Lesson 4

Структуры. Unions. Enumerations. Динамическое распределение памяти/Динамические массивы. Препроцессор. Директивы препроцессора.

Динамическое распределение памяти.

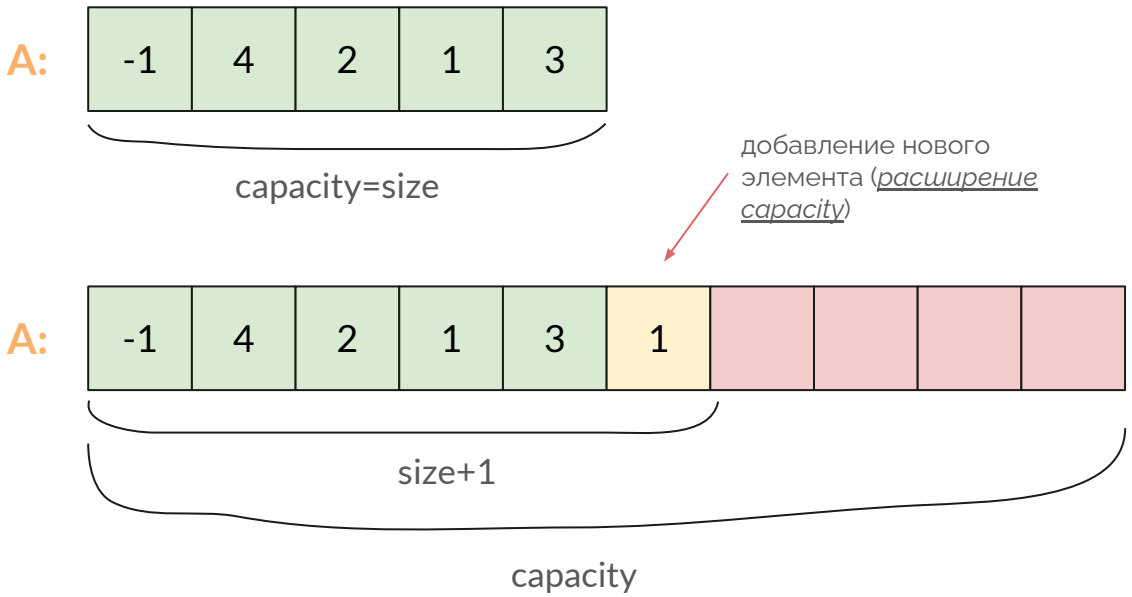
В языке программирования C++ различают следующие способы распределения памяти:

1) **Static** (fixed) выделение памяти. В этом случае память выделяется только один раз во время компиляции. Например: `int M[100];`
2) **Dynamic** выделение памяти. В этом случае используется комбинация операторов new и delete. Оператор new выделяет память для переменной (массива) в специальной области памяти. Оператор удаления освобождает выделенную память. Например:

```
// allocation of memory by the new operator
int * p; // pointer to int
p = new int; // allocate memory for the pointer
*p = 15; // write values into memory
```

```
// use of memory, allocated for the pointer
int d;
d = *p; // d = 15
// free the memory, allocated for the pointer - obligatory
delete p;
```

Динамические массивы.



size - текущий размер массива как бы;

capacity - допустимый размер (вместимость);

Динамические массивы.#1

```
int size, capacity;

// Запрос размера массива
std::cout << "Введите размер массива: " ;
std::cin >> size;

// Установка начальной емкости
capacity = size;

// Выделение памяти под массив
int *arr = new int[capacity];

// Инициализация массива (пример)
for (int i = 0; i < size; ++i) {
  arr[i] = i; // Пример инициализации массива
}
```

```
delete[] arr;
```

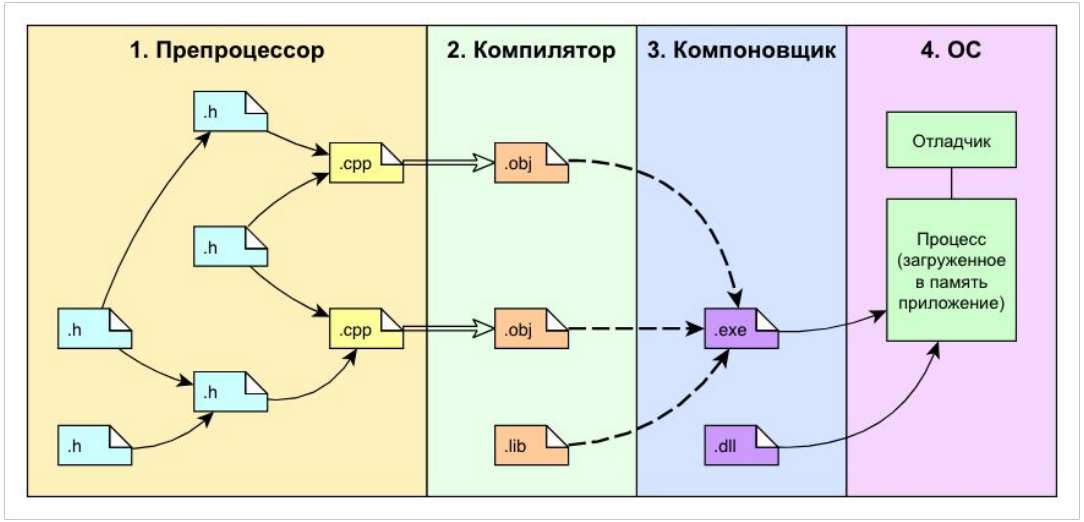
Динамические массивы.#2

```
void fillArray(int* arr, int size) {
  // Заполнение массива
  for (int i = 0; i < size; ++i) {
      std::cout << "Введите элемент " << i << ": ";
      std::cin >> arr[i];
  }
}
```

Препроцессор. Директивы препроцессора. Макросы.

Препроцессор в C++ — это часть компилятора, которая выполняет специальные инструкции, называемые директивами препроцессора, до того, как компилятор начинает фактическую компиляцию исходного кода. **Директивы препроцессора** начинаются с символа **#** и исп-ся для предварительной обработки исходного кода до компиляции. Частые:

- **#include** - исп-ся для включения содержимого файла в программу.
- **#define** - исп-ся для создания макросов, то есть абстракций для фрагментов кода.
- **#ifdef, #ifndef, #endif** - исп-ся для условной компиляции в зависимости от наличия или отсутствия определенных макросов.
- **#pragma** - исп-ся для управления компилятором с помощью специфичных для платформы или компилятора инструкций. `#pragma warning(disable: 4996)` *- отключ. предупреждение 4996, которое связано с исп-м устаревших или небезопасных функций в C++.*


1. Препроцессор
2. Компилятор
3. Компоновщик
4. ОС
.h
.h
.h
.h
.h
.cpp
.cpp
.obj
.obj
.lib
.exe
.dll
Отладчик
Процесс (загруженное в память приложение)

#define.1

```
#define PI 3.14159 // определение макроса для числа Пи
#define DX 5
#define MX 10.1
```

#define.2

```
// Пример макроса, имитирующего функцию
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define ADD(a, b) (a + b)
...
int x = 5, y = 10;
int max_value = MAX(x, y); // 10
```

#ifdef, #ifndef, #endif

```cpp
#include <iostream>
// Определение макроса DEBUG
#define DEBUG

int main() {
#ifdef DEBUG
    std::cout << "Отладочная информация: программа начинает выполнение" << std::endl;
#endif

    // Некоторый код программы

#ifdef DEBUG
    std::cout << "Отладочная информация: программа завершает выполнение" << std::endl;
#endif
    return 0;
}
```

#ifdef, #ifndef, #endif (продолжение)

В примере, если макрос **DEBUG** определен (с помощью **#define DEBUG**), то блоки кода между **#ifdef DEBUG** и **#endif** будут включены в компиляцию. Если макрос DEBUG не определен, эти блоки кода будут исключены из компиляции.

Если вы хотите проверить, что макрос не определен, используйте **#ifndef** (**ifndef** означает "если не определено").

И др.

- **NDEBUG** - исп-ся для отключения проверок, связанных с отладкой, в вашем коде;
- **WIN32** - может исп-ся для условной компиляции кода, который зависит от ОС Windows;
- **... (UNIX,..)**

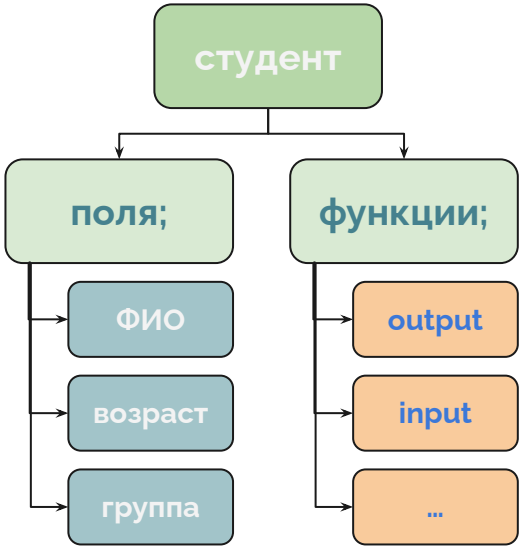
Структуры.

Структура — это объединение одной или нескольких переменных в одной области памяти с одинаковым именем. Переменные, включенные в структуру, называются полями структуры. Такие переменные могут быть разных типов и должны иметь уникальное имя. Существует два типа конструкций:

1) Нативные - структуры (*классические*)
2) Управляемые структуры (*объявленные как ссылка или значение*)

Синтаксис классической структуры данных:

```
struct type_structure name {
    type element_1;
    type element_2;
    ...
    type element_N;
} struct_variables;
```



Пример структур.

Чтобы описать **структурную переменную**, необходимо сначала определить шаблон структуры. Шаблон структуры также называется форматом структуры. *Например*:

```
struct Worker {
    int code;   // code worker
    char name[50]; // name worker
    int hours; // hours worked
    float cost;
};
```

```
// description of the variable by the previously created template
Worker w1;
w.code = 225;
strcpy(w.name, "John");
w.hours = 21;
w.cost = 10.75f;
```

Важное о структурах.

1) **Поля структур** (переменные) автоматически становятся **публичными**, т.е. все методы имеют к ним доступ.
2) Вы также можете создавать функции в структурах, которые являются методами.

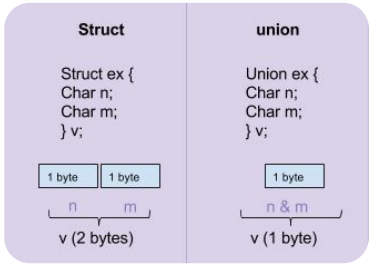
```
struct Book {
  char title[100];
  char author[100];
  int year;
  double price;

  void Print()
  {
    cout << "title = " << title << endl;
    cout << "author = " << author << endl;
    cout << "year = " << year << endl;
    cout << "price = " << price << endl;
  }
};
```

Unions.

Объединение — это группа переменных, которые используют одно и то же пространство памяти. В зависимости от интерпретации осуществляется доступ к той или иной переменной объединения. Объединение позволяет представить данные в компактной форме, которую можно изменить. Объявление объединения (типа объединения или шаблона объединения) начинается с ключевого слова объединения:

```
union type_unit name
{
    type variable1;
    type variable2;
    ...
    type variableN;
};
```



Длина объединения — это объем памяти в байтах, выделенный для одной переменной этого типа объединения. Длина объединения рассчитывается как максимальная из всех длин отдельных полей шаблона.

Enumerations.

Enum Тип данных используется для создания именованных идентификаторов, которые представляют собой некоторое целочисленное постоянное значение. Перечисления позволяют объединять группы константных значений в одно целое число. Синтаксис объявления перечисления следующий:

```
enum TypeName
{
    name1 [=const1],
    name2 [=const2],
    ...
    nameN [=constN]
};
```

Существует **явная** инициализация переменных, т.е присвоение ей некоторого значения, и **неявная**, где значение берется из индекса (первая переменная — ноль, вторая — первая и т.д.).

Thank you for your attention!