеспечивается приложением.

Кроме того, Юникод поддерживает комбинированные тексты, сочетающие фрагменты с разным направлением письма.

И многое другое...

ICU

http://site.icu-project.org/ (http://site.icu-project.org/)

Инструменты

Valgrind

Заменяет стандартное выделение памяти своей реализацией отслеживающей корректное ее использование и освобождение.

Позволяет определить:

- Попытки использования неинициализированной памяти
- Чтение/запись в память после её освобождения
- Чтение/запись за границами выделенного блока
- Утечки памяти

Статический анализ. CppCheck

Используется для статического анализа кода. Возможности:

1. Проверяет выход за пределы
2. Обнаруживает утечки памяти
3. Обнаруживает возможное разыменовывание NULL-указателей
4. Обнаруживает неинициализированные переменные
5. Обнаруживает неправильное использование STL
6. Проверяет обработку исключительных ситуаций на безопасность
7. Находит устаревшие и неиспользуемые функции
8. Предупреждает о неиспользуемом или бесполезном коде
9. Находит подозрительные участки кода, которые могут содержать в себе ошибки

```
[check.cpp:11]: (error) Array 'c[10]' index 10 out of bounds
[check.cpp:5]: (error) Memory leak: __p
[check.cpp:17]: (error) Memory leak: a
[check.cpp:14]: (error) Mismatching allocation and deallocation: A::__p
[check.cpp:8]: (error) Null pointer dereference
```

Doxygen

Документацию можно писать в коде. Doxygen проанализирует исходный код, построит документацию (html, latex и т.д.) из определений типов и дополнит ее найденными комментариями записанными в специальном формате:

```
/*!
Находит сумму двух чисел
@param a,b Складываемые числа
@return Сумму двух чисел, переданных в качестве аргументов
*/
double sum(const double a, const double b);
```

Функции

```
double sum ( const double  a,
             const double  b
           )
```

Находит сумму двух чисел

Аргументы

a,b Складываемые числа

Возвращает

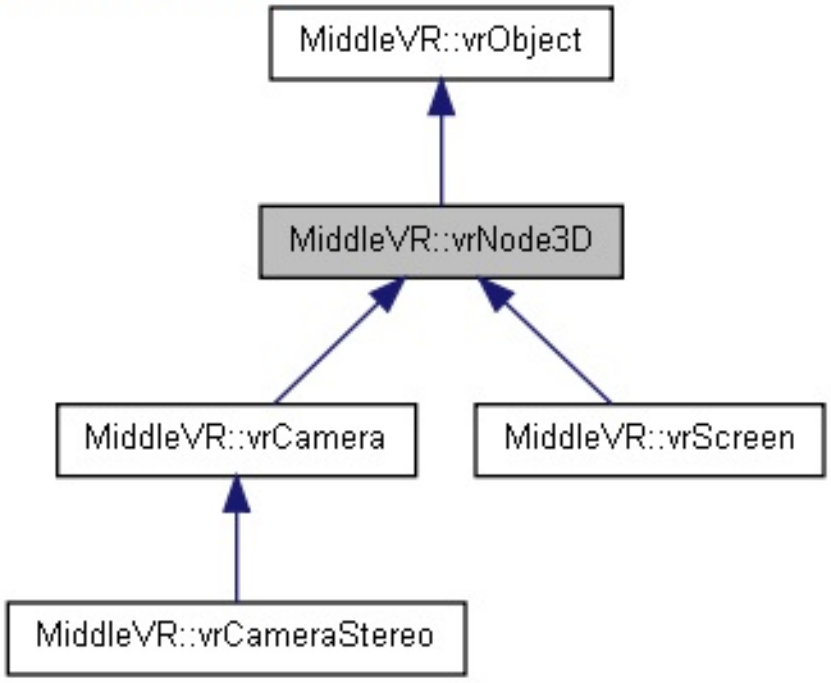
Сумму двух чисел, переданных в качестве аргументов

Если установлен Graphviz можно дополнить документацию графом вызовов и диаграммами наследования.

MiddleVR::vrNode3D Class Reference

```
#include <vrNode3D.h>
```

Inheritance diagram for MiddleVR::vrNode3D:



Ускорение сборки

В разных единицах трансляции могут использоваться одни и те же заголовочные файлы – это приводит к многократной компиляции одного и того же кода.

Напрашивается идея собрать общие заголовочные файлы в один и скомпилировать их один раз – это предварительно откомпилированные заголовочные файлы (precompiled headers):

stdafx.h

```
#include <vector>
```

a.cpp

```
#include”stdafx.h” // Всегда первой строкой
#include “a.h”

...
```

При изменении stdafx.h или включенного в него заголовочного файла перекомпилируется весь проект!

Домашнее задание

Выполнить сортировку бинарного файла содержащего числа uint64_t в 2 потока. Доступно 8 Мб памяти, больше выделять нельзя ни явно, ни неявно (например, в виде контейнеров), при этом файл может быть размером превышающим доступную память. Пространство на диске можно считать бесконечным. Сортировку выполнять в новый файл.

Тестов нет.

EOF

Друзья, домашние задания я объявляю на лекции, поэтому имею полное право до лекции их изменять. Простите