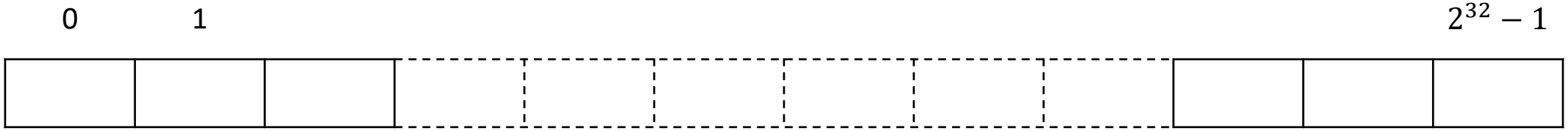
Алгоритмизация и программирование

Лекция 6 (C++)

Модель памяти

С точки зрения работающей программы память представляется как последовательность байт.

Все байты пронумерованы от 0 и до некоторого максимального значения зависящего от разрядности шины адреса процессора. Для 32-битной шины последний байт будет иметь номер $2^{32} - 1$ (т.е. максимум 4 ГБ), для 64-битной шины последний байт будет иметь номер $2^{64} - 1$ (т.е. максимум 16 ЭБ).



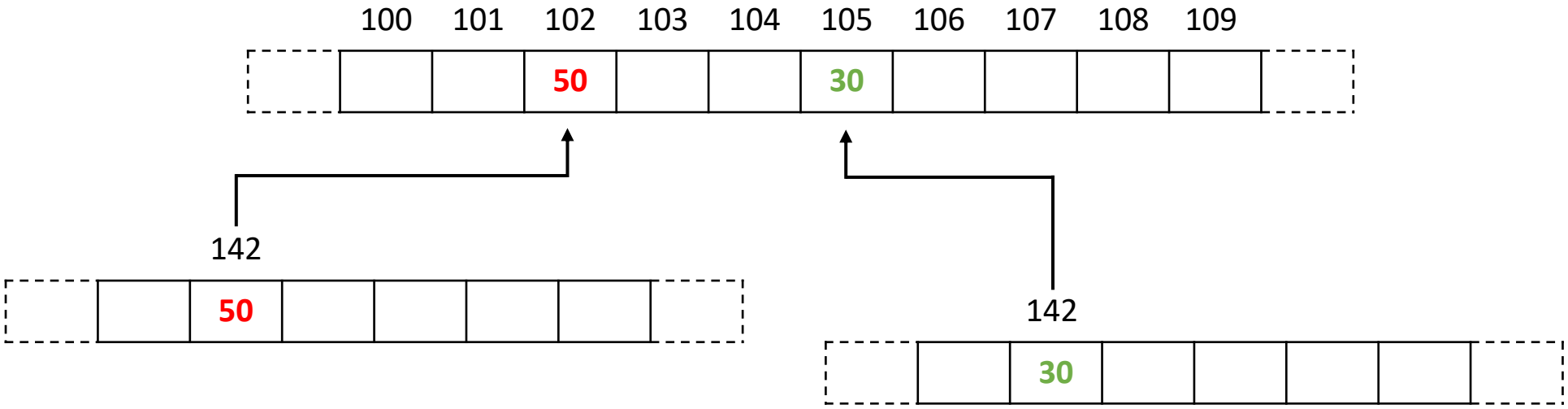
Минимальная адресуемая ячейка памяти – байт.

Если программа запускается под управлением операционной системы, то любые обращения к памяти контролируются ОС. В этом случае, максимально допустимый адрес определяется не только процессором, но и операционной системой. Т.е. если сама ОС 32-битная, то и программы под её управлением не смогут получить больше 4 ГБ памяти, не зависимо от того, какой у вас процессор и сколько у вас физической памяти.

Модель памяти

Перед запуском, исполняемый файл загружается (если большой, то просто маппится) в оперативную память.

Чтобы программы не мешали друг другу работать операционная система изолирует их друг от друга при помощи механизма виртуальной памяти. Т.е. если два процесс обратятся к одной и той же (с их точки зрения) ячейке памяти, то в реальности они обратятся к разным ячейкам физической памяти (кроме специальных случаев).



Модель памяти

Каждая программа (игра, калькулятор, hello_world приложение и т.д.) считает, что ей доступны все ячейки памяти (вплоть до 16 ЭБ), но в реальности физической памяти может быть меньше, поэтому программа не сама определяет куда размещать данные, а "спрашивать" у ОС. При этом ОС резервирует память для программы и сообщает ей место, где эта память расположена (естественно сначала физ. адрес транслируется в виртуальный).

Физическая память разделяется ОС на сегменты (страницы по 4 КБ для Intel) и по мере необходимости программе передаётся некоторое количество таких страниц. При этом ОС фиксирует, какой программе какие страницы были отданы и после завершения программы эти страницы отмечаются как свободные.

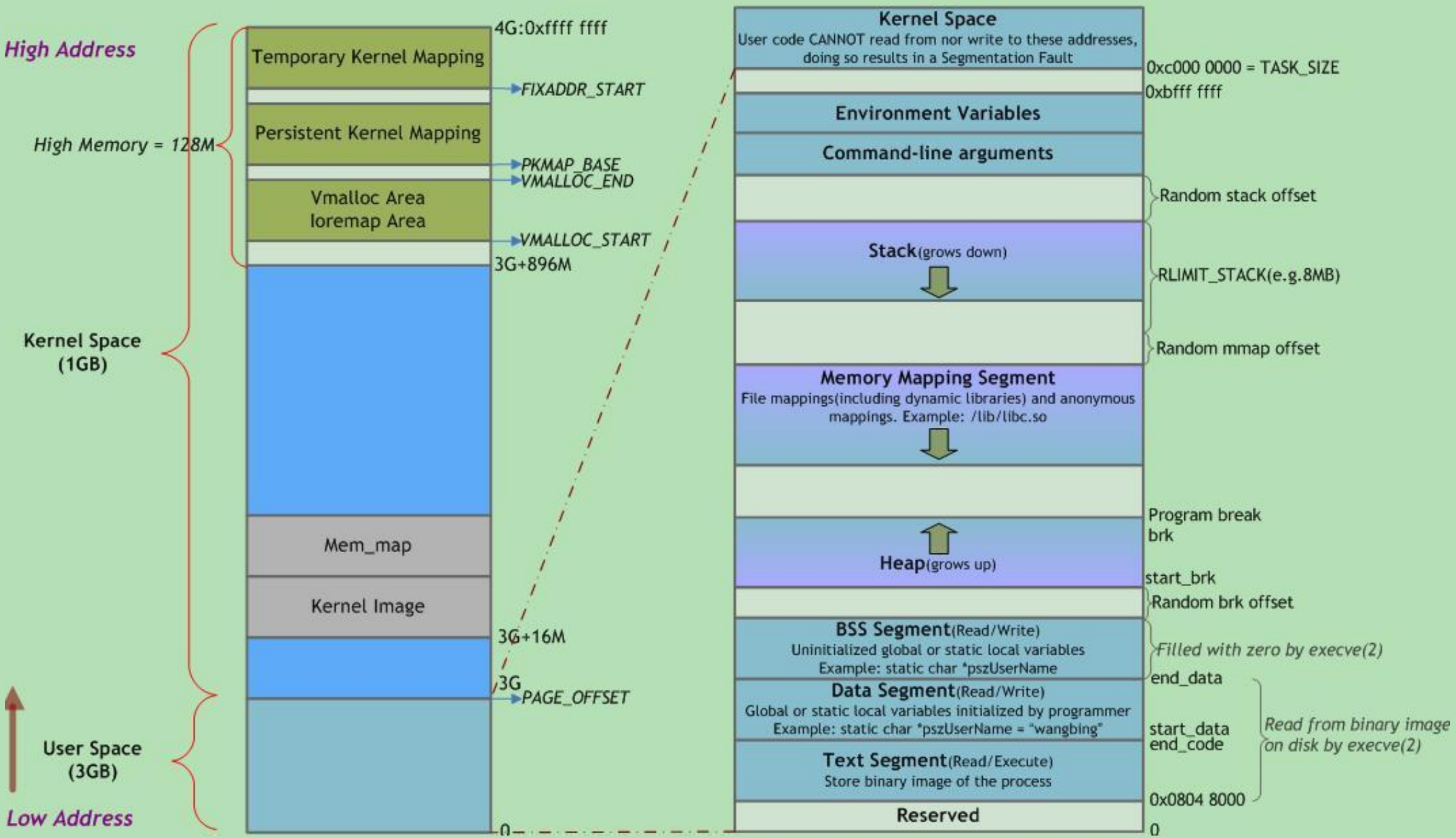
Если в программе обратиться по случайному адресу, то ОС попробует получить соответствующий адрес в физической памяти и если этот адрес находится в пределах страницы которая не принадлежит этой программе, то она будет завершена с ошибкой сегментации (segmentation fault). Такое может произойти, например если вылезти за границы массива или обратиться к переменной, которая была уничтожена (через висячий указатель).

Модель памяти

Каждой программе при старте выделяется область памяти которая называется стек (размер определяется ОС ~ 8 МБ, но можно настроить).

Каждый раз, когда в программе создаётся локальная переменная, вызывается функция, в функцию передаются параметры и из функции возвращается значение – используется память стека. Таким образом вся стековая память может быть израсходована, например при большом количестве рекурсивных (не обязательно) вызовов или, если создать достаточно большой статический массив (`int a[8*1024*1024]`). Переполнение стека, как правило, сопровождается ошибкой stack overflow. Память выделенная под локальные переменные или параметры **автоматически** освобождается при выходе из функции и в дальнейшем используется повторно.

Вся оставшаяся свободная память называется кучей (Heap). Если требуется разместить в памяти что-то большое, то можно попросить у ОС выделить фрагмент памяти требуемого размера из кучи и, если такой есть (непрерывный), он будет динамически выделен. Как только программа больше не нуждается в этой памяти её нужно вернуть операционной системе **вручную** иначе она так и будет числиться как занятая программой, пока программа не завершится. Такой случай называется утечкой памяти.


High Address
Low Address
High Memory = 128M
Kernel Space (1GB)
User Space (3GB)
4G:0xffff ffff
Temporary Kernel Mapping
FIXADDR_START
Persistent Kernel Mapping
PKMAP_BASE
VMALLOC_END
Vmalloc Area
Ioremap Area
VMALLOC_START
3G+896M
Mem_map
Kernel Image
3G+16M
3G
PAGE_OFFSET
0
Kernel Space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc000 0000 = TASK_SIZE
0xbfff ffff
Environment Variables
Command-line arguments
Random stack offset
Stack(grows down)
RLIMIT_STACK(e.g.8MB)
Random mmap offset
Memory Mapping Segment
File mappings(including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
Program break
brk
Heap(grows up)
start_brk
Random brk offset
BSS Segment(Read/Write)
Uninitialized global or static local variables
Example: static char *pszUserName
Filled with zero by execve(2)
end_data
Data Segment(Read/Write)
Global or static local variables initialized by programmer
Example: static char *pszUserName = "wangbing"
start_data
end_code
Text Segment(Read/Execute)
Store binary image of the process
Read from binary image on disk by execve(2)
0x0804 8000
Reserved
0

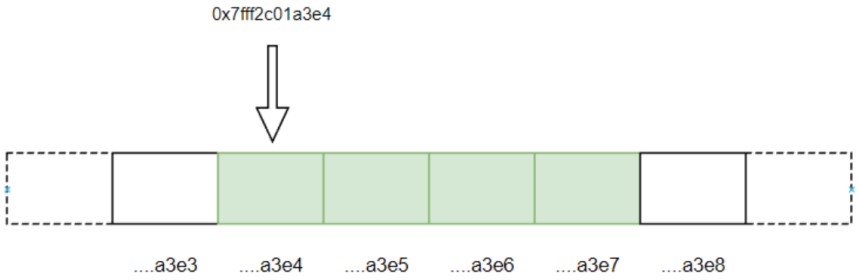
Указатель

Что такое указатель

В большинстве случаев мы взаимодействуем с памятью при помощи **имён** (переменных/констант/функций), но иногда приходится работать с ячейками памяти и их содержимым напрямую.
Для этой цели существует специальный тип данных, который называется **указателем (pointer)**. Переменная-указатель может хранить **адрес** любого байта доступного программе, а зная адрес мы можем прочитать или изменить данные которые там находятся.
Адрес можно воспринимать, как номер байта, и его можно вывести на экран как обычное десятичное число, но традиционно адреса записывают в шестнадцатеричной форме, для краткости.

```cpp
#include <iostream>

int main(){
    int i;
    int* p = &i; // Сохраняем адрес в указатель p
    std::cout << p;
}
```

```
0x7fff2c01a3e4
```



Объявление

Переменная-указатель объявляется как: Тип_данных * Имя_указателя:

```
void *i;
int *j;
string *k;
vector<string> *m;
```

Количество пробелов слева и справа от звёздочки значения не имеет:

```
int*  i;
int  *  i;
int  *i;
```

При объявлении нескольких переменных-указателей одновременно, звёздочка должна находится у каждой, иначе переменная воспринимается как переменная-значение

```
int *a, b, c;  // a - указатель, b и c – просто int-ы
int *i, j, *k; // i и k - указатели
```

Инициализация

Как и для переменных других типов, не инициализированная локальная переменная-указатель получит мусорное значение.
Если указатель хранит мусорный адрес или адрес уничтоженной переменной такой указатель называют **висячим**.

```
int* a;    // Мусорный адрес
double* b; // Мусорный адрес
```

Для того, чтобы явно показать, что указатель пока не хранит какой-то полезный адрес его инициализируют специальным значением. Такой указатель часто называют нулевым (null pointer):

```
int* a = nullptr; // Спец. значение, говорит о том, что указателю не присвоен конкретный адрес
int* b = NULL;    // Аналогично, но в стиле C
int* c = 0;       // Так тоже можно. По смыслу, то же, что и предыдущие два
int* e;           // Если переменная глобальная или static, то гарантируется значение 0
```

В C++ предпочтительно использовать `nullptr`. Это единственное значение специального типа `std::nullptr_t` специально добавленного в язык для этой цели.

```
int* i = new int[10];  // Пытаемся выделить блок памяти
if (i == nullptr) cout << "Ошибка, память не выделена";
```

Способы получения адреса

Указатели хранят адреса. В норме, адрес не записывают в коде вручную, а получают у существующего объекта или как результат функции.

```
int two = 2; // У two адрес есть
2;           // У 2 адреса нет
two + 2;     // У результата адреса нет. Временный объект

void foo(int x){}  // У foo адрес есть
```

Адрес можно получить у любого объекта, который имеет своё постоянное место в памяти. У временных объектов получить адрес нельзя.

Для получения адреса существует оператор взятия адреса - **&** (символ называется амперсанд). Чтобы получить адрес он располагается слева (перед) от переменной адрес которой хотим получить:

```
int one;
&one;    // Адрес переменной one

int arr[10];
&arr[2]; // Адрес элемента массива arr с индексом 2

int* i;
&i;      // Адрес переменной i
```

Для того, чтобы адрес можно было сохранить в переменную, тип указателя должен совпадать с типом адресуемого объекта.

Виды указателей

Не типизированный указатель

Хранит адрес чего угодно, не обладает никакой дополнительной информацией о том, на что указывает:

```
void* a;           // Не инициализированный. Указывает по мусорному адресу
void* b = nullptr; // Спец. значение, говорит о том, что указателю не присвоен конкретный адрес
void* c = NULL;    // Аналогично, но в стиле C
void* d = 0;       // Так тоже можно. По смыслу, то же, что и предыдущие два

int one;           // Просто переменная целого типа (не указатель)
double two;        // Просто переменная вещественного типа (не указатель)
void* x = &one;    // Берём адрес one и сохраняем его в x
void* y = &two;    // Берём адрес two и сохраняем его в y
y = x;             // Теперь и x и y хранят адрес one
```

Т.к. не типизированный указатель не обладает информацией от типе данных на которые указывает, то его невозможно использовать для доступа к данным в памяти.

Типичный сценарий использования – передача адреса в функцию или возврат адреса из функции в тех случаях, когда сами данные можно воспринимать как простую последовательность байт.

Не типизированный указатель

В C++ не типизированные указатели применяются для совместимости с функциями C.

Например функции: `malloc`, `free`, `memcpy` объявлены как:

```
void *malloc( size_t size );
void free( void *ptr );
void* memcpy( void* dest, const void* src, std::size_t count );
```

И могут быть использованы, например так:

```
void* mem = malloc(4 * sizeof(int)); // Получили 16 байт "сырой" памяти
int* array = (int*) mem;             // Преобразуем "сырую" память в типизированную
array[0] = 1;                        // Используем
free(array);                         // Освобождаем
```

Типизированный указатель

Хранит адрес только объектов определённого типа. Знает тип данных на которые указывает:

```
int* a;           // Не инициализированный. Указывает по мусорному адресу
int* b = nullptr; // Спец. значение, говорит о том, что указателю не присвоен конкретный адрес
int* c = NULL;    // Аналогично, но в стиле C
int* d = 0;       // Так тоже можно. По смыслу, то же, что и предыдущие два

int one;
int* i = &one;    // Получаем адрес и сохраняем в переменную i

double two;
int* j = &two;    // Ошибка. Не правильный тип данных double*, j ожидает int*

int* k = &1137;   // Ошибка. У временных объектов нет постоянного адреса в памяти
```

В качестве типа указателя может быть использован любой тип данных:

```
bool* a;
string* b;
vector<string>* c;
vector<vector<int>>* d;
map<string, string>* e;
```

```
int* f[10];
int** g;
int*** j;
const int**** k;
double (*n)(double, double);
```

Типизированный указатель

Типизированный указатель позволяет получить доступ к памяти
на которую указывает при помощи унарного оператора
разыменования (dereference) * (символ называется звездочка или астериск).

Получить доступ – значит, что дальнейшие операции будут выполняться так,
как если бы вместо указателя была переменная того типа с которым работает
указатель.

```cpp
#include <iostream>

int main(){
    int one = 1;
    int* i = &one;
    cout << *i;
}
```

```
1
```

```cpp
int one = 1;
int* i = &one;
*i += 1;    // i - по прежнему указывает на one. Значение one теперь 2

int five = 5;
i = &five;  // i - теперь указывает на five. Значение one по прежнему 2

string two = "2";
string *j = &two;
*j += 1;    // Ошибка, т.к. тип string не поддерживает операцию сложения с числами
```

Указатели и константность

Указатели и константность

Сам указатель – это переменная и мы можем запретить изменять его значение при помощи const, тем самым превратив его в константный указатель.

```
int one = 1;
int two = 2;
int* const p = &one;
*p += 5;   // Допустимо, т.к. изменяется one
p = &two;  // Ошибка! Попытка изменить p
```

Данные, адрес которых хранит указатель – тоже могут быть константой, в этом случае у нас будет указатель на константу. Такой указатель может только читать данные, но не изменять.

```
int one = 1;
int two = 2;
const int* p = &one;
*p += 5;   // Ошибка! Попытка изменить константу
p = &two;  // Допустимо, т.к. изменяется p
```

В тоже время мы можем захотеть создать константный указатель на константные данные.

```
int one = 1;
int two = 2;
const int* const p = &one;
*p += 5;   // Ошибка! Попытка изменить константу
p = &two;  // Ошибка! Попытка изменить p
```

Если указатель **гарантирует** константность данных, то он может сохранить адрес как константы, так и не константы.
В обратном случае, только адрес не константы.

```
int one = 1;
const int two = 2;
int* p = &one;
p = &two;  // Ошибка! p - не гарантирует
           // константности данных
```

Указатели и константность

Возможны и более сложные комбинации слова const и * при описании типа указателя.
Не нужно пытаться прочитать тип слева на право. Для начала найдите самую ближайшую к имени переменной звёздочку и всё, что справа от неё будет относится к самому указателю, а всё, что слева к типу данных на которые он ссылается:

```
const int one = 1;
const int* p1 = &one;  // p1 - переменная указывающая на: const int
const int** p2 = &p1;  // p2 - переменная указывающая на: const int*
const int** const * p3 = &p2; // p3 - переменная указывающая на: const int** const
const int** const ** const p4 = &p3; // p4 - константа указывающая на: const int** const *
```

Размер указателя

Размер указателя

Определим размер, который необходимо выделить в памяти для размещения переменных-указателей разного типа:

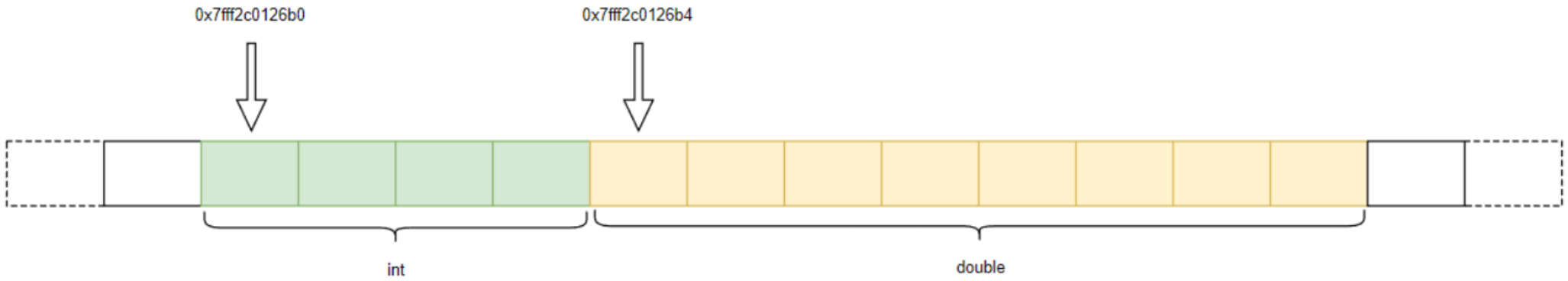
```
cout << sizeof(int *) << '\n'          // 8
     << sizeof(double *) << '\n'       // 8
     << sizeof(vector<int> *) << '\n'; // 8
```

Как видно, для всех трёх случаев мы получили одинаковый результат. На самом деле, в зависимости от разрядности системы результат будет отличаться: на x32 - 4 байта, на x64 - 8 байт и т.д.

Размер указателя

Переменная `int` и `vector<int>` разного размера, каким образом они "помещается" в указателе, если все указатели одинакового размера?

На самом деле, указатель хранит информацию только об адресе первого байта, но не об адресе каждого байта переменной на которую указывает.
Первый байт - это точка где переменная начинается (адрес байта с минимальным номером). Т.к. указатель знает тип на который указывает, то он знает и размер этого типа. Зная точку начала и размер, можно определить где переменная заканчивается и все промежуточные байты.



Преобразование типа указателя

Преобразование типа указателя

Преобразование указателя типа A к указателю типа B - это НЕ безопасная операции, поэтому **неявно** она не выполняется.

```
int* i;
double* j;
i = j; // Ошибка, т.к. указатели разных типов
j = i; // Ошибка, т.к. указатели разных типов
```

Программист должен явно прописать преобразование в коде, чтобы оно выполнилось.
Можно использовать преобразование в стиле C или в стиле C++.

```
int* i;
double* j;
i = (int*) j; // OK
i = reinterpret_cast<int*>(j); // OK
i = static_cast<int*>(j);      // Ошибка
```

В процессе преобразования типа указателя, значение которое он хранит не изменяется, т.е. он продолжает хранить адрес той же ячейки в памяти что и раньше.

```
int one = 1;
int* i = &one;

cout << i << endl;   // 0x7ffe5903203c
cout << (double*) i; // 0x7ffe5903203c
```

Разница будет заметна, после применения операции разыменования, т.к. байты в памяти будут трактоваться согласно новому типу.

```
int one = 1;
int* i = &one;

cout << *i << endl;   // 1
cout << *(double*) i; // 9.61056e+128
```

Преобразование типа указателя

Для не типизированных указателей сделано исключение,
т.к. их основная задача – быть промежуточным хранилищем адресов любого типа, то любой адрес неявно может быть преобразован в `void*`.

```
int* i;
double* j;
void* k;

k = i;  // OK
k = j;  // OK
```

В обратном направлении **неявное** преобразование не работает,
но, как и для остальных типов, работает явное.

```
int* i;
double* j;
void* k;

i = (int*) k;  // OK
j = reinterpret_cast<double*>(k);  // OK
j = static_cast<double*>(k);       // OK
```

Всё вышесказанное справедливо для любых типов, кроме классов связанных через иерархию наследования. Для них указатели (и ссылки) преобразуются по другому.

Логические операции с указателями

Логические операции с указателями

Логические операции работают с указателями так, как если бы **адреса**, которые хранят указатели, были обычными числами, поэтому все логические операции с указателями работают.
Логические операции между указателями разных типов запрещены.

```
int one = 2;   // Локальная переменная
int two = 2;   // Локальная переменная

&one > &two;   // false
&one < &two;   // true
&one == &two;  // false
&one != &two;  // true
```

Преобразование адреса в булев тип происходит по правилу: нулевой указатель – `false`; не нулевой – `true`;

```
int* a = new int[10];  // Пытаемся выделить блок памяти. Получим nullptr или адрес
int* b = new int[10];  // Аналогично

if (!a || !b) cout << "Ошибка! Память не выделена";
else cout << "Массивы созданы";
```

Арифметика указателей

Арифметика указателей

В отличие от логических операторов, арифметические НЕ работают с адресами как с числами. Кроме того, арифметические операторы `*, /, %` не применимы к указателям в принципе. Разрешены только операторы инкремент/декремент, увеличивающие/уменьшающие значение указателя на целое число и вычисляющий разность между двумя указателями одного типа. Результат при этом определяется как самими адресами, так и размером типа указателя.

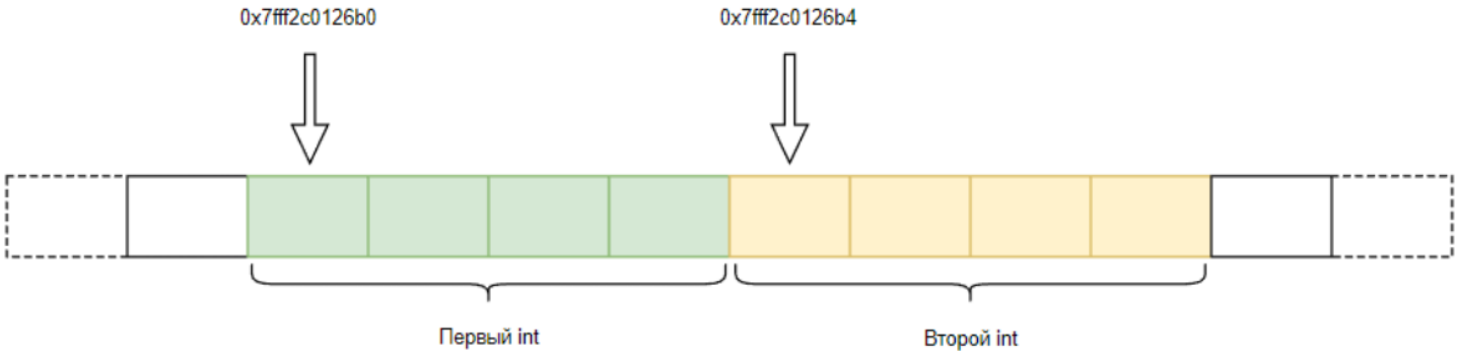
Для не типизированного указателя, по стандарту, нет разрешённых арифметических операций (т.к. у `void` нет размера), хотя некоторые компиляторы позволяют применять к ним операторы инкремент/декремент и увеличивающие/уменьшающие значение указателя на целое число (при этом размер `void` принимается за 1 байт).

Операторы ++ и --

Увеличивают или уменьшают **адрес** хранимый в указателе на:

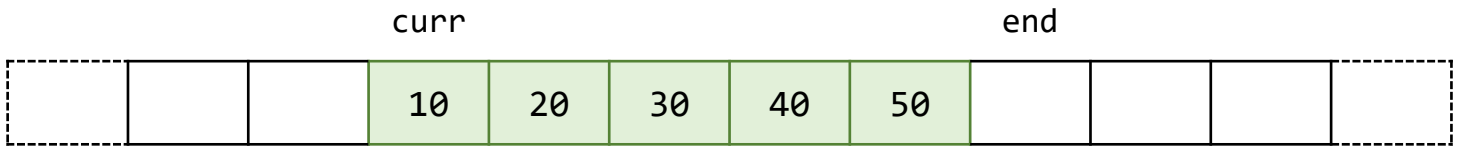
- если посмотреть изменение адреса в байтах, то разница будет равна размеру типа указателя в байтах;
- если представить всю память как массив элементов типа указателя, то разница будет 1 элемент;

```
int one = 1;
int* i = &one;
cout << sizeof(int) << endl; // 4
cout << i << endl; // ....26b0
++i;
cout << i;         // ....26b4
```



Типичное применение операторов ++ и -- – это итерирование по элементам массива:

```
int array[5] = {10, 20, 30, 40, 50};
int* curr = &array[0];
int* end = &array[5];

while (curr != end){
    std::cout << *curr++ << ' ';
}
```



Операторы +, -, +=, -=

- Случай сложения/вычитания адреса и целого числа.

Сложение адреса и целого числа N смещает адрес вперёд на:

- `N * sizeof(тип)` байт;
- N элементов типа указателя.

Вычитание, соответственно, смещает адрес назад.

```
double one = 1;
double* i = &one;
cout << sizeof(double) << endl; // 8
cout << i << endl; // ....7f98
i += 10;
cout << i;         // ....7fe8
```

- Сложение адресов НЕ работает.

- Случай вычитания адреса из адреса.

Существует специальный тип, предназначенные для сохранения результата разности указателей: std::ptrdiff_t.

Вычитание адреса из адреса возвращает НЕ количество байт, а количество элементов типа указателя которое помещается между двумя адресами, т.е. разницу в байтах делённую на размер типа в байтах.

```
double one = 1;
double* i = &one;
double* j = i+1;
cout << sizeof(double) << ' ' << j - i << endl; // 8 1
cout << sizeof(int)    << ' ' << (int*)j - (int*)i << endl; // 4 2
cout << sizeof(char)   << ' ' << (char*)j - (char*)i << endl; // 1 8
```

Оператор [ ]

Т.к. указатели часто используются для работы с массивами, к ним применим оператор квадратные скобки [ ].

С точки зрения C++ выражение: `указатель + число`

эквивалентно выражению: `указатель[число]`

```
int array[3] = {10, 20, 30};
int* i = &array[0];

cout << *i   << ' ' <<  *(i + 1) << ' ' << *(i + 2) << endl; // 10 20 30
cout << i[0] << ' ' <<  i[1]     << ' ' << i[2] << endl;     // 10 20 30
```

При этом само число вполне может быть отрицательным:

```
int array[3] = {10, 20, 30};
i = &array[2];

cout << i[-2] << ' ' << i[-1] << ' ' << i[0] << endl;        // 10 20 30
```

Оператор [ ]

Оператор [ ] позволяет в выражении поменять указатель и число местами.

С точки зрения C++ выражение: `число + указатель`
эквивалентно выражению: `число[указатель]`

```
int array[3] = {10, 20, 30};
int* i = &array[0];

cout << *i   << ' ' <<  *(i + 1) << ' ' << *(i + 2) << endl; // 10 20 30
cout << 0[i] << ' ' <<  1[i]     << ' ' << 2[i] << endl;     // 10 20 30
```

Такая форма записи, как правило, нигде не используется, но с точки зрения языка, конструкция валидная.

Варианты применения указателей

Динамическое выделение памяти

Для приложений сложнее "Hello, World" 8 МБ памяти стека, как правило, не достаточно и приходится использовать дополнительную память из кучи (Heap). Чтобы не занимать память, в те моменты, когда программа в ней не нуждается, память из кучи выделяется и освобождается динамически в процессе работы программы.
Локальные переменные создаются обычным образом позволяют получить доступ к выделенной памяти через имя. К переменной созданной динамически можно получить доступ только через адрес и чтобы его сохранить нужна переменная-указатель:

```
int a = 10;  // Локальная переменная a (создана на стеке)
int* p = new int{10}; // Локальная переменная p и
                      // переменная созданная в куче (без имени)
int& r = *new int{10}; // Так можно, но обычно не делают
```

Локальные переменные (и параметры функции) автоматически уничтожаются после выхода из блока в котором они были созданы и освобождают занимаемую память. Переменная созданная вручную будет существовать, пока не будет освобождена вручную. Если потерять адрес переменной, то удалить её будет не возможно. Такой случай называется утечкой памяти:

```
delete p;  // Освобождаем память
delete &r; // Освобождаем память
```

Динамическое выделение памяти

Кроме выделения памяти под одиночную переменную можно выделить блок памяти (массив).

```
int a[5] {1, 2, 3, 4, 5}; // массив a (создан на стеке)
int* p = new int[5] {1, 2, 3, 4, 5}; // Локальная переменная-указатель p и
                                     // массив созданный в куче (без имени)
int* cp = (int* )malloc(5 * sizeof(int)); // Аналогично но в стиле C
```

В случае статического массива, его размер должен быть константой времени компиляции, т.е. быть известен до того, как программа будет запущена. Для массивов, созданных динамически такого ограничения нет.

Некоторые компиляторы позволяют задавать размер статического массива в процессе выполнения программы, но это является нестандартным расширением и **ошибкой** с точки зрения C++. Такая возможность допускается компиляторами для лучшей совместимости с C, т.к. в C есть такая возможность, которая называется VLA (Variable-length array).

Динамически созданные массивы должны быть уничтожены тоже вручную при помощи соответствующей функции:

```
delete[] p;  // Освобождаем память выделенную через new[]
free(cp);    // Освобождаем память выделенную через malloc
```

Обратите внимание, что в этот раз у `delete` присутствуют пустые квадратные скобки. Кроме того, указатели нельзя смещать иначе удаление произойдет не правильно. Удалять элементы массива по одному тоже нельзя. Блоком создали, блоком удалили.

Динамическое выделение памяти

Обратите внимание, что `delete` и `free` не изменяют значение указателей, а только освобождают память. Из за этого можно допустить ошибку обращения к памяти, которая уже освобождена, что в лучшем случае приведет к падению программы. Если вы собираетесь использовать этот же указатель далее по тексту программы, имеет смысл вручную присвоить ему значение `nullptr` после освобождения памяти.

Вызов `delete` или `free` для указателей равных `nullptr` не является ошибкой.

Работа с массивами

По правилам C и C++ первое измерение статического массива может "деградировать" (распадаться) до указателя (Array Decay). Decay - автоматическое преобразование к более простому типу.

```
int a[10];
int b[20];

int* i = a; // OK. array decay
int* j = b; // OK. array decay

Здесь a и b - массивы int-ов.
i и j - адреса нулевых элементов этих массивов
```

Например для многомерных статических массивов:

```
int a[10][5];
int b[8][4][2];

int (*i)[5] = a;
int (*j)[4][2] = b;

Здесь a - массив из 10 массивов по 5 int-ов
i - адрес нулевого массива из 5 элементов

b - массив из 8-ми матриц 4x2 типа int
j - адрес нулевой матрицы 4x2 типа int
```

На второе и последующие измерения это правило не распространяется

```
int a[10][5];

int** i = a;  // Ошибка
```

Работа с массивами

Для создания динамических массивов (неудачное название), классически используется конструкция с ручным выделением и освобождением блоков памяти:

```
int* arr = new int[10];  // Создаём
delete[] arr;            // Освобождаем
```

```
int rows = 10, cols = 20;

// Создаём
int** arr2D = new int*[rows];
for (int i=0; i<rows; i++) arr2D[i] = new int[cols];

// Освобождаем
for (int i=0; i<rows; i++) delete[] arr2D[i];
delete[] arr2D;
```

```
int rows = 10, cols = 20;

// Создаём
int** arr2D = new int*[rows];
int* buff = new int[rows * cols];
for (int i=0; i<rows; i++) arr2D[i] = buff + i*cols;

// Освобождаем
delete[] arr2D;
delete[] buff;
```

При этом адрес блока памяти сохраняется в указатель и в дальнейшем он используется для доступа к элементам массива при помощи квадратных скобок.

В данном случае сам массив выделяется из памяти кучи (heap), а память под указатель выделяется на стеке. Память выделенная в куче не будет освобождена, пока мы её руками не почистим. Обратите внимание, что для массивов используется `delete[]`.

Кроме того, в данном примере указатели нельзя смещать, иначе удаление будет не корректным.

Работа с массивами

Из за перечисленного выше, у многих складывается ошибочное впечатление, что массив и указатель это одно и тоже. На самом деле это совершенно разные вещи. При создании массива выделяется достаточно памяти, чтобы поместились все его элементы. При создании указателя, выделяется только место достаточное для хранения адреса.

Передача аргументов в функцию

В C есть только 2 варианта передать данные в функцию: по значению и по указателю. В C++ добавлены ещё ссылки, которые под капотом, как правило сводятся к тем же указателям и являются скорее синтаксическим сахаром.
Передача в функцию адреса вместо значения, позволяет не создавать лишнюю копию и изменять исходное значение.

```
void set_zero(int* p){
    *p = 0;
}
int i = 10;
set_zero(&i);
std::cout << i;  // 0
```

В данном примере, параметр p получает копию адреса переменной `i` и он используется для изменения значения `i`.

Передача аргументов в функцию - массив

Передать в функцию статический массив по значению в явном виде не возможно, только указатель/ссылку на него.

```
Все эти 3 объявления - идентичны:

int sum(int* arr,  int size);
int sum(int arr[],  int size);
int sum(int arr[100],  int size); // Число не имеет значения
```

Все перечисленные объявления соответствуют этому определению:

```
int sum(int* arr,  int size){
    int accum = 0;
    for(int i=0; i<size; i++) accum += arr[i];
    return accum;
}

int a[4] = {1, 2, 3, 4};
cout << sum(a, 4); // 10
```

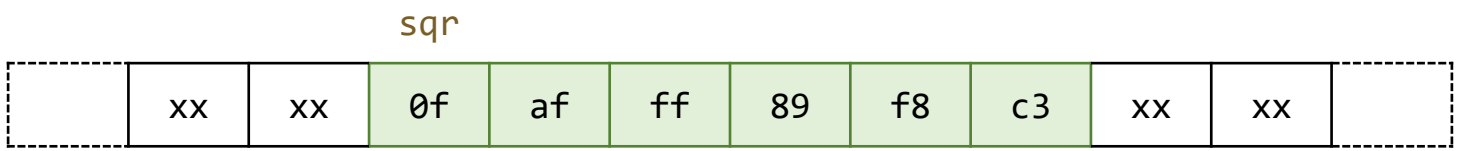
Если попробовать **дополнительно** определить функцию с любым из перечисленных выше заголовков, мы получим сообщение, о повторном определении, т.к. с точки зрения компилятора они не различимы.

Передача аргументов в функцию - функция

С точки зрения процессора, фикция – это просто последовательность байт, которая начинается по некоторому адресу. Процессор читает эти байты и последовательно исполняет. Например функция:

```
int sqr(int num) {
    return num * num;
}
```

Может быть представлена в виде:



Таким образом ничто не мешает нам сохранить адрес этого байта в переменную-указатель (указатель на функцию) и в дальнейшем использовать её вместо имени функции:

```
int sqr(int num) {
    return num * num;
}

int (*p)(int) = sqr;
p(5); // 25
```

Передача аргументов в функцию - функция

Разумеется, создавать дополнительные имена для функций не очень полезная возможность, поэтому указатели на функцию, как правило применяются для передачи функции как аргумент в другую функцию (функции высшего порядка). Часто применяется в сортировках, поиске и других стандартных алгоритмах:

```
int mult(int a, int b) {
   return a * b;
}

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

using bin_operation = int(*)(int, int); // псевдоним в стиле C++
// type int(*bin_operation)(int, int); // псевдоним в стиле C

int Do(int a, int b, bin_operation op){
    return op(a, b);
}

std::cout << Do(10, 50, mult); // 500
std::cout << Do(10, 50, sum);  // 60
```

Передача аргументов в функцию - aliasing

В случае передачи данных в функцию по указателю (или ссылке) возможен случай, когда два и более параметров-указателей в действительности ссылаются на одну и тоже переменную или просто пересекающуюся область памяти.
В C++ компилятор по умолчанию предполагает, что для указателей **разных типов** это не так (Strict aliasing). Нарушение данного правила приводит к неопределённому поведению.

```
int foo( float* f, int* i ) {
    *i = 1;
    *f = 0.0; // Изменение f не влияет на i?
    return i;
}

int x = 0;
x = foo(reinterpret_cast<float*>(&x), &x);
```

В C ситуация обратная и из-за этого компилятор не может генерировать эффективный код и добавляет лишние операции работы с памятью. Чтобы дать компилятору гарантии, что объявляемый указатель адресует область памяти, на которую не ссылается никакой другой указатель используется ключевое слово `restrict` (в C++ его нет).

Для указателей одинаковых типов, компилятор C++ не может полагаться на strict aliasing и поэтому может генерировать не очень оптимальны код. Многие компиляторы C++ поддерживают нестандартное расширение, добавляющее ключевое слово `__restrict` (указывается после *), которое позволяет гарантировать компилятору непересеченние памяти.

Возврат адреса из функции

В C++, редко используют возврат сырых указателей или ссылок из функции, т.к. это может приводить к появлению висячих указателей/ссылок:

```
int* foo(){
    int a = 10;
    return &a;
}

int* p = foo(); // Висячий указатель, т.к. переменная a удалена
```

Такое можно встретить, когда функция специально предназначена для создания объектов в куче или функции передаётся уже выделенный блок памяти и возвращаемый указатель будет находится где-то внутри этого блока:

```
int* find(int* begin, int* end, int target) {
    for (; begin != end; ++begin)
        if (*begin == target) return begin;
    return nullptr;
}

int* newArray(int size, int value=0){
    int* res = new int[size];
    for (int i=0; i<size; i++) res[i] = value;
    return res;
}
```

Полиморфизм через наследование

На указателях и ссылках основывается одна из реализаций идеи полиморфизма в C++, а точнее реализация полиморфизма через наследование.
Об этом подробнее будет в блоке про указатели и ООП.

Как читать тип указателя

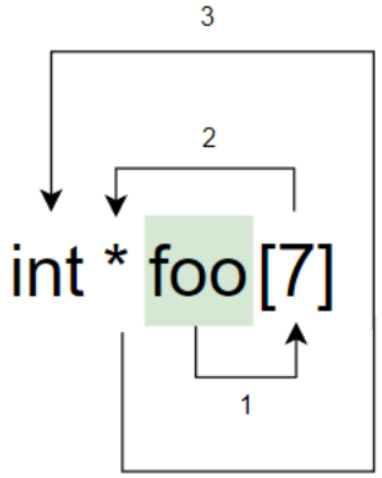
Декларация типов С

Когда появляются указатели типы становится очень сложно читать.
Чтобы научится это делать, нужно запомнить специальное правило:
Направо, налево, наружу, повторить.

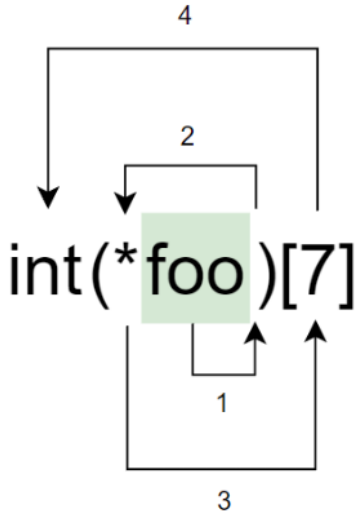
Это значит, что когда вы читаете тип переменной, то читаете его как бы по спирали начиная от имени:
0) найти имя переменной;
1) Посмотреть что стоит сразу справа от имени и прочитать один элемент:
  - Если ничего, то переходим на (2)
  - Если закрывающая скобочка, то переходим на (2)
  - Если открывающая скобочка, то читаем все до закрывающей скобки.
  - Если после квадратных скобок идут ещё квадратные скобки, продолжаем читать их тоже.
2) Посмотреть что стоит сразу слева от имени и прочитать 1 элемент.
3) Выйти на следующий круг и начать с пункта (1)

Здесь можно получить расшифровку типа указателя: https://cdecl.org/

Декларация типов C




```
foo - это (1) массив из 7 (2) указателей на (3) int

int a0, a1, a2, a3, a4, a5, a6;
int* foo[7];
foo[0] = &a0;   // Указатель на int
foo[1] = &a1;   // Указатель на int
....
foo[6] = &a6;   // Указатель на int
```




```
foo - это (1-2) указатель на (3) массив из 7 (4) int-ов

int array[7];           // Массив
int(*foo)[7] = &array;  // Указатель на массив
```

```
Здесь на шаге (1) справа от имени скобочки и через них мы пройти
не можем пока не прочитали всё что в них есть, после чего мы
выходим из скобочек наружу и читаем дальше
```

Декларация типов C

```
int(*foo[2])[3][3];
```

```
foo - это массив из 2 указателей на двумерный массив 3x3 с элементами типа int

int mat1[3][3];  // Матрица
int mat2[3][3];  // Матрица

int(*foo[2])[3][3];
foo[0] = {&mat1};
foo[1] = {&mat2};
```

```
int (*(*foo)(int, double));
```

```
foo - это указатель на функцию от двух аргументов int и
double в возвращающую указатель на int

int* func(int a, double b){  // Функция
  // Код
}

int (*(*foo)(int, double)) = &func;
```

Декларация типов C

Чтобы не выписывать сложные типы вручную, можно создать переменную-указатель при помощи ключевого слова auto. В этом случае, компилятор автоматически выведет тип переменной на основе типа инициализатора:

```
auto one = 1;   // Тип one: int
auto i = &one;  // Тип i: int*
auto j = &i;    // Тип j: int**

int* func(int a, double b){  // Функция
  // Код
}
auto foo = &func; // Тип foo: int *(*foo)(int, double)
```

Замечания про * и &

```
void* a;            // В таком выражении * - это часть типа. Тип: void*
int* one;           // Здесь тоже. Тип: int*

*one;               // Здесь * - это оператор, который совершает действие с указателем.
                    // При этом указатель - это one, но не *one.
                    // *one - это int.
*one = 1;           // ОК

int value = 0;
int& two = value;   // В таком выражении & - это часть типа. Тип: int&
two = 2;

&value;             // Здесь & - это оператор, который совершает действие с value.
                    // При этом &value - это не ссылка, это адрес
&value = 2;         // Ошибка!
```

Указатель в ООП

Доступ к полю структуры/класса

Если тип указателя – структура или класс

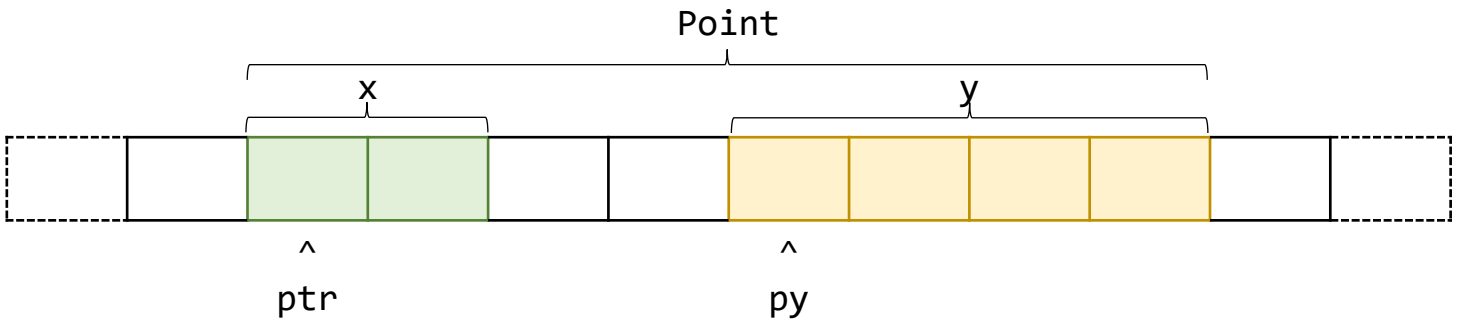
```
struct Point{
    short x;
    int y;
};

Point p{0, 0};
Point* ptr = &p;
```

Можно получить доступ к её полю или **методу**:

```
(*ptr).x = 100;
ptr->x   = 100;   // Эквивалентно предыдущему
```

Кроме того, можно напрямую получить указатель на поле (но не на метод) класса:

```
int* py = &p.y;
*py = 100;
```



Доступ к полю структуры/класса - static

Указатели на статические поля можно брать как через экземпляр класса, так и без него. Физически статические поля располагаются НЕ рядом с остальными, в той же области памяти, что и глобальные переменные:

```
struct Point{
  short x;
  int y;
  static int z;
};

int Point::z = 0;

Point p;
std::cout << &p.x << ' ' << &p.y << ' ' << &p.z << std::endl;
std::cout << &Point::z << std::endl;
```

```
0x7fff3122eaf0 0x7fff3122eaf4 0x5655f3ed8154
0x5655f3ed8154
```

Указатель на член класса - поле

В отличие от привычных указателей, которые получают **абсолютные** адреса ячеек памяти, существуют указатели на член класса (на самом деле только на не статический).

```
class Bowl {
public:
    int apples;
    int oranges;
};

int count_fruit(Bowl arr[], int size, int Bowl::* fruit){
    int count = 0;
    for (int i = 0; i < size; i++){
        count += arr->*fruit;  // arr.*fruit если бы arr не был указателем
        arr++;
    }
    return count;
}

int main(){
    Bowl bowls[2] = { { 1, 2 }, { 3, 5 } };
    std::cout << "I have " << count_fruit(bowls, 2, & Bowl::apples) << " apples\n";
    std::cout << "I have " << count_fruit(bowls, 2, & Bowl::oranges) << " oranges\n";
}
```

Указатель на член класса - метод

Чтобы получить указатель на нестатический метод класса можно написать так:

```
class Robot {
public:
    std::string name;
    void speak(){ std::cout << name << std::endl; }
    void    go(){ std::cout << name << " is go" << std::endl; }
};

using work = void (Robot::*) ();

void Invoke(Robot r, work func) {
  (r.*func)();
};

int main(){
    Robot bowls[2] = { std::string("one"), std::string("two") };
    Invoke(bowls[0], & Robot::speak);
    Invoke(bowls[1], & Robot::go);
}
```

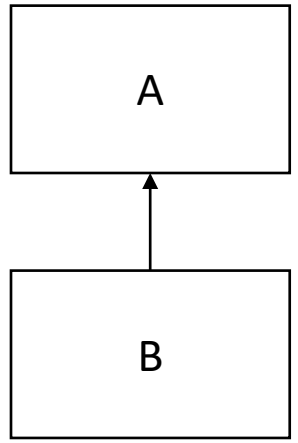
Если нужен указатель на статический метод, то используйте указатель на обычную функцию.

Подробнее про указатели на члены класса можно прочитать тут.

Указатель в ООП

В отличие от других типов данных классы можно связать через наследование.
Если 2 или больше классов связаны через наследование, то между ними возможно преобразование вниз или вверх по иерархии.
Преобразование типа указателя вверх по иерархии классов (UpCast) - безопасно, с точки зрения работы с памятью, поэтому оно выполняется неявно.
Преобразование вниз по иерархии (DownCast) может быть опасно, поэтому такое преобразование можно выполнить только явным образом (программист берёт ответственность на себя)

```
struct A{
};

struct B: A{
};

A* a = new A;
B* b = new B;

1) a = b;  // Ошибок нет, т.к. преобразование вверх по иерархии (UpCast)
2) b = a;  // Ошибка! Преобразование вниз по иерархии (DownCast)
3) b = (B*) a;  // Ошибок нет, т.к. используется явное преобразование в стиле C
4) b = static_cast<B*>(a);  // Ошибок нет, т.к. используется явное преобразование в стиле C++
```



Почему преобразование вверх по иерархии безопасно (с точки зрения памяти)?

Что подразумевается под безопасностью с точки зрения памяти?

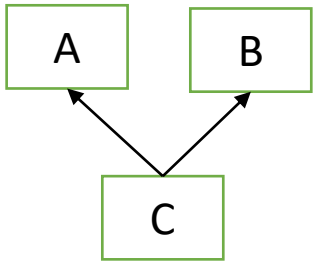
Опасно это:
- портить данные других переменных, т.к. приведёт к не предсказуемому поведение программы;
- залезать в память не выделенную для программы. Приведёт к падению программы с ошибкой сегментации

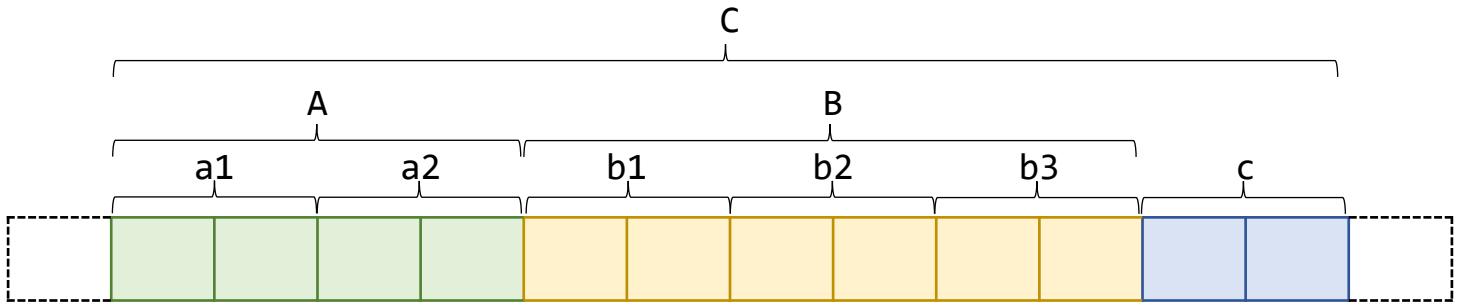
Почему преобразование вверх по иерархии безопасно (с точки зрения памяти)?

Если класс C является наследником классов A и B, то в памяти занятой объектом класса C можно выделить фрагмент занимаемый полями A, фрагмент занимаемый полями B и добавочную часть - личные поля C.

Т.е. C как бы собран из полноценных объектов класса A и B и полей класса C

```
struct A{
    short a1;
    short a2;
};

struct B{
    short b1;
    short b2;
    short b3;
};

struct C: A, B{
    short c;
};

C object;
```

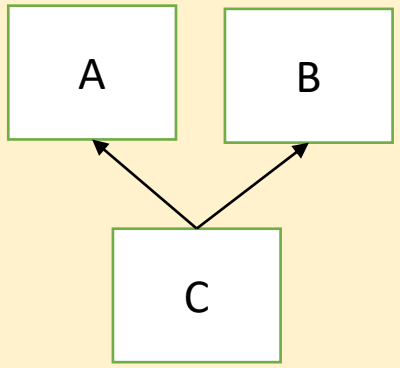




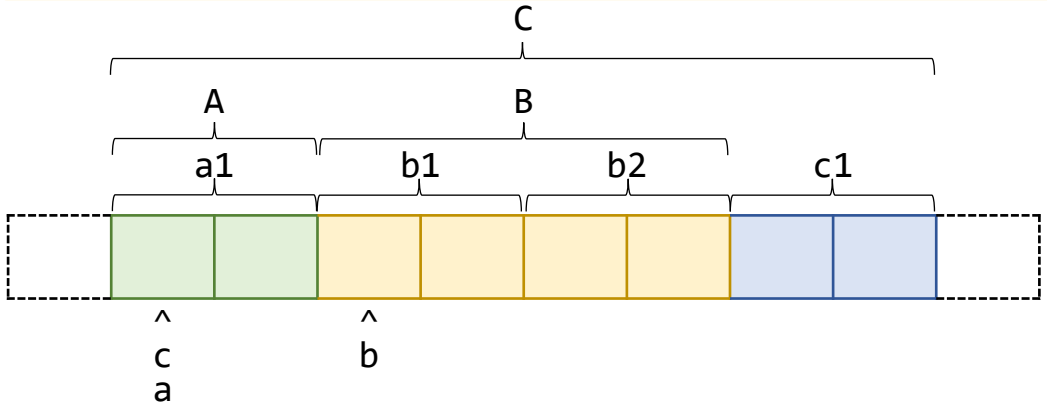
Почему преобразование вверх по иерархии безопасно (с точки зрения памяти)?

Из за такой структуры размещения в памяти, в наследнике, всегда возможно найти точку начиная с которой размещён каждый из предков и установить на него указатель.
Т.е. в процессе преобразования адреса потомка к адресу предка результирующий адрес может быть сдвинут вперёд

```
struct A{
    short a1;
};

struct B{
    short b1;
    short b2;
};

struct C: A, B{
    short c1;
};

C object;
A* a = &object;
B* b = &object;
cout << &object << '\n' << a << '\n' << b;
```



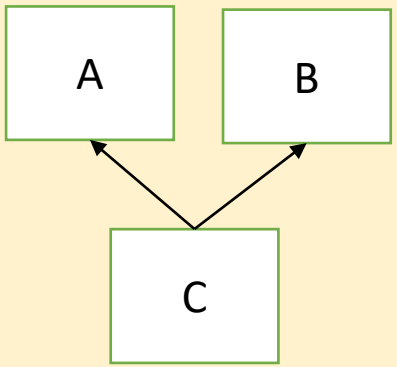

```
Вывод:
....60
....60
....62

Такой вывод объясняется тем, что "подобъект" B внутри C
расположен после A и для корректной работы с ним нужно
сдвинуть указатель начала на длину объекта A вперёд
```



Почему преобразование вниз по иерархии может быть опасно (с точки зрения памяти)?

Как упоминалось выше, в процессе преобразования типа указателя с адресом могут происходить различные преобразования из-за которых мы можем не намеренно получить доступ к чужим данным.

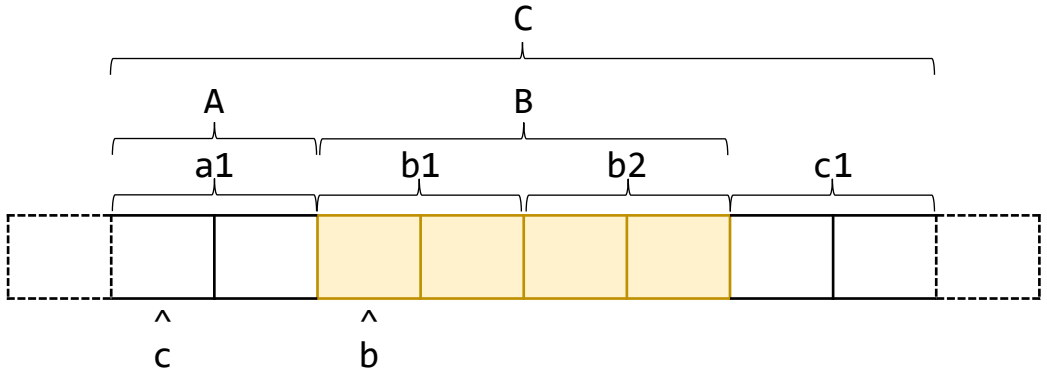
```
struct A{
    short a1;
};

struct B{
    short b1;
    short b2;
};

struct C: A, B{
    short c1;
};

B object;
C* c = static_cast<C*>(&object);
cout << &object << '\n' << c << '\n' << &c->c1;
```



Вывод:

```
....902
....900
....906
```

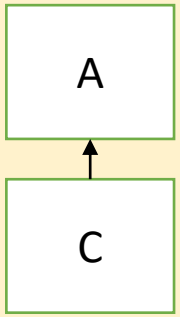
Как видно, чтобы получить адрес объекта класса C адрес B был сдвинут влево на длину равную размеру первого предка. Теперь при попытка обратиться к полям a1 и c1 мы попадём в участки памяти реально нам не принадлежащие. Там могут быть например другие переменные или временно сохранённые значения регистров процессора или вообще всё что угодно



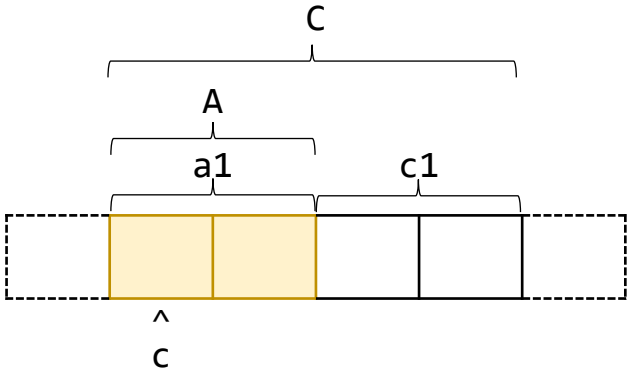
Почему преобразование вниз по иерархии может быть опасно (с точки зрения памяти)?

Даже если преобразований адреса не происходит, то вероятность залезть в чужую память остаётся

```
struct A{
    short a1;
};

struct C: A{
    short c1;
};

A object;
C* c = static_cast<C*>(&object);
cout << &object << '\n' << c << '\n' << &c->c1;
```



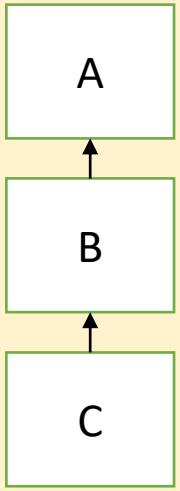

```
Вывод:
....600
....600
....602

Как видно, смешения адреса не произошло, но при
попытке получить доступ к c1 мы выходим за
пределы реально выделенной памяти и снова
попадаем неизвестно куда
```



Когда преобразование вниз по иерархии безопасно?

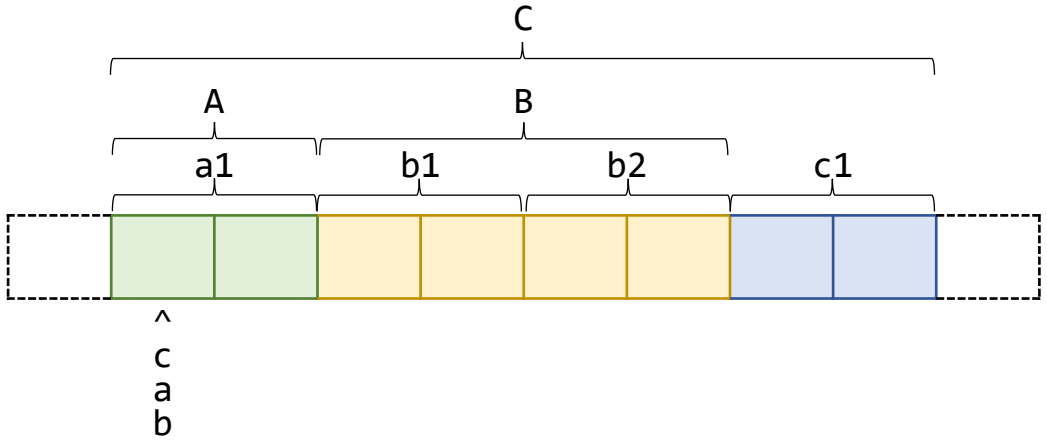
Чтобы преобразование было безопасно, **реально** размещённый в памяти объект должен быть ниже или на том же уровне иерархии, что и тип указателя к которому выполняется преобразование

```
struct A{
    short a1;
};

struct B: A{
    short b1;
    short b2;
};

struct C: B{
    short c1;
};

C object;
A* a = &object;     // Здесь OK т.к. UpCast
B* b = static_cast<B*>(a);
cout << &object << '\n' << a << '\n' << b;
```




```
Вывод:
....60
....60
....60
```

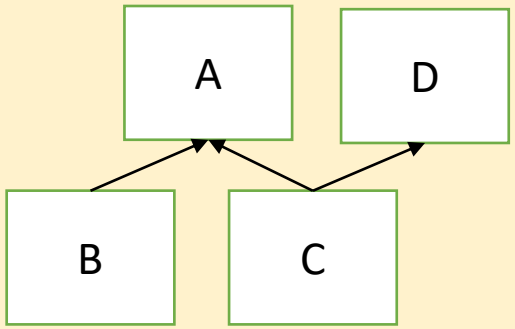
В этом примере класс C наследуется только от B, который наследуется от A.
Т.е. реальный объект на который указывает A принадлежит классу C, то преобразование вниз к B безопасно. т.к. в памяти присутствуют все данные для корректной работы B



Указатель на класс виртуальным методом

Если класс содержит хотя бы один виртуальный метод, то, кроме членов-данных в памяти каждого объекта хранится таблица виртуальных методов, поэтому для преобразования вниз вместо static_cast нужно использовать dynamic_cast, который перед преобразованием сопоставляет таблицу виртуальных методов исходного и конечного классов и в случае их не совместимости возвращает нулевой указатель:

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

struct B: A{
};

struct D{
};

struct C:A, D{
};

C object;
A* a = &object;  // OK, т.е. UpCast
B* b1 = static_cast<B*>(a);
B* b2 = dynamic_cast<B*>(a);
C* c = dynamic_cast<C*>(a);

cout << (b1 == nullptr) << '\n' << (b2 ==nullptr) << '\n' << (c ==nullptr);
```




```
Результат:
false
true
false
```

Здесь в классе A присутствует виртуальный деструктор, который наследуется его классами потомками.
Попытка преобразования из A в B при помощи static_cast проходит успешно, хотя это потенциальная ошибка, т.к. реальный объект под указателем это объект класса C.
Попытка сделать тоже самое, но через dynamic_cast проваливается и это правильное поведение.
Преобразование из A обратно в C через dynamic_cast, как и ожидалось проходит успешно

Ссылки

Что такое ссылка

В C++ существует два вида ссылок: l-value ссылки (например `int&`) и r-value ссылки (например `int&&`). Далее под словом ссылка будет подразумеваться l-value ссылка.
Ссылку в C++ — можно воспринимать, как альтернативное имя для существующего объекта. Под капотом ссылка может быть реализована как указатель, поэтому ссылки часто называют саморазименовывающимися указателями.
Основное назначение ссылок – более удобная форма записи по сравнению с указателями.

Объявление

Переменная-ссылка объявляется как: Тип_данных & Имя_ссылки:

```
int &j = /* инициализатор */;
string &k = /* инициализатор */;
vector<string> &m = /* инициализатор */;
```

Количество пробелов слева и справа от амперсанда значения не имеет:

```
int&  i   = /* инициализатор */;
int  &  i = /* инициализатор */;
int  &i   = /* инициализатор */;
```

При объявлении нескольких переменных-ссылок одновременно, амперсанд должна находится у каждой, иначе переменная воспринимается как переменная-значение:

```
int v = 1;
int &a = v, b, c;  // a - ссылка, b и c – просто int-ы
int &i = v, j, &k = v; // i и k - ссылки
```

Инициализация

Создать не инициализированную ссылку нельзя. Синтаксис требует, чтобы инициализатор обязательно присутствовал.

```
int& a =  /* инициализатор */;
double& b = /* инициализатор */;
```

После создания, ссылка не может быть привязана к другому объекту, поэтому для ссылок нет никакого аналога `nullptr`.

```
int value = 10;
int value2 = 20;
int& ref = value; // Ссылка ref связана с value.
ref = value2;     // Аналогично value = value2
```

Для инициализации ссылки после символа = в объявлении достаточно указать объект походящего типа. Никаких дополнительных действий не требуется.
Ссылка может быть связана только с объектом у которого есть постоянный адрес в памяти. Обычную ссылку на временный объекта создать нельзя:

```
int& ref = 2; // Ошибка! 2 – временный объект
```

Виды ссылок

Не типизированная ссылка

Не типизированных ссылок не существует:

```
void& a;          // Ошибка! Нельзя объявить ссылку на void
```

Типизированная ссылка

В качестве типа ссылки может быть использован любой тип данных, кроме массива, т.к. его нельзя инициализировать:

```
bool& a = /* инициализатор */;
string& b = /* инициализатор */;
vector<string>& c = /* инициализатор */;
vector<vector<int>>& d = /* инициализатор */;
map<string, string>& e = /* инициализатор */;
```

```
int& f[10]; // Ошибка! Массив ссылок
int (&f)[10] = /* инициализатор */; // Ок! Ссылка на массив
double (&n)(double, double) = /* инициализатор */;
```

Объектные обёртки над массивами типо `vector, map` и любые другие также не могут иметь ссылку как параметр шаблона.

R-value ссылка

Ссылки данного типа мы не будем рассматривать в этот раз, хотя они довольно часто встречаются.

В отличие от l-value ссылок, они предназначены для объектов время жизни которых подходит к концу и не могут работать с объектами время жизни которых пока не заканчивается:

```
int a = 10;
int&& ref = a;    // Ошибка!

int&& ref2 = 2+2; // OK!
```

Ссылки данного типа были добавлены в C++ именно для того, чтобы отличать одни объекты от других.

Ссылки и константность

Ссылки и константность

В отличие от указателя, ссылка не может быть константой, поэтому квалификатор const не применим к ссылке:

```
int one = 1;
int& const r = one; // Ошибка
```

При этом объект для которого создаётся ссылка – могут быть константой, в этом случае у нас будет ссылка на константу. Такая ссылка может только читать данные, но не изменять:

```
int one = 1;
const int& r = one;
r += 5;   // Ошибка! Попытка изменить константу
```

Если ссылка **гарантирует** константность данных, то она может сохранить ссылаться как константы, так и не константы.
В обратном случае, только на не константы:

```
int one = 1;
const int two = 2;
int& r1 = one; // OK!
int& r2 = two; // Ошибка! r2 - не гарантирует
               // константности данных
```

В отличие от НЕ константной ссылки, константане могут ссылаться на временные объекты, тем самым продлевая им жизнь, пока живёт сама ссылка:

```
int& r1 = 2;         // Ошибка!
const int& r2 = 2;   // OK!
```

Размер ссылки

Размер ссылки

В общем случае измерить размер ссылки возможности нет, т.к. ссылка прозрачно преобразуется в объект на который ссылается. Компилятор вправе вообще ничего не создавать когда мы создаём ссылку. В некоторых случаях вместо ссылки будет создан указатель, но это зависит от компилятора, поэтому в общем случае ссылка не имеет размера:

```
cout << sizeof(int &) << '\n'          // 4
     << sizeof(double &) << '\n'       // 8
     << sizeof(vector<int> &) << '\n'; // 24
```

Преобразование типа ссылки

Преобразование типа ссылки

Как правило преобразование типа ссылки является не допустимой (и бессмысленной) операцией, кроме преобразования между объектами связанными иерархией наследования и каста не константной ссылки в константную:

```
double pi = 3.14;
int& ref = static_cast<int&>(d); // Ошибка! Нельзя привести ссылку

int& r = (int&)pi;   // OK! Аналог reinterpret_cast<int&>
cout << r << endl;   // 1374389535
```

Операции с ссылками

Операции с ссылками

Для ссылок такого понятия нет, т.к. ссылка прозрачно преобразуется в переменную к которой привязана.

Варианты применения указателей

Передача аргументов в функцию

Основное предназначение ссылок – это сделать код более чистым по сравнению с указателями. Поэтому, как правило ссылки применяются при передаче данных в функцию.

```
void swap(int* a, int* b) {
  int t = *a;
  *a = *b;
  *b = t;
};

int a = 10;
int b = 20;
swap(&a, &b);
```

```
void swap(int& a, int& b) {
  int t = a;
  a = b;
  b = t;
};

int a = 10;
int b = 20;
swap(a, b);
```

Как видно, после применения ссылок код избавился от лишних звёздочек и амперсандов, и стал визуально проще.

Нет пользы от передачи по ссылке маленьких типов (только в том случае, если вы хотите их изменить в точке вызова). Добавление квалификатора const к параметру-ссылке позволит передавать в функцию не только переменные, но и временные объекты, поэтому часто в параметрах можно встретить константане ссылки.

Передача аргументов в функцию - массив

Как правило ссылки не применяются при передаче в функцию статических массивов, т.к. от этого не особенном много пользы, скорее наоборот, т.к. под каждый размер массива придётся создавать одесную функцию.

```
int sum(int (&arr)[4], int size){
   int accum = 0;
   for(int i=0; i<size; i++) accum += arr[i];
   return accum;
}

int a[4] = {1, 2, 3, 4};
cout << sum(a, 4); // 10
```

Например в этом примере, функция sum будет работать только с массивами размером 4 элемента и никакими другими.

Передача аргументов в функцию - функция

С функциями таже история, что и с массивами. Пользы от использования ссылок на функции нет:

```
int sqr(int num) {
    return num * num;
}

int Do(int a, int (&p)(int)){
    return p(a);
}

cout << Do(5, sqr); // 25;
```

Передача аргументов в функцию - aliasing

Для ссылок также как и для указателей актуальна проблема aliasinga, т.к. диапазоны памяти с которыми связаны ссылки могут пересекаться.
Решение такое же как и для указателей.

Возврат ссылки из функции

В C++, редко используют возврат ссылок из функции (ещё реже чем указатели):

```
int& foo(){
    int a = 10;
    return a;
}

int p = foo(); // Висячая ссылка, т.к. переменная a удалена
```

```
int& newArray(int size, int value=0){
    int* res = new int[size];
    for (int i=0; i<size; i++) res[i] = value;
    return *res;
}
```

Полиморфизм через наследование

На указателях и ссылках основывается одна из реализаций идеи полиморфизма в C++, а точнее реализация полиморфизма через наследование.
Об этом подробнее будет в блоке про указатели и ООП.

Как читать тип ссылок

Декларация типов C

Т.к. нет возможности создавать что-то типо такого:

```
const int&& const && const p4 = p3;
```

то нет необходимости вводить какие-то специальные правила чтения ссылок.

Указатель в ООП

Доступ к полю структуры/класса

Если тип ссылки – структура или класс

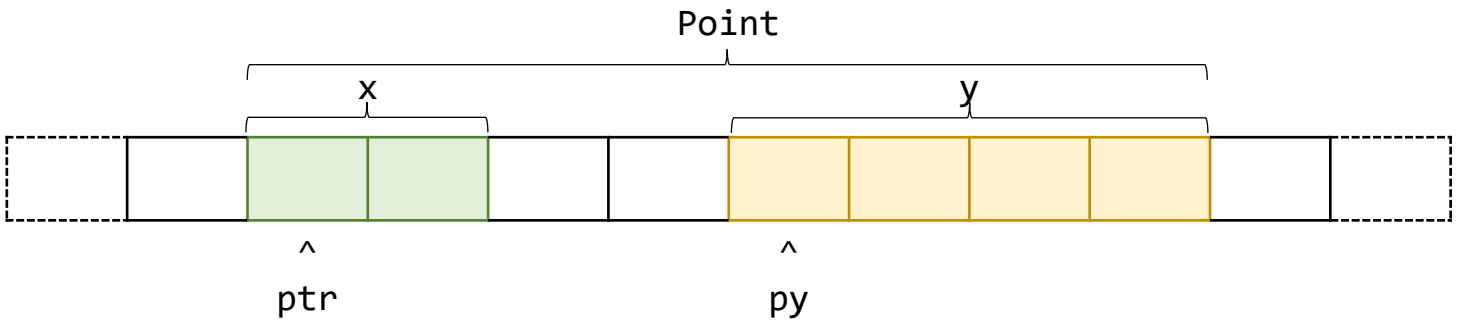
```
struct Point{
    short x;
    int y;
};

Point p{0, 0};
Point& ref = p;
```

Можно получить доступ к её полю или **методу**:

```
ref.x = 100;
```

Кроме того, можно напрямую получить ссылку на поле (но не на метод) класса:

```
int& ry = p.y;
ry = 100;
```



Доступ к полю структуры/класса - static

Ссылку на статические поля можно брать как через экземпляр класса, так и без него. Физически статические поля располагаются НЕ рядом с остальными, в той же области памяти, что и глобальные переменные:

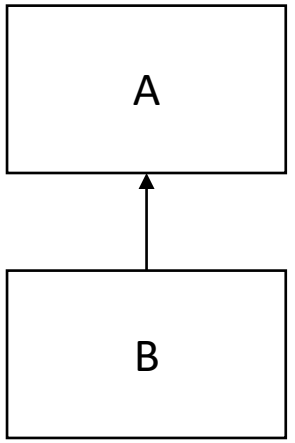
```
struct Point{
  short x;
  int y;
  static int z;
};

int Point::z = 10;

Point p;
int& ref = Point::z;
std::cout << ref<< std::endl;
```

```
10
```

Ссылка в ООП

В отличие от других типов данных классы можно связать через наследование.
Если 2 или больше классов связаны через наследование, то между ними возможно преобразование вниз или вверх по иерархии.
Преобразование типа ссылки вверх по иерархии классов (UpCast) - безопасно, с точки зрения работы с памятью, поэтому оно выполняется неявно.
Преобразование вниз по иерархии (DownCast) может быть опасно, поэтому такое преобразование можно выполнить только явным образом (программист берёт ответственность на себя)

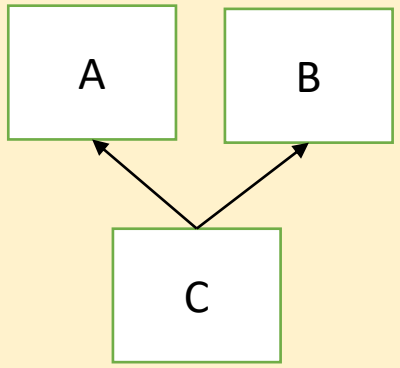
```
struct A{
};

struct B: A{
};

A& a = *new A;
B& b = *new B;

1) A& a2 = b;  // Ошибок нет, т.к. преобразование вверх по иерархии (UpCast)
2) B& b2 = a;  // Ошибка! Преобразование вниз по иерархии (DownCast)
3) B& b2 = (B&) a;  // Ошибок нет, т.к. используется явное преобразование в стиле C
4) B& b2 = static_cast<B&>(a);  // Ошибок нет, т.к. используется явное преобразование в стиле C++
```



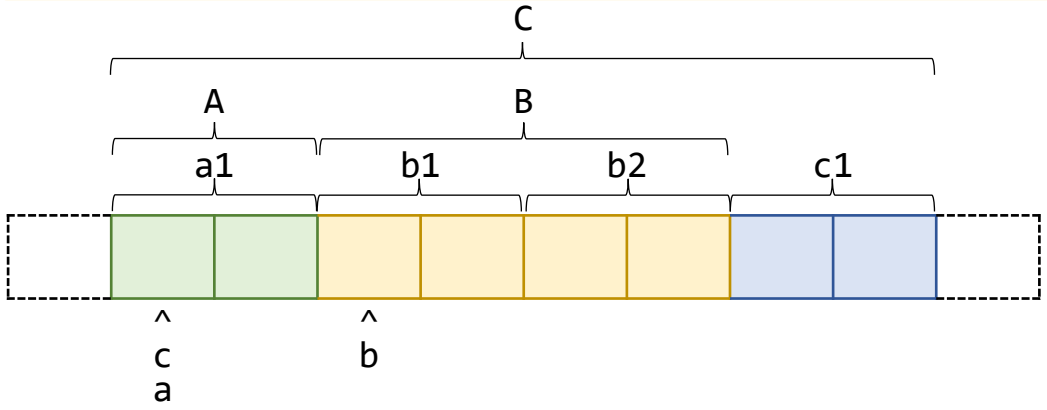
Почему преобразование вверх по иерархии безопасно (с точки зрения памяти)?

Из за такой структуры размещения в памяти, в наследнике, всегда возможно найти точку начиная с которой размещён каждый из предков и установить на него ссылку.
Т.е. в процессе преобразования адреса потомка к адресу предка результирующий адрес может быть сдвинут вперёд

```
struct A{
    short a1;
};

struct B{
    short b1;
    short b2;
};

struct C: A, B{
    short c1;
};

C object;
A& a = object;
B& b = object;
cout << &object << '\n' << &a << '\n' << &b;
```



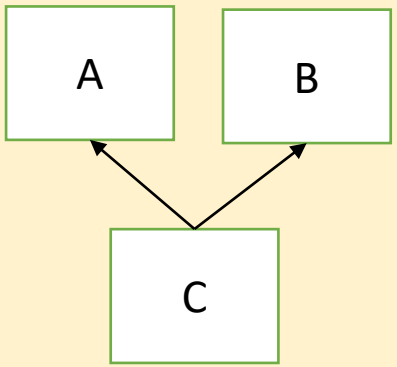

```
Вывод:
....60
....60
....62

Такой вывод объясняется тем, что "подобъект" B внутри C
расположен после A и для корректной работы с ним нужно
сдвинуть указатель начала на длину объекта A вперёд
```



Почему преобразование вниз по иерархии может быть опасно (с точки зрения памяти)?

Как упоминалось выше, в процессе преобразования типа ссылки с адресом могут происходить различные преобразования из-за которых мы можем не намеренно получить доступ к чужим данным.

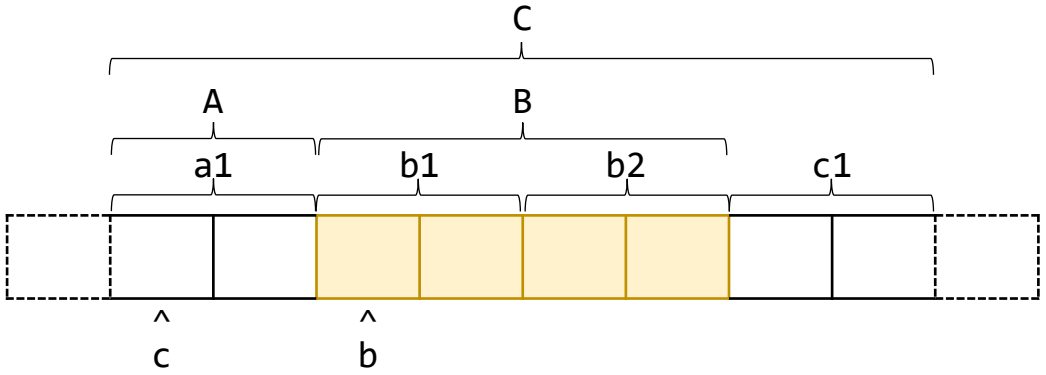
```
struct A{
    short a1;
};

struct B{
    short b1;
    short b2;
};

struct C: A, B{
    short c1;
};

B object;
C& c = static_cast<C&>(object);
cout << &object << '\n' << &c << '\n' << &c.c1;
```



Вывод:

```
....902
....900
....906
```

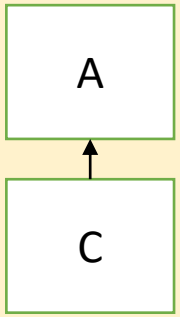
Как видно, чтобы получить адрес объекта класса C адрес B был сдвинут влево на длину равную размеру первого предка. Теперь при попытка обратиться к полям a1 и c1 мы попадём в участки памяти реально нам не принадлежащие. Там могут быть например другие переменные или временно сохранённые значения регистров процессора или вообще всё что угодно



Почему преобразование вниз по иерархии может быть опасно (с точки зрения памяти)?

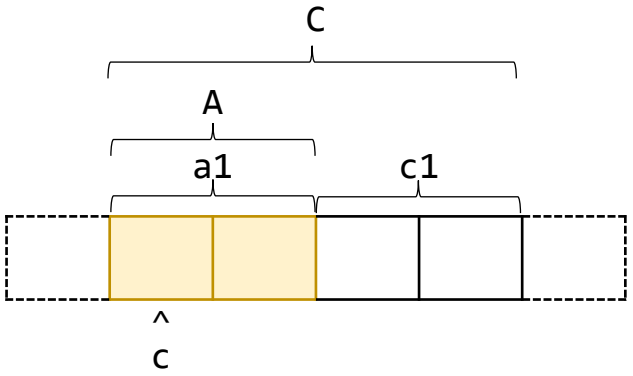
Даже если преобразований адреса не происходит, то вероятность залезть в чужую память остаётся

```
struct A{
    short a1;
};

struct C: A{
    short c1;
};

A object;
C& c = static_cast<C&>(object);
cout << &object << '\n' << &c << '\n' << &c.c1;
```



Вывод:
```
....600
....600
....602
```

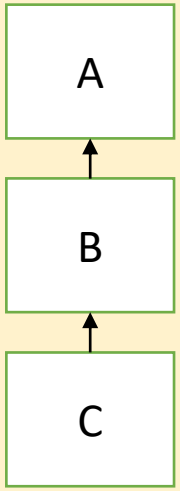
Как видно, смешения адреса не произошло, но при попытке получить доступ к c1 мы выходим за пределы реально выделенной памяти и снова попадаем неизвестно куда



Когда преобразование вниз по иерархии безопасно?

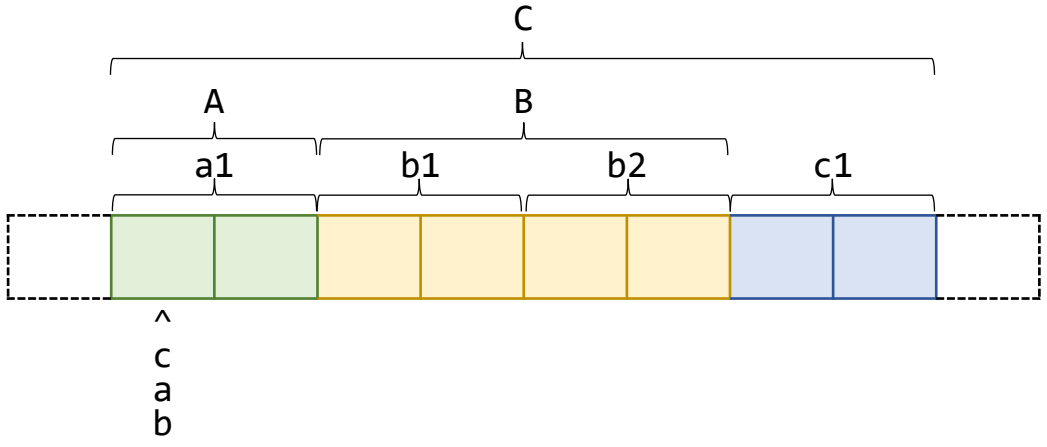
Чтобы преобразование было безопасно, **реально** размещённый в памяти объект должен быть ниже или на том же уровне иерархии, что и тип ссылки к которой выполняется преобразование

```
struct A{
    short a1;
};

struct B: A{
    short b1;
    short b2;
};

struct C: B{
    short c1;
};

C object;
A& a = object;    // Здесь OK т.к. UpCast
B& b = static_cast<B&>(a);
cout << &object << '\n' << &a << '\n' << &b;
```




```
Вывод:
....60
....60
....60
```

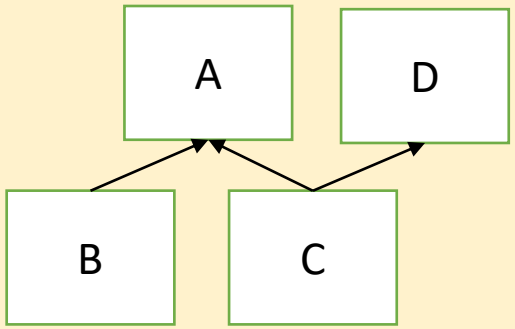
В этом примере класс C наследуется только от B, который наследуется от A.
Т.е. реальный объект на который указывает A принадлежит классу C, то преобразование вниз к B безопасно. т.к. в памяти присутствуют все данные для корректной работы B



Указатель на класс виртуальным методом

Если класс содержит хотя бы один виртуальный метод, то, кроме членов-данных в памяти каждого объекта хранится таблица виртуальных методов, поэтому для преобразования вниз вместо static_cast нужно использовать dynamic_cast, который перед преобразованием сопоставляет таблицу виртуальных методов исходного и конечного классов и в случае их не совместимости кинет исключение `std::bad_cast`:

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

struct B: A{
};

struct D{
};

struct C:A, D{
};

C object;
A& a = object; // OK, т.е. UpCast
B& b1 = static_cast<B&>(a);
B& b2 = dynamic_cast<B&>(a);
C& c = dynamic_cast<C&>(a);
std::cout << (&b1 == nullptr) << '\n' << (&b2 ==nullptr) << '\n' << (&c
==nullptr);
```




```
Результат:
false
Исключение
false

Здесь в классе A присутствует виртуальный деструктор, который наследуется его классами потомками.
Попытка преобразования из A в B при помощи static_cast проходит успешно, хотя это потенциальная ошибка, т.к. реальный объект под указателем это объект класса C.
Попытка сделать тоже самое, но через dynamic_cast проваливается и это правильное поведение.
Преобразование из A обратно в C через dynamic_cast, как и ожидалось проходит успешно
```