Ответы на вопросы по О-ОП

1. Структура программы на С++.

- Программа на C++ состоит из набора инструкций. Каждая инструкция выполняет определенные действия. В конце инструкции ставится точка с запятой. Данный знак указывает на завершение инструкции. Набор инструкции может представлять блок кода. Блок кода заключается в фигурные скобки, а инструкции между ними.

- Программа, написанная на Cи или С++ с использованием средств процедурного программирования, в общем случае включает:

· команды препроцессора;

· объявления переменных, констант и типов;

· прототипы функций;

· определения функций;

***Препроцессор***– это программа, которая обрабатывает исходный текст до компилятора. Посредством команд препроцессора, например, определяется перечень файлов, содержащих прототипы стандартных функций из библиотек, которые должны быть подключены

при компиляции программы.

- Функция **main()** – основная программа. Именно с неё начинается выполнение приложения.

- Команды препроцессора – команды, выполняющиеся перед компиляцией программы.

Пример:

<Команды препроцессора>

[<Объявление типов, переменных, и констант>]

[<Объявление функций>]

<Описание функции main()>

[<Описание других функций>]

Заметка:

<…> - конструкции на C++

[<…>] – конструкции, которые не обязательно присутствуют в коде

1. Скалярные типы данных С++. Определение констант и переменных.

Все типы делятся на:

1) скалярные (простые)

2) структурированные (составные)

Все допустимые в языке типы подразделяются на две большие группы: скалярные и структурированные.

К стандартным скалярным типам относятся данные целочисленного, вещественного, булевского типов и указатели.

К структурированным типам относятся классы, структуры строки, массивы.

|  |  |  |
| --- | --- | --- |
| Тип | Байт | Диапазон |
|  |  | Целочисленный (логический) тип данных |
| bool | 1 | 0 / 255 |
|  |  |  |
|  |  | Целочисленный (символьный) тип данных |
| char | 1 | -128 / 127 |
|  |  |  |
|  |  | Целочисленные типы данных |
| short int | 2 | -32 768    /    32 767 |
| unsigned short int | 2 | 0  /  65 535 |
| int | 4 | -2 147 483 648   /   2 147 483 647 |
| unsigned int | 4 | 0     /     4 294 967 295 |
| long int | 4 | -2 147 483 648    /    2 147 483 647 |
| unsigned long int | 4 | 0     /     4 294 967 295 |
|  |  |  |
|  |  | Типы данных с плавающей точкой |
| float | 4 | -2 147 483 648.0  / 2 147 483 647.0 |
| long float | 8 | Очень много. Просто поверь |
| double | 8 | Очень много. Просто поверь |

Для представления константы в Си использовалась только директива препроцессора #define:

#define MAX 100

Но использование const имеет несколько преимуществ по сравнению с #define.

· При объявлении константы с использованием const явно указывается тип величины.

· Идентификаторы const подчиняются тем же правилам, что и переменные за исключением того, что переменные могут изменяться в процессе выполнения программы, а константы остаются неизменными. Можно создавать константы с различной областью видимости.

1. Операции над скалярными данными С++. Приоритеты операций.

|  |  |  |
| --- | --- | --- |
| **Тип операции** | **Операции** | **Приоритет** |
| Унарные | **!, ++, --, +, -** | **Высший** |
| Арифметические(бинарные) | Мультипликативные **\*, /, %** . Аддитивные **+, -** |  |
| Отношения | Неравенства **<**, **>**, **<=**, **>=** Равенства **==**, **!=** |  |
| Логические | И - **&&**, ИЛИ - **||**, НЕ - **!** |  |
| Условная | **? :** |  |
| Присваивания | **=, +=, -=, \*=, /=, %=** | **Низший** |

Операции в С++ принято классифицировать.

Классификация:

- **Арифметические операции**

- Унарные

- Бинарные (основные математические операции: сложение, вычитание, деление, умножение)

- **Логические Операции** (И - **&&**, ИЛИ - **||**, НЕ - **!**)

- **Операции Отношения** (Неравенства **<**, **>**, **<=**, **>=** Равенства **==**, **!=** )

- **Операции Присваивания** (**=**, **+=**, **-=**, **\*=**, **/=**, **%=** )

1. Основные управляющие операторы С++.

Управляющим называют операторы, способные изменять линейность процесса вычислений.

- **if** – условная передача управления

- **switch** – оператор выбора

- **while, do, for** – операторы циклов

- **goto, break, continue, exit** – операторы безусловной передачи управления

**Условная передача управления (if,switch)** – конструкция, позволяющая выбрать одну из возможных альтернатив в зависимости от результата условия.

**Цикл-пока (while)** – выполняется до тех пор, пока результат выражения отличен от нуля (true). Если условие нарушено сразу при входе в цикл, то тело цикла не выполнится ни разу.

**Цикл с постусловием (do…while)** – Условие проверяется после выполнения оператора тела цикла. Если условие нарушено, тело цикла выполнится хотя бы один раз.

**Оператор счетного цикла (for)** – реализует цикл, для которого известен диапазон цикла.

1. Указатели и ссылки.

**Указатель** – это переменная, в которой хранится адрес некоторого объекта программы: другой переменной, поименованной константы, подпрограммы и т.п.

**Ссылка** – это тип переменной в C++, который работает как псевдоним другого объекта или значения.

В чём же разница между указателями и ссылками?

Основное назначение указателя – это организация динамических объектов, то есть размер, которых может меняться (увеличиваться или уменьшаться). Тогда как ссылки предназначены для организации прямого доступа к тому, или иному объекту.

Главное отличие состоит во внутреннем механизме работы. Указатели ссылаются на участок в памяти, используя его адрес. А ссылки ссылаются на объект, по его имени (тоже своего рода адрес).

При создании указателя мы можем вывести именно адрес или значение переменной, на которую указывает указатель:

1. **int** var = 11;
2. **int** \* ptr; // объявление указателя, который указывает на целое
3. ptr = &var; // присваиваем указателю адрес var
4. // разыменовываем указатель и работаем не с адресом, а со значением, например \*ptr = var,   равносильно строке выше
5. cout << ptr; // выводим адрес
6. cout << \*ptr; // выводим значение ptr, выведется 11, так как мы присвоили ptr адрес var
7. Управление динамической памятью С++.
8. Операция выделения памяти под одно значение:

<Типизированный указатель> = new <Тип>[(<Значение>)];

1. Операция освобождения памяти:

delete <Типизированный указатель>;

1. Операция выделения памяти под несколько значении:

<Указатель> = new <Тип>[<Количество>];

1. Операция высвобождения памяти, выделенной под несколько значений:

Delete[] <Типизированный указатель>

Пример:

1. **int** \*ptr\_i;
2. **double** \*ptr\_d;
3. **struct** person \*human;
4. …
5. ptr\_i = **new** **int**;
6. ptr\_d = **new** **double**[10];
7. human = **new** **struct** person;
8. **delete** ptr\_i;
9. **delete**[] ptr\_d;
10. **delete** human;
11. Адресная арифметика С++.

Адресная арифметика - это способ вычисления адреса какого-либо объекта при помощи арифметических операций над указателями, а также использование указателей в операциях сравнения. Значение адреса изменяется на n, умноженное на размер элемента данных(единиц):

Пример:

<Указатель> + n тоже самое, что и <Адрес> + n \* sizeof(<Тип данных>)

Пример:

Допустим мы объявили массив – int mas[5];

В массиве mas[5], mas - это указатель на нулевой элемент массива. Если указателю присвоить имя массива, он соответственно получит значение нулевого элемента массива

Для получения доступа к элементам массива через указатель можно использовать операцию \*(mas+i) где i - номер искомого элемента, mas - адрес массива

В этом случае, прибавляя i к адресу мы смещаемся на определенное количество байтов в памяти (если i фигурирует в счётчике ). Если I = 1, тогда мы сместимся на 4 байта в памяти, или же на одну ячейку в массиве.

/\* Этот пример показывает возможности работы с адресной арифметикой в массиве. Полезен будет метод получения элементов массива через адресную арифметику при передаче массива в функцию по адресу \*/

1. Массивы С++.

**Массив** – это структура данных, представленная в виде группы ячеек одного типа, объединенных под одним единым именем. Имя массива является указателем. Отдельная ячейка данных массива называется элементом массива.

Нумерация ячеек массива всегда начинается с 0.

**Одномерный массив** – массив с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив – это массив, у которого может быть только одна строка и n-e количество столбцов.

**Двумерный массив** – это обычная таблица, со строками и столбцами. Двумерный массив – это одномерный массив одномерных массивов.

1. Строки С++. Стандартные функции, работающие со строками.

В C++ символьная строка определяется как **массив символов**, который заканчивается нуль символом (Маркер конца строки).

Char-функции:

- strlen(имя\_строки) – определяет длину указанной строки.

- strcpy(имя\_строки1, имя\_строки2) – выполняет поэтапное копирование символов из имя\_строки2 в имя\_строки1.

- strcmp(имя\_строки1, имя\_строки2) – сравнение строк.

- strcat(имя\_строки1, имя\_строки2) – суммирование строк.

-strchr(имя\_строки1, symbol) – поиск первого вхождения символа symbol в строке имя\_строки1.

1. Структурный тип С++.

**Структура** – одна из конструкций, реализующих объединение разнотипных данных. Переменные, входящие в состав структуры, называются полями структуры.

Синтаксис языка C (Он мне больше привлекателен, чем С++):

Struct [<Имя структуры>]

{<Описание полей>};

1. Функции С++. Передача параметров и возвращение результатов.

В каждой программе обязательно присутствует функция main(). Кроме нее в программу может входить n-ое количество функций, выполнение которых прямо или косвенно выполняется в функции main().

Явно – через параметры (Как мы обычно делаем).

Существует несколько способов передачи данных в функцию

1. Передача по значению – передается копия аргумента.
2. Передача по ссылке – передается сам аргумент. (Важной особенностью передачи аргументов по ссылке является то, что функ- ция имеет прямой доступ к значениям аргументов. К достоинствам ссылочного механизма также относится возможность возвращения функцией программе не одного, а множества значений.)
3. Передача по указателю - передается адрес аргумента.

Для возвращения результата из функции применяется оператор return.

1. Параметры структурных типов.

**Параметры – структуры**

Когда структура используется как аргумент функции, передается вся структура с помощью стандартной передачи по значению. Это означает, что любые изменения, внесенные в содержимое структуры внутри функции, не повлияют на структуру, используемую в качестве аргумента. Тип аргумента должен соответствовать типу параметра.

1. **struct** massiv
2. {
3. **int** n;
4. **int** sum;
5. };
7. **int** sum(massiv x);
8. // функция будем принимать в качестве аргумента структуру massiv
9. **int** main()
10. {
11. **int** show;
12. massiv mas;
13. mas.n = 1;
14. mas.sum = 0;
15. show = sum(mas); // структура как аргумент функции
16. cout « show;
17. **return** 0;
18. }
20. **int** sum(massiv x)
21. {
22. x.sum += x.n;
23. **return** x.sum;
24. }
25. Классы памяти переменных.

Класс памяти определяет:

1. Часть памяти программы, в которой размещается переменная.
2. Область действия.
3. Время жизни переменной.

Существует два класса памяти: **automatic** (автоматический) и **static** (статический).

- У переменных, имеющих класс памяти **automatic**, время жизни равно времени жизни функции, внутри которой они определены.

- У переменных, имеющих класс памяти **static**, время жизни равно времени жизни всей программы.

1. Внешние переменные

- Размещение – Глобальная память

- Область действия – Везде, где она определена

- Время жизни – Все время работы программы

1. Автоматические переменные (auto)

- Размещение – Локальная память

- Область действия – Внутри функции

- Время жизни – Все время работы функции

1. Статические переменные (static)

- Размещение – Глобальная память подпрограммы

- Область действия – Внутри подпрограммы

- Время жизни – Все время работы программы

1. Внешние статические переменные (extern static)

- Размещение – Глобальная память модуля

- Область действия – Внутри всех функций модуля

- Время жизни – С момента вызова и до возврата управления операционной системе

1. Регистровые переменные (register)

Регистровые переменные аналогичны автоматическим, но по возможности программа должна размещать их в регистровую память процессора.

- Плюсы:

Ускоряет операции над этими переменными.

-Минусы:

Свободных регистров очень мало.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Локальная** | **Статическая Локальная** | **Глобальная** |
| **Область видимости** | **Функция** | **Функция** | **Программа** |
| **Время Жизни** | **Функция** | **Программа** | **Программа** |
| **Начальное значение** | **Случайное** | **0** | **0** |
| **Область памяти** | **Стек** | **Динамическая** | **Динамическая** |
| **Назначение** | **Переменные, используемые отдельной функцией, уничтожающиеся при выходе из нее** | **Переменные, используемые отдельной функцией, но сохраняющие свои значения между вызовами функции** | **Переменные, используемые несколькими функциями** |

1. Параметры-функции.

**У функции в качестве параметров могут находиться другие функции. Одним из примеров служат указатели на функции, которые могут являться параметрами другой функции. В C++ имя функции является адресом этой функции. Механизм передачи функции в функцию через указатель можно увидеть в следующей программе.**

1. // Function\_pointer.cpp
2. // options;
4. #include <iostream>
5. #include <stdio.h>
6. **using** **namespace** std;
8. **void** check(**char** \*a, **char** \*b, **int**(\*cmp)(**const** **char** \*, **const** **char** \*));
10. ////////////////////////////////
12. **int** main(**void**)
13. {
14. **char** s1[80], s2[80]; // массивы символов
15. **int**(\*p)(**const** **char** \*, **const** **char** \*); // указатель на функцию с двумя параметрами типа \* const char
17. p = strcmp; // strcmp - библиотечная строковая функция сравнения строк
18. // присваивает адрес функции strcmp указателю p
20. cout << "Input two strings: " << endl;
21. cin >> s1;
22. cin >> s2;
23. check(s1, s2, p); /\* Передает адрес функции strcmp
24. посредством указателя p \*/
25. **return** 0;
26. }
28. **void** check(**char** \*a, **char** \*b, **int**(\*cmp)(**const** **char** \*, **const** **char** \*))
29. {
30. cout << " Check coincidence: ";
31. **if** (!(\*cmp)(a, b))  cout << " Equal ";
32. **else** cout << " Not Equal";
33. }

В данном примере указатель на функцию имеет вид: int(\*p)(\*const char, \* const char).

Это значит, что мы создали указатель на фукнцию, которая имеет два параметра типа “\*const char”. Этот указатель мы используем в качестве аргумента.

1. Правила, определяющие видимость переменных в функциях.

Параметры каждой функции действительны только внутри этой функции.

Пример:

1. **void** sum();
3. **int** main()
4. {
5. **int** i = 100; //Глобальная переменная
6. sum();
7. **return** 0;
8. }
10. **void** sum()
11. {
12. **int** i = 20; //Переменная i локальная и действует только внутри sum()
13. }
14. Пространства имен.

Конфликт имен возникает, когда два идентификатора находятся в одной области видимости и компилятор не может понять, какой из них двух использовать. Компилятор выдаст ошибку. Всё, что объявлено внутри пользовательского пространства имен, принадлежит только этому пространству имен, а не глобальному. Пространство имен определяет область кода, в которой гарантируется уникальность всех идентификаторов.

1. Компоновка модулей С++. Защита от повторной компиляции.

Модуль C++ включает два файла: заголовочный файл с расширением “.h” и файл реализации с расширением “.cpp”.

Заголовочный файл играет роль интерфейсной секции модуля. В него помещают:

1. Заголовки процедур и функций
2. Объявление переменных типов и констант.

Файл реализации представляет собой секцию реализации модуля. Содержит команды подключения используемых модулей, описание процедур и функций, а также объявление внутренних ресурсов модуля.

Дополнительные команды препроцессора позволяют исключить повторную компиляцию текста программы при многократном подключении заголовочного файла (ifndef; endif).

1. Перегрузка функций.

**Перегрузка функций** – это особенность C++, которая позволяет создать несколько функций с одним и тем же именем, но с разными параметрами. Компилятор может определить сам, какую, версию следует вызывать на основе аргументов, используемых в вызове функции. Можно определить столько перегруженных функций, сколько хотим до тех пор, пока каждая из них будет иметь свои уникальные параметры.

Пример:

1. // overload.cpp
2. // перегрузка функций
3. #include <iostream>
4. **using** **namespace** std;
5. **void** repchar(); // прототипы
6. **void** repchar(**char**);
7. **void** repchar(**char**, **int**);
8. **int** main()
9. {
10. repchar();
11. repchar('=');
12. repchar('+', 30);
13. **return** 0;
14. }
15. //---функция repchar()-----------------------------
16. // выводит на экран 45 символов '\*'
17. **void** repchar() {
18. **for**(**int** j = 0; j < 45; j++) // цикл, выполняющийся 45 раз
19. cout << '\*'; // вывод символа '\*'
20. cout << endl;
21. }
22. //--- функция repchar()----------------------------- // выводит 45 заданных символов
23. **void** repchar(**char** ch) {
24. **for**(**int** j = 0; j < 45; j++) // цикл, выполняющийся 45 раз
25. cout << ch; // вывод заданного символа
26. cout << endl;
27. }
28. //--------------------------------------------------------
29. // функция repchar()
30. // выводит заданный символ заданное число раз
31. **void** repchar(**char** ch, **int** n) {
32. **for**(**int** j = 0; j < n; j ++) // цикл, выполняющийся  n раз
33. cout << ch; // вывод заданного символа
34. cout << endl;
35. }
36. Функции с параметрами по умолчанию.

C++ предоставляет возможность при определении функции присваивать значения некоторым параметрам. При объявлении переменных в основной программе и использовании их в качестве аргументов при вызове подпрограммы имеющие параметры по умолчанию, будут использованы значения, присвоенные не параметрам, а те, которые аргументы получили в основной программе. Если подпрограмма вызывается не используя ни одного параметра (при вызове будут пустые скобки), или хотя бы один параметр будет не задействован, то эти параметры будут действительными для основной программы и будут работать как параметры по умолчанию.

В общем целом, функции с параметрами по умолчанию предоставляют возможность создавать такие функции, в которых мы можем не задействовывать все параметры. В том случае, если мы используем параметры, то значения параметров остаются равными их значениями, полученными в основной программе.

Есть несколько случаев работы с параметрами по умолчанию:

1. **Стандартное использование параметров по умолчанию:**
2. // default.cpp
3. // options;
5. #include <iostream>
6. **using** **namespace** std;
8. **void** func(**int** a = 10, **int** b = 12)
9. {
10. cout « a « endl;
11. cout « b « endl;
12. }

15. ///////////////////////////////////

18. **int** main()
19. {
20. **int** numa;
21. **int** numb;
22. func();
23. cout « endl;
24. **return** 0;
25. }
27. //////////////////////////////////  выведется 10 и 12

2)  **Без использования параметров**

1. // default.cpp
2. // options;
4. #include <iostream>
5. **using** **namespace** std;
7. **void** func(**int** a = 10, **int** b = 12)
8. {
9. cout « a « endl;
10. cout « b « endl;
11. }

14. ///////////////////////////////////

17. **int** main()
18. {
19. **int** numa = 4;
20. **int** numb = 5;
21. func(numa, numb);
22. cout « endl;
23. **return** 0;
24. }
26. ////////////////////////////////// выведется 5 и 4

3 . **Описание функции с параметрами по умолчанию после главной функции:**

1. // default.cpp
2. // options;
4. #include <iostream>
5. **using** **namespace** std;
7. **void** func(**int**, **int**);

10. ///////////////////////////////////

13. **int** main()
14. {
15. func();
16. cout « endl;
17. **return** 0;
18. }
20. //////////////////////////////////
22. **void** func(**int** a = 10, **int** b = 12)
23. {
24. cout « a « endl;
25. cout « b « endl;
26. }

недопустимая форма записи, параметры по умолчанию должны стоять в начале перед функцией main   
или в прототипе и иметь **дальнейшее описание** или иметь **описание** перед функией main

также недопустима такая запись:

1. // default.cpp
2. // options;
4. #include <iostream>
5. **using** **namespace** std;
7. **void** func(**int** a = 10, **int** b = 12);

10. ///////////////////////////////////

13. **int** main()
14. {
15. func();
16. cout « endl;
17. **return** 0;
18. }
20. //////////////////////////////////
22. **void** func(**int** a = 10, **int** b = 12)
23. {
24. cout « a « endl;
25. cout « b « endl;
26. }

Параметры по умолчанию должны быть объявлены один раз и перед главной функцей в которой происходит вызов

1. Текстовые файлы. Отличие от Delphi Pascal.

**Текстовыми** называются файлы, состоящие из любых символов. Они организуются по строкам, каждая из которых заканчивается символом «конца строки». Конец самого файла обозначается символом «конца файла». При записи информации в текстовый файл, просмотреть который можно с помощью любого текстового редактора, все данные преобразуются к символьному типу и хранятся в символьном виде.

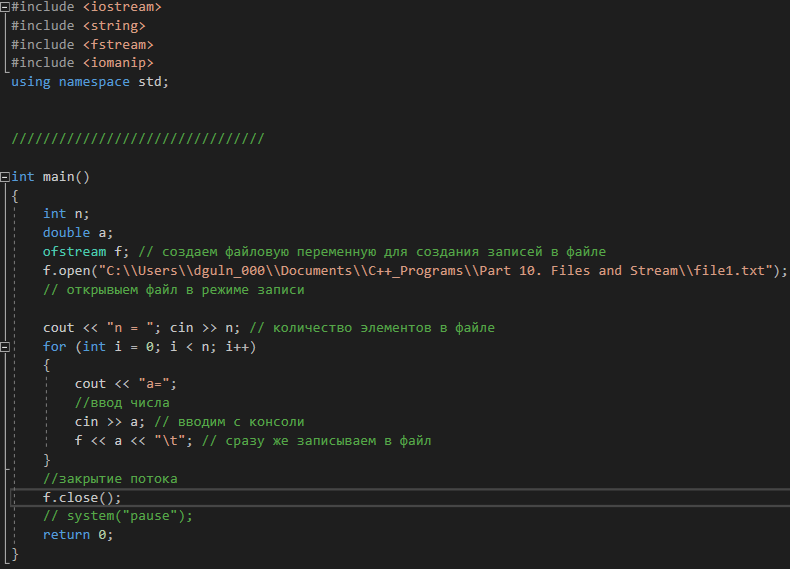
На самом деле для компьютеров нет различия между бинарными файлами и текстовыми файлами, это различие существует только из-за нашего восприятия мира. Вы легко можете прочитать самый простейший текстовый файл при бинарном подходе, при этом вам текст будет точно так же понятен, как он понятен и сам по себе, в блокноте. **Текстовые файлы** — **это только-лишь частный случай бинарных файлов.**

Для работы с файлами используются специальные типы данных, называемые потоками. Поток **ifstream**служит для работы с файлами в режиме чтения, а **ofstream** в режиме записи. Для работы с файлами в режиме как записи, так и чтения служит поток **fstream**.

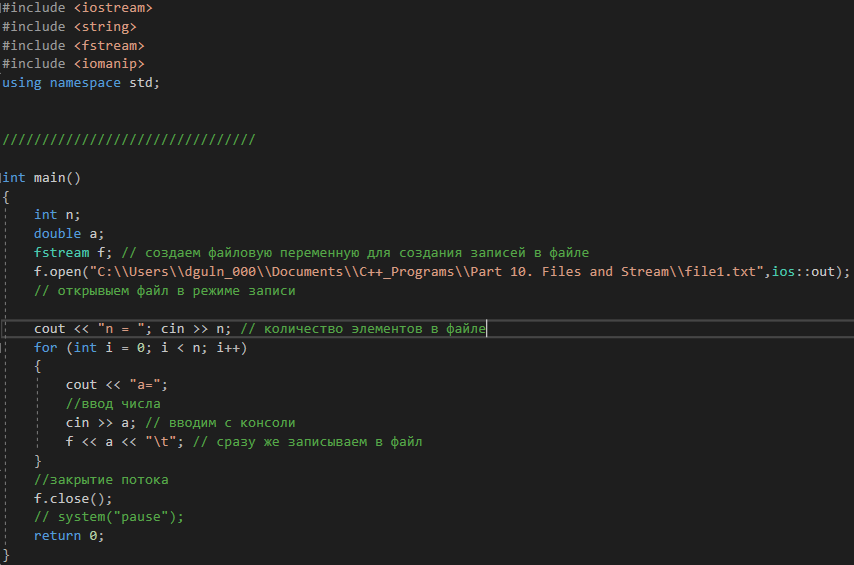
В программах на C++ при работе с текстовыми файлами необходимо подключать *библиотеки* **iostream** и **fstream**.

Также можно создать переменную поточного типа **fstream**, и выбрать для файловой переменной нужный режим работы. Далее будет продемонстрирована работа с файлом с помощью потока **ofstream** и с помощью режима **ios::out** (объявление через **fstream**)

Работа с файлом с помощью потока **ofstream:**



Работа с файлом с помощью общего потока **fstream:**

****

**f**.**open**(**«file»**, **mode**); - что значит эта строка

**«file» –** путь к файлу, как указано в выше описанных программах (используем «\\» для описания директорий)

**mode –** режим.

Список всех доступных режимов:

* *ios::in* — открыть файл в режиме чтения данных; режим является режимом по умолчанию для потоков **ifstream**;
* *ios::out* — открыть файл в режиме записи данных (при этом информация о существующем файле уничтожается); режим является режимом по умолчанию для потоков **ofstream**;
* *ios::app* — открыть файл в режиме записи данных в конец файла;
* *ios::ate* — передвинуться в конец уже открытого файла;
* *ios::trunc* — очистить файл, это же происходит в режиме ios::out;
* *ios::nocreate* — не выполнять операцию открытия файла, если он не существует;
* *ios::noreplace* — не открывать существующий файл.
* *ios::binary –* открытие потока файла в двоичном режиме

Итак, отличие текстовых файлов Delphi от C++ в самом представлении файла.

Если в Delphi текстовый файл имеет блочный ввод и определённый тип(textfile), то в **C++** любые данные добавленные в файл конвертируются в **текстовую информацию** (за исключением режима: *ios::binary*). Если создать программным образом файл в С++ , занести в него данные типа **int**, то при чтении этого файла пользователем этот файл будет иметь **текстовое представление**. Однако в Delphi при занесении информации в текстовый файл мы имеем право записывать/добавлять в него информацию только символьного типа.

Простыми словами все файлы в C++ являются бинарными, но в нашем представлении мы их разделяем на видимые нами и видимые компьютером.

Воспринимаемые нами соответственно текстовые, а видимые компьютером – бинарные (при должном уровне подготовке человек может понимать и бинарные файлы)

1. Двоичные файлы. Отличие от Delphi Pascal.

Двоичный файл отличается от текстового тем, что данные в нем представлены во внутренней форме (текстовые файлы являются альтернативным представлением двоичного файла). А поскольку при внутреннем представлении используется двоичная система счисления, то «в честь ее» файлы и называются двоичными. Разделители компонентов в файле отсутствуют. При записи информации в двоичный файл символы и числа записываются в виде последовательности байт.

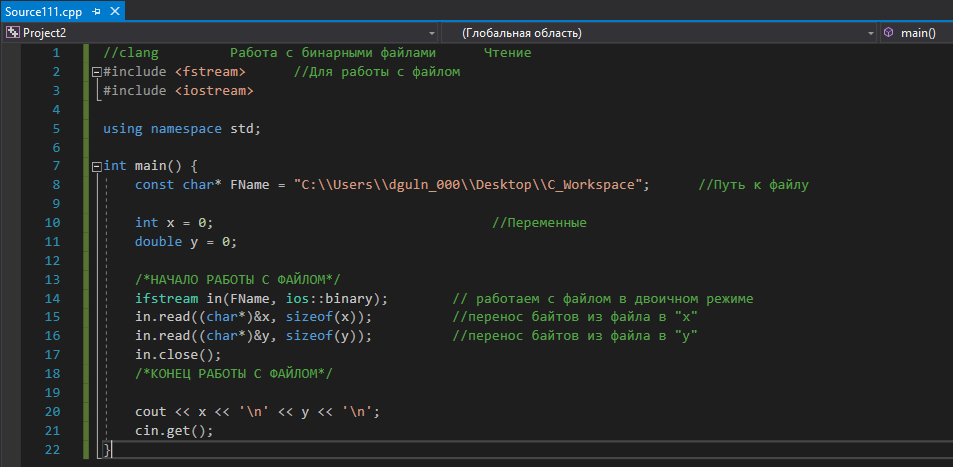
Когда мы говорим о бинарных данных, тип **char** выступает в роли типа **byte**

В С++ нету типа **byte** , а вместо типа **byte** используют тип **char**. Согласно правилам языка С++, тип **char** оказывается наименьшей единицей информации. Из-за этого восприятие нами ситуации в целом немного искажается, ведь мы видим **char**\*, а на самом деле это как **byte**\*

Чтобы мы могли записать какое-нибудь значение в бинарном представлении, нам нужно для начала вывести это бинарное представление, а чтобы записалось правильное количество байт, нужно явно указывать это количество. Это выглядит приблизительно следующим образом:

(char\*)&x Так мы делаем строку байтов для того, чтобы отдать потоку, открытому в двоичном режиме

sizeof(x) Так мы ограничиваем число уходящих в поток байтов нужным числом



Отличие бинарных файлов Delphi от С++ заключается в том, что в Delphi эти файлы называются типизированными или не типизированными, имеют четкие границы (в компьютерном представлении являются двоичными, для пользователя типизированными).

В Delphi такой файл принимает информацию блоками и может получать определённы

й тип данных. C предоставляет гибкость работы и даёт возможность выделять нужную память для разнотипных данных. Работу с бинарным файлом в C++ можно назвать работой на нижнем уровне, а в Delphi на высшем.

Также можно сказать, что в Delphi вообще нет бинарного представления файлов **для программиста**, в Delphi файл строго содержит тип, а в С++ нет таких рамок. Проще говоря С и С++ предоставляет возможность работы на бинарном уровне, а Delphi нет.

1. Определение класса, компоненты класса. Ограничение доступа.

**Класс** – общая форма, еще не существующих объектов, описывающая методы и свойства.

В класс входят **информационные поля**(переменные различных типов) и **методы** (они же являются функциями). В классе можно *объявить* функции (создавать прототипы) или *описать* функцию целиком.

Все свойства и методы классов имеют права доступа. Для того, чтобы разрешить доступ к данным класса извне, используют модификатор **public**. Закрытые данные класса размещают после модификатора доступа **private**. Если отсутствует модификатор **public**, то все функции и переменные, по умолчанию является закрытыми. Компоненты класса, объявленные в секции **protected**, называются защищенными. Они доступны компонентным функциям не только данного класса, но и его потомков.

1. Инициализация полей при отсутствии конструктора.

Пример:

1. **class** Item
2. {
3. **private**:
4. **int** count;
5. **float** price;
6. **public**:
7. **void** Init(**int** c, **float** p)
8. {
9. count = c;
10. price = p;
11. };
12. };
13. Конструкторы. Инициализация полей при наличии конструктора.

**Конструктор** – это метод класса, выполняющийся автоматически в момент создания объекта. Имя конструктора в точности совпадает с именем класса. Если необходимо, то можно сделать несколько конструкторов (то есть возможность создания перегруженного конструктора класса).

Возможные способы применения конструктора:

Допустим существует класс **Triangle,** тогда его конструктор может выглядеть следующими образами:

**1)**

1. - Triangle(): a(0), b(0)  {}

// в этом способе создается **инициализирующий список**. Возможно объявление непосредственно внутри(тела) конструктора. название: ***конструктор без аргументов() – пустой*.** Нижнее описание конструктора равносильно **инициализирующему списку**.

1. Triangle()
2. {
3. a = 0;
4. b = 0;
5. }

////////////////////////////////////////////////////////////////////////////////////////////////////

**2)**

1. Triangle(**int** num1, **int** num2): a(num1),b(num2)  {}

// также **инициализирующий список** или объявление внутри конструктора. название: ***конструктор с аргументами - два аргумента*.** Нижнее описание конструктора также равносильно **инициализирующему списку**.

1. Triangle(**int** num1, **int** num2)
2. {
3. a = num1;
4. b = num2;
5. }

////////////////////////////////////////////////////////////////////////////////////////////////////

**3)**

1. Triangle(**int** num1 = 15, **int** num2 = 14) : a(num1), b(num2)

// конструктор, в котором по умолчанию могут быть присвоены значения аргументам и передаваться полям объекта название: ***конструктор по умолчанию***

// также внутри этого конструктора возможна инициализация полей объекта, то есть поля уже проинициализированы по умолчанию, но мы их инициализируем заново

////////////////////////////////////////////////////////////////////////////////////////////////////

При объявлении объекта происходит вызов конструктора класса объекта, в этот момент мы можем проинициализировать поля таким образом:

1. Triangle t\_object(20,30);

Тело Класса **Triangle**:

1. **class** Triangle
2. {
3. **private**:
4. **int** a, b;
5. **public**:
6. Triangle(…):{…}
7. };
8. Деструкторы.

Деструктор – это метод класса, автоматически вызываемый при уничтожении объекта. Деструкторы не имеют аргументов. Основная задача – освобождение памяти, выделенной конструктором при создании объекта. Имя деструктора в точности совпадает с именем класса, но с приставкой ~. В классе может быть только один деструктор.

Пример:

1. **class** someClass
2. {
3. **private**:
4. **int** m1, m2, m3;
5. **public**:
6. ~someClass();
7. };
8. someClass::~someClass()
9. {...}
10. Инициализация полей объектов при наличии и отсутствии конструктора.

Пример без конструктора:

1. **class** Item
2. {
3. **private**:
4. **int** count;
5. **float** price;
6. **public**:
7. **void** Init(**int** c, **float** p)
8. {
9. count = c;
10. price = p;
11. }
12. };

Пример с конструктором:

1. **class** Triangle
2. {
3. **private**:
4. **int** a,b;
5. **public**:
6. Triangle(**int** n1, **int** n2):a(n1), b(n2)
7. };
8. Простое и множественное наследование классов.

Наследованием называют конструирование новых более сложных производных классов (классов-потомков) из уже имеющихся базовых классов (классов-родителей) посредством добавления полей и методов

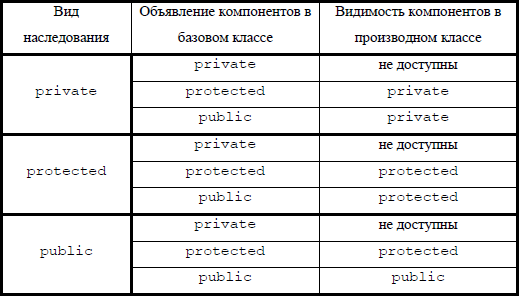
Множественное наследование позволяет одному производному классу иметь несколько родителей.

Пример простого наследования:

1. **class** Figure
2. {/\*..\*/}
4. **class** MyFigure : **public** Figure
5. {/\*...\*/}

Пример множественного наследования:

1. **class** A { /\*…\*/ };
3. **class** B { /\*…\*/ };
5. **class** C: **public** A, **private** B { /\*…\*/ };
6. Наследование. Ограничение доступа при наследовании.



1. Конструкторы и деструкторы производных классов.

Как базовый класс, так и производный класс могут иметь конструкторы. Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Конструктор базового класса выполняется перед конструктором производного класса. Деструктор производного класса вызывается перед деструктором базового класса. Деструкторы также не наследуются.

1. **class** A
2. {
3. **int** x;
4. **public**:
5. A(**int** ax) : x(ax) {}  // конструктор базового класса
6. };
7. **class** B : **public** A
8. {
9. **int** y;
10. **public**:
11. B(**int** ax,**int** ay) : A(ax),y(ay) {}  // конструктор производного класса
12. };

Если вы хотите использовать конструктор производного класса от уже производного класса (тройное наследование), то модель и программы будут выглядеть так:

базовый класс(класс родитель) -> производный класс1 (класс потомок и класс родитель) -> производный класс2 (класс потомок производного класса1)

1. #include <iostream>
2. **using** **namespace** std;
3. **class** A
4. {
5. **public**: **int** a;
6. **A(int v):a(v){}**
7. **void** printa(){ cout<<a<<endl; }
8. };
9. **class** B: **public** A
10. {
11. **public**: **int** b;
12. **B(int va,int vb):A(va),b(vb) {}**
13. **void** printb(**void**) { cout<<a<<" "<<b<<endl; }
14. };
15. **class** C: **public** B
16. {
17. **public**: **int** c;
18. **C(int va,int vb,int vc):B(va,vb),c(vc) {}**
19. **void** printc(**void**)
20. { cout<<a<<" "<<b<<" "<<c<<endl; }
21. };
22. **void** main()
23. {
24. A aa(10); // вызывается конструктор класса А
25. B bb(10,100); //вызывается конструктор класса А, а затем – В
26. C cc(10,100,1000); // вызываются конструкторы классов А, В и С
27. aa.printa();
28. bb.printb();
29. cc.printc();
30. system("pause");
31. }

При наследовании классов, в которых есть конструкторы, нужно учитывать, что если вы создаёте конструктор производного класса и вызываете в нем конструктор базового класса, сперва вызовется конструктор базового класса, а потом уже производного

1. Композиция.

Композицию используют для включения одних объектов в другие в тех случаях, когда нецелесообразно или невозможно с той же целью применить наследование. Композицией называют такое отношение между классами, при котором объекты одного являются неотъемлемой частью другого.

1. Наполнение.

Наполнение используют для включения одних объектов в другие в тех случаях, когда нецелесообразно или невозможно с той же целью применить наследование. Наполнением называют такое отношение классов, при котором количество объектов некоторого класса, включаемых в другой класс, не ограничено и может меняться от нуля до достаточно больших значений. При использовании **наполнения** зачастую создаются динамические структуры данных при помощи объектов.

1. **class** Symbol
2. {...};
3. **class** Record
4. {
5. **private**:
6. Symbol arrayOfSymbols[15];
8. **public**:
9. **void** printArrayOfSymbols();
10. Record();
11. };
12. Полиморфное наследование. Простой полиморфизм.

В языке С++ предусмотрен механизм **полиморфизма**, обеспечивающий возможность определения разных описаний некоторого единого по названию метода для классов различных уровней иерархии. При этом различают *простой полиморфизм*, базирующийся на механизме раннего связывания, и *сложный полиморфизм*, использующий механизм позднего связывания.

Простой (можно использовать термин "статический") полиморфизм поддерживается языком С++ на этапе компиляции и реализуется с помощью *механизма переопределения (перегрузки) функций*. Поэтому такие полиморфные функции называются в С++ переопределяемыми. В соответствии с общими правилами переопределения функций они должны отличаться *сигнатурой*, т. е. количеством, типом и порядком следования передаваемых параметров.

Пример:

1. // Poli\_morfo.cpp
2. //
4. #include <iostream>
5. #include <string>
6. **using** **namespace** std;
8. **class** ClassOne
9. {
10. **protected**:
11. **int** num;
12. **public**:
13. ClassOne()
14. {
15. num = 0;
16. };
17. **void** func() // инкрементация поля класса num, внутри функции func()
18. {
20. ++num;
21. cout << num << '\n';
22. };
23. ~ClassOne()
24. {
25. cout << "\n Destructor activated " << endl;
26. }
27. };
29. **class** ClassTwo : **public** ClassOne
30. {
31. **public**:
32. **void** func()  // декрементация поля класса num внутри функции той жеfunc(), (переопределение функции) - один из видов перегрузки
33. {
34. --num; cout << num << '\n';
35. };
36. ~ClassTwo()
37. {
38. cout << "\n Destructor activated " << endl;
39. }
40. };
42. **int** main()
43. {
44. ClassOne \* One = **new** ClassOne;
45. ClassTwo \* Two = **new** ClassTwo;
46. One->func();
47. Two->func();
48. **delete** One;
49. **delete** Two;
50. **return** 0;
51. }

Однако использование переопределенных методов не всегда безопасно. Известны три случая, когда при их применении возникают ошибки, связанные с некорректным определением типа объекта на этапе компиляции, а, следовательно, и требуемого аспекта вызываемого метода.

1. Полиморфное наследование. Сложный полиморфизм.

Полиморфизм – это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Выполнение каждого конкретного действия будет определяться типом данных.

**Существует 2 типа полиморфизма (условно 2**, по сути одно и тоже, только по-разному интерпретируется)**:**

1. Создание указателя на базовый класс и использование виртуальной функции в базовом классе
2. Создание функции, в которой есть параметр, являющийся указателем на базовый класс, также используется виртуальная функция в базовом классе

Пример **сложного полиморфизма:**

В программе в базовом классе объявляется виртуальная функция **show()** и наследники базового класса будут воспринимать эту функции как невидимую(виртуальную). При вызове одноимённой функции базового и производного класса **show()**, вызовется функция наследника, так как функция базового класса является виртуальной:

1. // virt.cpp
2. // Доступ к виртуальным функциям через указатели
3. #include <iostream>
4. **using** **namespace** std;
5. ///////////////////////////////////////////////
6. **class** Base // Базовый класс
7. {
8. **public**:
9. **virtual** **void** show() // Виртуальная функция
10. { cout << "Base\n"; }
11. };
12. //////////////////////////////////////////////
13. **class** Derv1 : **public** Base // Производный класс 1
14. {
15. **public**:
16. **void** show()
17. { cout << "Derv1\n"; }
18. };
19. //////////////////////////////////////////////
20. **class** Derv2 : **public** Base // Производный класс 2
21. {
22. **public**:
23. **void** show()
24. { cout << "Derv2\n"; }
25. };
26. //////////////////////////////////////////////
27. **int** main()
28. {
29. Derv1 dv1; // Объект производного класса 1
30. Derv2 dv2; // Объект производного класса 2
31. Base\* ptr; // Указатель на базовый класс
32. ptr = &dv1; // Адрес dv1 занести в указатель
33. ptr->show(); // Выполнить show()
34. ptr = &dv2; // Адрес dv2 занести в указатель
35. ptr->show(); // Выполнить show()
36. **return** 0;
37. }
38. Статические компоненты классов.

Когда член класса объявляется как статический, то тем самым компилятору дается указание, что должна существовать только одна копия этого члена, сколько бы объектов этого класса не создавалось. Статический член используют совместно со всеми объектами данного класса. Статическое поле существует даже при отсутствии объектов, а их инициализация в определении класса недопустима.

Пример:

1. **class** counter
2. {
3. **static** **int** count;
4. **public**:
5. **void** setcount(**int** i)
6. {
7. count = i;
8. }
9. **void** showcount()
10. {
11. cout << count << '';
12. }
13. **void** incCount()
14. {
15. count++;
16. }
17. };
19. **int** counter::count = 0;
21. **int** main()
22. {
23. counter a,b;
24. a.showcount();  //выводит 0
25. b.showcount();  //выводит 0
26. a.setcount(10); //установка значения count = 10
27. b.showcount();  //выводит 10
28. b.incCount();   //увел. значение на 1
29. a.showcount();  //выводит 11
30. **return** 0;
31. }
32. Особенности работы с динамическими объектами.

Продолжительность жизни динамических объектов не зависит от того, где они созданы. Динамические объекты существуют, пока не будут удалены явным образом. Динамические объекты размещаются в динамической памяти.

Создание динамического объекта осуществляется при помощи оператора new:

Пример:

Object \* ptr = new Object;

После завершения используемых динамических объектов следует освободить их память с помощью оператора delete:

Пример:

object \* p1 = new object(12);

Delete p1;

Объекты следует удалять только один раз, иначе это может привести к ошибкам в программе. Если оператор delete применен к одному из указателей, то память объекта освобождается, и по второму указателю этот объект мы использовать уже не сможем.

1. Объекты с динамическими полями. Копирующий конструктор.

Например, мы передаем объект в функцию в виде параметра. Функция будет работать не с самим переданным объектом, а с его копией. Допустим в конструкторе класса, при создании объекта, выделяется определенный объем памяти, а деструктор класса эту память освобождает. Указатель копии объекта будет хранить тот же адрес памяти, что и оригинальный объект. И, когда при завершении работы функции и уничтожении копии объекта, сработает деструктор, он обязательно освободит память, которая была занята объектом-оригиналом. В придачу, еще и при завершении работы программы, деструктор сработает повторно и попытается еще раз освободить этот объем памяти, что неизбежно приведет к ошибкам программы. Та же участь постигнет и память, выделенную для указателя объекта, если будет удаляться копия возвращаемого функцией объекта, и копия при инициализации объекта класса другим объектом.

Конструктор копирования нужен нам для того, чтобы создавать «реальные» копии объектов класса, а не копию объекта. Иногда это принципиально важно. Такую «реальную» копию объекта надо создавать в нескольких случаях:

1) Когда мы передаем объект в какую-либо функцию в виде параметра.

2) Когда какая-либо функция должна вернуть объект класса в результате своей работы.

3) Когда мы в главной функции один объект класса инициализируем другим объектом класса.

1. Дружественные функции, методы и классы.

Бывают случаи, когда функции/методы/классы, не являющиеся компонентными, должны иметь возможность обращаться к внутренним компонентам класса. Эти функции/методы/классы получили название дружественных.

Дружественная функция/класс — это функция/класс, которые имеют доступ к закрытым членам другого класса, как если бы они сами были членами этого класса. Это позволяет функции/классу работать в тесном контакте с другим классом, не заставляя другого класса делать открытыми свои закрытые члены.

Функция не может стать другом класса «без его согласия». Для получения «прав друга» функция/метод должна быть описана в теле класса со спецификатором friend.

Пример дружественной функции:

1. **class** Anything
2. {
3. **private**:
4. **int** m\_value;
5. **public**:
6. Anything() { m\_value = 0; }
7. **void** add(**int** value) { m\_value += value; }
9. // Делаем функцию reset() дружественной классу Anything
10. **friend** **void** reset(Anything &anything);
11. };
13. // reset() теперь является другом класса Anything
14. **void** reset(Anything &anything)
15. {
16. // И мы имеем доступ к закрытым членам объектов класса Anything
17. anything.m\_value = 0;
18. }
20. **int** main()
21. {
22. Anything one;
23. one.add(4); // добавляем 4 к m\_value
24. reset(one); // сбрасываем m\_value к 0
26. **return** 0;
27. }
28. Переопределение операций.

Переопределяемые операции реализуются как особый вид функции со специальным именем operator@, где @ – символ переопределяемой операции. Такие функции обычно называются функциями-операторами. Функция-оператор может быть определена как компонент класса или как внешняя независимая от класса (свободная) функция.

При переопределении операций следует помнить, что  
• нельзя переопределять операции \*, sizeof, ?:, #, ##, ::, class::;  
• операции =, [], () можно переопределять только в составе класса;  
• переопределенная операция = не наследуется в производных классах;  
• нельзя изменять приоритет и ассоциативность операции (порядок выполнения операций одного приоритета).

1. Шаблоны классов.

Сам по себе шаблон не вызывает генерацию компилятором какого-либо кода. Генерация кода не происходит до тех пор, пока функция не будет реально вызвана в ходе выполнения программы. Переменная следующая за словом class, называется аргументом шаблона (Также это тип самого шаблона). Шаблоны классов отличаются от шаблонов функций способом реализации. Для создания шаблонной функции мы вызываем ее с аргументами нужного типа. Классы же реализуются с помощью определения объекта, использующего шаблонный аргумент.

Пример:

1. **template** <**class** Type>
2. **class** Stack
3. {
4. **private**:
5. Type var;
6. **public**:
7. Stack(Type value): var(value)
8. {}
9. Type returnValue()
10. {
11. **return** var;
12. }
13. };
15. **int** main()
16. {
17. Stack<**int**> s1(10);
18. Stack<**float**> s2(12.35);
19. cout << s1.returnValue() << '\n';
20. cout << s2.returnValue() << '\n';
21. **return** 0;
22. }
23. Шаблоны функций.

Сам по себе шаблон не вызывает генерацию компилятором какого-либо кода. Генерация кода не происходит до тех пор, пока функция не будет реально вызвана в ходе выполнения программы. Переменная следующая за словом class, называется аргументом шаблона (Также это тип самого шаблона). Говоря нормальными словами шаблон функции – это трафарет для создания множества функций из одной.

Пример:

1. **template** <**class** T>
2. T abs(T n)
3. {
4. **return** (n < 0) ? -n : n;
5. }
7. **int** main()
8. {
9. **int** int1 = 10;
10. **float** float1 = 12.5;
11. cout << abs(int1) << '\n';
12. cout << abs(float1) << '\n';
13. **return** 0;
14. }
15. Организация библиотеки ввода/вывода С++. Операции извлечения и вставки.

**Операции ввода/вывода** выполняются с помощью классов istream (потоковый ввод) и ostream (потоковый вывод). Третий класс, iostream, является производным от них. Для удобства в библиотеке определены три стандартных объекта-потока:

· **cin** – объект класса istream, соответствующий стандартному вводу. В общем случае он позволяет читать данные с терминала пользователя;

· **cout** – объект класса ostream, соответствующий стандартному выводу. В общем случае он позволяет выводить данные на терминал пользователя;

· **cerr** – объект класса ostream, соответствующий стандартному выводу для ошибок. В этот поток мы направляем сообщения об ошибках программы.

Помимо чтения с терминала и записи на него, библиотека iostream поддерживает чтение и запись в файлы. Для этого предназначены следующие классы:

· **ifstream**, производный от istream, связывает ввод программы с файлом;

· **ofstream**, производный от ostream, связывает вывод программы с файлом;

· **fstream**, производный от iostream, связывает как ввод, так и вывод программы с файлом.

1. Организация контейнеров на классах.

**Контейнерный класс** (или **класс-контейнер**) в C++ — это [**класс**](https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/), предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных). наиболее часто используемым контейнером в программировании является [**массив**](https://ravesli.com/urok-74-massivy-chast-1/). Но большинство программистов используют контейнерные классы-массивы — [**std::array**](https://ravesli.com/urok-94-vvedenie-v-std-array/)или[**std::vector**](https://ravesli.com/urok-95-vvedenie-v-std-vector-vektory/) из-за преимуществ, которые они предоставляют. В отличие от стандартных массивов, контейнерные классы-массивы имеют возможность динамического изменения своего размера, когда элементы добавляются или удаляются. В C++ контейнеры обычно содержат только один тип данных. Как только контейнерный класс работает, вы можете его повторно использовать где-угодно без каких-либо дополнительных действий/усилий по части программирования.

1. Организация контейнеров на шаблонах.

Существует возможность создать такую структуру, которая будет хранить в себе разнотипные данные и иметь неограниченное количество элементов. Контейнеры в шаблонах выглядят примерно следующим образом: это класс или структура, внутри которого создается динамическая структура (связный список например). В свою очередь элементы этого списка могут иметь разнотипные данные, тут и применяется шаблоны. Мы создаем контейнеры и такие структуры данных делаем шаблонными. Мы создаем класс, в котором будут храниться неопределённые данные абсолютно любых типов. Данные будут храниться в качестве списка. Вот и все !!!

**Определение:** Шаблонный контейнер – структура данных, переменные (объекты) которой могут воспринимать различные типы данных в динамической структуре. Компилятор сам определит, какого типа будут элементы динамической структуры.

1. Программирование «под Windows» с использованием библиотеки Qt.

**Qt**  — [кроссплатформенный](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%BE%D1%81%D1%81%D0%BF%D0%BB%D0%B0%D1%82%D1%84%D0%BE%D1%80%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) [фреймворк](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) для разработки программного обеспечения

на языке программирования [C++](https://ru.wikipedia.org/wiki/C%2B%2B). Есть также «привязки» ко многим другим языкам программирования.

Основная философия разработчиков Qt Creator заключается в реализации механизма ООП, с помощью которого реализована иерархия всех классов в библиотеки Qt. Для создания оконных приложений операционной системы по типу «Windows», которая кстати переводится как «окна», используется основной класс виджет.

Библиотека Qt предоставляет разработчику множество уже готовых интерфейсных компонентов, которые в Qt, как и в Linux, принято называть *виджетами*.

В простейшем случае виджеты Qt могут встраиваться в программный код без построения специального класса, объект которого соответствовал бы окну.

В библиотеки Qt в начале дерева иерархии классов располагаются классы **QWidget** и **QObject**, в которых заложены основные методы, сигналы, слоты для работы и взаимодействия с объектами этих классов. Что представляют из себя эти классы? Для начала что такое виджет.

В библиотеке Qt используется терминология виджета (от англ. *widget* — штуковина, приспособление), как элемента GUI (graphic user interface)-интерфейса. Минимальный общий набор свойств таких элементов представлен классом **QWidget**.

**Создания простейшего пользовательского интерфейса, в котором используются основные классы библиотеки Qt.**

Первостепенно в начале программы посредством оператора #include подключаем заголовочный файл модуля, содержащего описание используемых интерфейсных классов Qt.

Аналогично любой, построенной по объектной технологии и событийно управляемой программе приложение Hello минимально должно включать два объекта:

* объект-приложение;
* объект-окна приложения.

Объект-приложение создается как объект класса **QApplication**. Объекты этого класса отвечают за создание и инициализацию главного окна, а также за запуск цикла обработки сообщений от операционной системы.

В качестве окна приложения будем использовать объект класса **QWidget**. Класс **QWidget** – базовый класс всех виджетов. Его объекты обладают свойствами контейнера, который управляет виджетами визуальных компонентов: метками, кнопками и др., размещенными в окне.

1. #include <QWidget>
2. #include <QApplication>
3. #include <QLabel>
4. #include <QPushButton>
5. #include <QHBoxLayout>
7. **int** main(**int** argc,**char** \*argv[])
8. {
9. QApplication app(argc,argv);  // создание объекта-приложения
10. QWidget win;                  // создание объекта управления окном
11. win.setWindowTitle("Hello");  // изменение заголовка окна
13. QLabel \* helloLabel=**new** QLabel("Hello!",&win); // создание метки
14. QPushButton \* exitButton=**new** QPushButton("Close",&win);
15. // создание кнопки
16. QHBoxLayout \* layout = **new** QHBoxLayout(&win); // создание
17. // менеджера компоновки для управления размещением метки и кнопки в окне win
18. layout->addWidget(helloLabel);  // добавление метки к компоновщику
19. layout->addWidget(exitButton);  // добавление кнопки  к компоновщику
20. // связь сигнала нажатия кнопки с закрытием окна win
21. QObject::connect(exitButton,SIGNAL(clicked(**bool**)),
22. &win,SLOT(close()));
23. win.show();    // визуализация окна win
24. app.exec();    // запуск цикла обработки сообщений приложения
25. }

В примере мы подключили несколько библиотек и создали примитивный пользовательский интерфейс.

Перечень библиотек:

**<QWidget>** - создали объект класса **QWidget**, который является главным окном приложения, установили имя главного окна с помощью метода класса - **setWindowTitle(“”);**

**<QApplication>** - создали объект класса **QApplication** - **app**, который является самим приложением, отвечает за все виджеты, хранящиеся в нём, отвечает за создание и уничтожение главного окна и всех виджетов, расположенных на главном окне.

**<QLabel>** - создаем указатель на объект класса **QLabel** - ***helloLabel*** и выделяем на него память, при выделении памяти устанавливаем имя виджета “Hello” и показываем что он располагается на главном окне - &win.(предопределенная операция выделения памяти, включающая имя и расположения виджета). **QLabel ­**– это что-то вроде метки, виджет, в котором хранится символьная информация и появляется на экране соответственно при появлении самого виджета. **QLabel** является наследником класса **QWidget** в библиотеки **Qt**, и имеет доступ ко всем его «публичным методам», «публичным слотам и сигналам».

**<QPushButton>** - создаем указатель на объект класса **QPushButton** - ***exitButton*,** выделяем на него память, устанавливаем его имя и устанавливаем расположение. **QPushButton** – это объект кнопка. Описания события нажатия кнопки будет ниже.

**<QHBoxLayout>** - создание указателя на объект класса **QHBoxLayout** - ***layout*,** который организовывает горизонтальную привязку первого и второго виджета, с помощью метода класса **addWidget** и **addLayout**. Более подробно в «создании менеджера привязки» (Организация интерфейса) - Организация интерфейса с использованием виджетов Qt.

Особого внимания заслуживает оператор (макрос) **Qt connect**, который связывает *сигнал* нажатия кнопки ***exitButton*** – clicked(bool) с его обработчиком:

QObject::connect(exitButton,SIGNAL(clicked(bool)),

&win,SLOT(close()));

Таким обработчиком – ***слотом*** окна **win** – является метод закрытия окна **win** - **close()** прописанный, как метод класса **QWidget** в библиотеке **Qt.** По правилам оконных приложений этот метод обеспечивает не только закрытие окна, но и завершение приложения, если закрываемое окно – последнее.

После визуализации окна выполняется метод **app.exec()**-метод класса **QApplication**, который запускается цикл обработки сообщений, организуя работу приложения.

1. Организация интерфейса с использованием виджетов Qt.

Современная концепция создания графических оконных приложений предлагает следующую возможность:

Разработка собственного класса окна приложения на основе библиотечных классов

Мы понимаем, что в библиотеке классов располагается класс **QWidget**, но нам нужно модифицировать этот класс и создать объект этого класса. Есть 2 способа: модифицировать библиотечный класс **QWidget**, что крайне нежелательно, ведь он является родителем многих других классов и модификация этого класса может повлиять на работу наследников, и вообще нарушить работу библиотеки.

2 способ, являющийся самым рациональным: создать наследник класса **QWidget** – **Widget**, который унаследует все методы класса и сможет включать новые.

**Листинг будет размещаться в заголовочном файле главного окна win.h.**

**Класс наследник выглядит примерно так:**

1. #ifndef win\_h
2. #define win\_h
3. #include <QtGui>
5. **class** Win: **public** QWidget
6. {
7. QLabel \*helloLabel;
8. QPushButton \*exitButton;
9. **public**:
10. Win(QWidget \*parent = 0);
11. };
12. #endif

Теперь мы имеем модифицированный класс **Win,** в котором объявлены два объекта

В большинстве случаев разработчики создают интерфейс главного окна в конструкторе класса, так как это наиболее логичный метод появления виджетов на главном окне. При создании объекта класса **Widget** будет вызываться конструктор, который создаст и охарактеризует все указатели на объект класса **Widget.**

Здесь пришло время рассказать о цепях владения, которые реализованы в системе классов Qt для решения проблемы уничтожения объектов для предотвращения случайных утечек памяти. Следует вспомнить, что объекты созданные динамически, т.е. с помощью оператора **new**, размещаются в специальной области памяти, называемой *кучей (heap)* и которые живут в куче до тех пор пока не будут явно уничтожены оператором **delete**. Если программист не отслеживает уничтожение объектов, которые стали ненужными и не вызывает для их уничтожения оператор **delete**, то это становится причиной утечки памяти в приложении, которая является серьезной проблемой для ряда языков программирования в группу которых входит язык C++.

Существует несколько известных схем автоматического слежения за уничтожением динамически созданных объектов. Один из них заключается в использовании умных указателей. Другой способ заключается в создании цепей владения, о котором сейчас пойдёт речь. Третьим способом является создание подсистемы уборки мусора, которая должна отлеживать ненужные объекты и уничтожать их. Последний способ, традиционный для ядер многих современных языков, в C++ практически не используется. Гораздо более популярными в традициях языка C++ являются первые два способа.

Итак, цепи владения реализуют следующую простую идею. Создается некоторый объект, за уничтожением которого мы обязуемся следить. Проще всего создать такой объект статическим определением и тогда он уничтожится автоматически, когда исполнение программы достигнет конца операторного блока { … } в котором он был определен. Далее, при динамическом создании других объектов будем назначать для них объекты владельцев. В обязанности владельцев будет входить уничтожении владеемых объектов в теле собственного деструктора. Напомним, что деструктор — это особый метод, который вызывается при уничтожении объекта. Таким образом, можно построить такую цепь владения относительно первого объекта, все элементы которой будут автоматически уничтожены при уничтожении первого объекта. При организации такой схемы следует лишь правильно учитывать время жизни объекта, который назначается владельцем другого объекта, чтобы объекты не были уничтожены преждевременно.

Итак, после того, как мы поняли, как работает механизм цепи владения и больше можем не беспокоиться о ручном освобождении памяти, пришло время показать

**Файл win.cpp, который содержит описание конструктора класса Win:**

1. #include "win.h"
2. Win::Win(QWidget \*parent):QWidget(parent)
3. {
4. setWindowTitle("Hello");
5. helloLabel=**new** QLabel("Hello!",**this**);
6. exitButton=**new** QPushButton("Exit",**this**);
7. QHBoxLayout \*layout = **new** QHBoxLayout(**this**); // создание элемента
8. // компоновки для управления размещением метки и кнопки в окне win
9. layout->addWidget(helloLabel);  // добавление метки к компоновщику
10. layout->addWidget(exitButton);  // добавление кнопки к компоновщику
11. // связь сигнала нажатия кнопки и слота закрытия окна
12. connect(exitButton,SIGNAL(clicked(**bool**)),
13. **this**,SLOT(close()));
14. }

## Файл main.cpp содержит основную программу:

1. #include "win.h"
2. **int** main(**int** argc, **char** \*argv[])
3. {
4. QApplication app(argc, argv);
5. Win win(0);
6. win.show();
7. **return** app.exec();
8. }

## **Альтернативный ответ на 45 вопрос**

## **Способы размещения виджетов**

В эволюции идей размещения элементов управления на оконных формах можно выделить три подхода.

1. **Координатное размещение.** Это самый простой и самый неудачный способ размещения элементов в форме. Простота его заключается в том, что он совершенно понятен любому человеку без какой-либо подготовки. С каждым элементом управления можно связать прямоугольную область в которой он размещается. Более того, большинство элементов имеют прямоугольные границы, поэтому можно не усложнять систему определений. Размещая один элемент поверх другого, координатным способом, надо лишь указать, например, координаты верхнего левого угла размещения элемента и его протяженность — ширину и высоту. В другом варинте, вместо ширины и высоты можно указать координаты нижнего правого угла размещения. *К недостаткам* такого способа размещения объектов следует отнести проблемы возникающие при изменении стилей элементов управления и отсутствие возможностей автоматической адаптации размеров элементов к изменению размеров формы.
2. **Установка привязки границ (установка якорей).** Фактически, этот способ размещения элементов является улучшенным способом координатного размещения. Основное отличие заключается в возможности установки так называемых якорей, которые определяют жесткую привязку границы одного элемента к границе другого элемента. Это позволяет успешно закладывать возможности изменения размеров некоторого вида форм. Например, привязывая нижнюю границу окна редактора к нижней границе формы, растягивая форму вниз, мы растягиваем окно редактора за счет привязки его нижней границы. Способ не является сложным для понимания, однако, не решает вопросы изменения размеров любых видов форм.
3. **Использование менеджеров компоновки.** Менеджерами компоновки называют специального вида объекты, которые действуют как резиновая система ячеек. Объект менеджера компоновки "натягивается" на некоторый базовый виджет (например, на окно формы), а в ячейки компоновщика укладываются виджеты, которые следует разместить на базовом виджете.

В зависимости от правил работы компоновщика и от политики изменения размеров виджетов, при изменении размеров базового виджета, производится автоматический пересчет размеров виджетов, уложенных в ячейки компоновщика. Существуют множество разных моделей компоновщиков в каждой из библиотек, где они используются. В ячейки компоновщиков можно вкладывать не только виджеты, но и другие компоновщики, в результате чего можно придумать сложные растягиваемые структуры. Кроме создания сложных компоновочных структур, можно управлять политиками растяжения виджетов, которые укладываются в ячейки компоновщиков. Таким образом, открываются очень широкие возможности для создания растягиваемых форм разной сложности и поведения. *К недостаткам* такого способа размещения объектов можно отнести лишь некоторую сложность использования компоновщиков, особенно на начальном этапе знакомства с ними..

В нашей программе мы используем менеджер компоновки класса **QHBoxLayout \* *layout*;**

Чтобы понимать работу менеджеров компоновки нужно знать следующее.

1. Цепи владения компоновщиков реализованы так, что владеть компоновщиком может только компоновщик.
2. Каждый элемент, который наследуется от класса **QWidget** знает о классах компоновки и умеет их использовать. Для указания компоновщика, виджеты имеют специальный метод *setLayout()*. Таким образом, чтобы привязать компоновщик к виджету, нужно создать объект компоновщика, а потом передать его в метод виджета *setLayout()*.
3. Особое внимание требует конструктор компоновщиков с сигнатурой *(QWidget \* parent=0)*. По внешнему виду он похож на конструкторы других виджетов выполняющих роли создания цепей владения. В случае компоновщиков это не совсем так (см. пункт 1). С одной стороны, виджет действительно владеет цепями компоновщиков которые он использует и автоматически уничтожает их при своем удалении. Однако, в данном случае, параметр **parent** используется не для того, чтобы компоновщик подключился к цепи владения объекта на который указывает **parent**, а для того, чтобы выполнить *parent->setLayout()*, т.е. установить себя компоновщиком указанного виджета.
4. Когда один менеджер компоновки (первичный) включает в свою ячейку другой менеджер компоновки (вторичный), то вторичный компоновщик создается без указания **parent**, так как у виджета может быть только один менеджер компоновки. Владение вторичным компоновщиком создается автоматически при добавлении вторичного компоновщика в ячейку первичного компоновщика. Таким образом, первичный компоновщик будет владеть вторичным и будет ответственным за его удаление. См. метод компоновщиков *addLayout()*.
5. В ячейки менеджеров компоновки можно установить либо виджет, либо вторичный менеджер компоновки. Для этого существуют методы *addWidget()* и *addLayout()* соответственно. При добавлении вторичного компоновщика, как уже говорилось, он включается в цепь владения первичного компоновщика. При добавлении виджета, он включается в цепь владения виджета, которому принадлежит цепь компоновщиков только в том случае, если для него не был ранее задан владелец. Такое поведение не характерно для старых версий Qt, поэтому еще существуют традиции, которые явно задают цепи владения для виджетов, которые устанавливаются в ячейки менеджеров компоновщики.

Пример. Управление размером окна посредством изменения геометрических свойств корневого виджета (Координатный метод).

1. #include <QtGui>
2. **int** main(**int** argc, **char** \*argv[])
3. {
4. QApplication app(argc, argv); // создаем объект-приложение
5. QWidget window;               // создаем корневой виджет
6. QRect rect=window.geometry(); // читаем размер окна по умолчанию
7. window.setGeometry(20,20,100,100); // устанавливаем размер окна
8. window.resize(300,100);             // меняем ширину и высоту окна
9. window.setWindowTitle("Main Window");// устанавливаем заголовок
10. window.setObjectName("window");      // сохраняем имя объекта
11. window.show();             // визуализируем окно
12. **return** app.exec();         // запускаем цикл обработки сообщений
13. }

В отличие от ручного варианта при компоновке с использованием менеджеров компоновки осуществляется автоматическая перестройка внешнего вида окна в зависимости от его размеров.

В Qt предусмотрены следующие элементы компоновки:

* QVBoxLayout – вертикальный компоновщик – управляет расположением виджетов в окне по вертикали;
* QHBoxLayout – горизонтальный компоновщик – управляет расположением виджетов в окне по горизонтали;
* QGridLayout – табличный компоновщик – управляет расположением виджетов в направляющей двумерной сетке – матрице или таблице.

**Пример.** Автоматическая компоновка виджетов в окне

1. #include <QtGui>
2. **int** main(**int** argc, **char** \*argv[])
3. {
4. QApplication app(argc, argv);
5. QWidget window;
6. window.setWindowTitle("Main Window");
7. window.setObjectName("window");
8. QLineEdit \*edit1=**new** QLineEdit("Edit1",&window);
9. QLineEdit \*edit2=**new** QLineEdit("Edit2",&window);
10. QHBoxLayout \*layout = **new** QHBoxLayout; // выравнивание по
11. // горизонтали
12. layout->setContentsMargins(5,5,5,5);  // внешние поля окна
13. layout->setSpacing(5);                // просвет между виджетами
14. window.setLayout(layout); // связывание layout с виджетом окна
15. // задание порядка следования элементов
16. layout->addWidget(edit1);
17. layout->addWidget(edit2);
18. window.show();
19. **return** app.exec();
20. }
21. Сигналы, слоты и события Qt.

При использовании Qt передача сообщений внутри приложения реализуется *механизмом* ***слотов и сигналов***. Это – наиболее важный механизм Qt, отличающий его от других библиотек интерфейсных элементов C++, например «родной» библиотеки Visual C++ – MFC.

По правилам Qt любой виджет может посылать **сигналы** другим виджетам, сообщая им об изменениях, произошедших с ним в процессе функционирования. Чаще всего причиной формирования **сигнала** бывают действия пользователя. Например, объекты класса **QPushButton** посылают приложению **сигнал** **clicked**(), когда пользователь щелкает мышкой по реализуемой им кнопке. Причинами генерации **сигналов** могут быть и достижения каких-либо значений, срабатывания таймеров, действия операционной системы или других приложений.

Посредством специального оператора Qt **connect** *каждый* ***сигнал*** *может быть подключен к одному или нескольким* ***слотам*** *других виджетов*. Тогда каждый раз при получении **сигнала** в виджетах-получателях будет активизироваться соответствующий обработчик **сигналов** – слот или последовательно несколько **слотов**.

В качестве **слота** может объявляться любой (перегруженный, виртуальный, общий, защищенный, закрытый) метод, который дополнительно объявлен слотом, что позволяют подключать его к сигналу. Такой метод также сохраняет возможность традиционного вызова, не связанного с сигналом.

Точно так же, како**дин сигнал может быть подключен к нескольким слотам, *к одному слоту может быть подключено несколько сигналов*. В последнем случае приложение одинаково реагирует на разные сигналы.**

Соединяемые **сигналы** и **слоты** должны иметь идентичные сигнатуры (т.е. количество и типы входных аргументов). Исключением является случай, когда сигнал имеет большее число аргументов, чем слот. В этом случае "лишние" аргументы просто не передаются в слот. Если типы входных аргументов не совместимы, или **сигнал** или **слот** не определены, Qt выдаст предупреждение *во время выполнения программы*. Точно так же Qt отреагирует, если в сигнатуры **сигналов** или **слотов** в макросе **connect()**включены имена аргументов.

Пример работа макроса **connect,** в котором связываются объект кнопка и объект главное окно.

1. connect(exitButton,SIGNAL(clicked(**bool**)),

**this**,SLOT(close()));

Данное событие можно представить для себя логически таким образом:

Когда будет нажата кнопка  **exitButton** (будет отправлен сигнал о нажатии), то будет закрыто главное окно (вызовется метод-слот который закроет главное окно)