ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Структура дисциплины

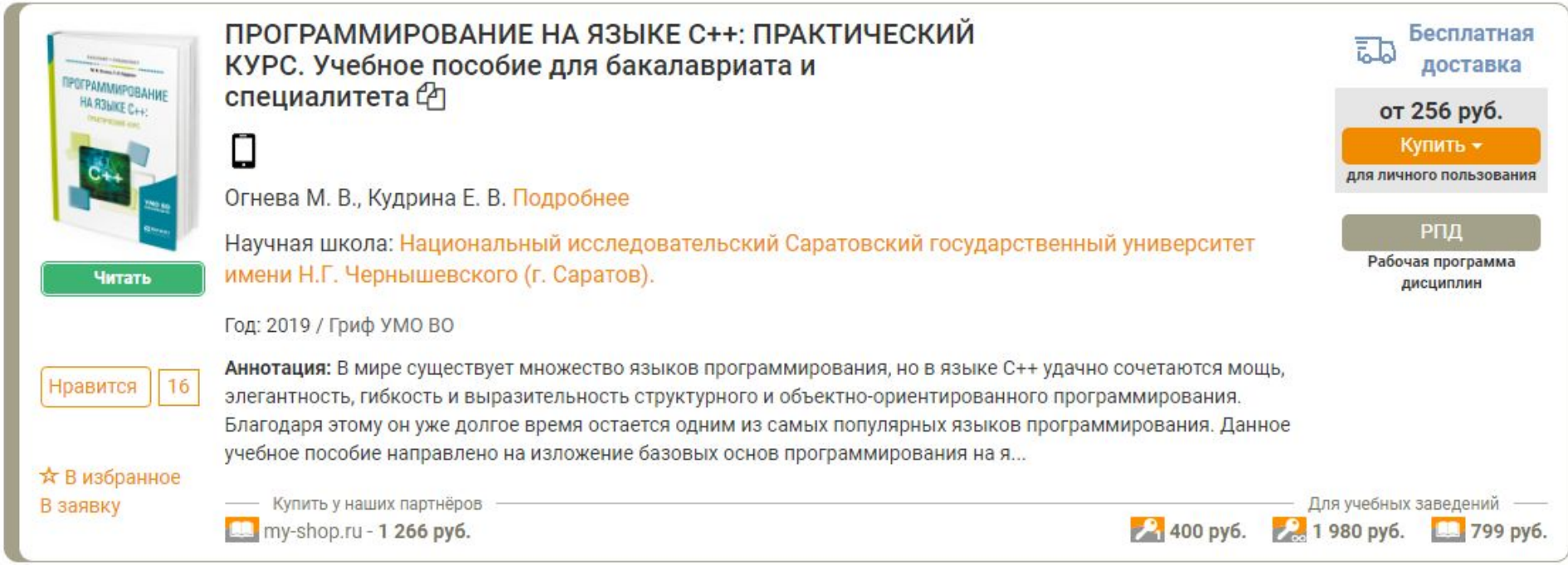
КУРС	СЕМЕСТР	ОБЩАЯ ТРУДОЁМКОСТЬ ЗАЧЕТНЫХ ЕДИНИЦ	ЧАСЫ (ПО НАЛИЧИЮ ВИДОВ ЗАНЯТИЙ)													Вид промежуточного контроля
			ОБЩАЯ ТРУДОЁМКОСТЬ	АУДИТОРНЫЕ ЗАНЯТИЯ						САМОСТОЯТЕЛЬНАЯ РАБОТА						
				ВСЕГО	ЛЕКЦИИ	ЛАБОРАТОРНЫЙ ПРАКТИКУМ	АУДИТОРНЫЙ ПРАКТИКУМ		ДРУГИЕ ВИДЫ ЗАНЯТИЙ	ВСЕГО	КУРСОВОЙ ПРОЕКТ	КУРСОВАЯ РАБОТА	РАСЧЁТНО-ГРАФ. РАБОТА	РЕФЕРАТ	ДРУГИЕ ВИДЫ САМОСТ. РАБОТЫ	
							ПРАКТИЧЕСК ИЕ ЗАНЯТИЯ	СЕМИНАРЫ								
1	2	5	180	68	34	-	34	-	-	112	-	18	-	-	94	Дифф. зачет, КР
2	3	5	180	68	34	-	34	-	-	112	-	18	-	-	94	Дифф. зачет, КР
ИТОГО		10	360	136	68	-	68	-	-	224	-	36	-	-	188	

Раздел 3. Стандартные и пользовательские типы данных в C++. Обработка исключений. Инкапсуляция и статический полиморфизм в C++

Теоретические занятия (лекции) - 8 часов.

Лекция 6. Информационная лекция (2 часа.)

В отличие от Си, в языке C++ существуют операторы размещения динамических переменных: **new**, **delete**. В ходе данной лекции будут рассмотрены операции преобразования типа Си в язык C++. Помимо операций размещения динамических переменных, на лекции будут рассмотрены операции **static_cast, dynamic_cast**, **const_cast, reinterpret_cast**.



Огнева, М. В.
Программирование на языке С++: практический курс : учебное пособие для бакалавриата и специалитета / М. В. Огнева, Е. В. Кудрина. — Москва : Издательство Юрайт, 2019. — 335 с. — (Бакалавр и специалист). — ISBN 978-5-534-05123-0. — Текст : электронный // ЭБС Юрайт [сайт]. с. 251 — URL: https://biblio-online.ru/bcode/438987/p.251 (дата обращения: 25.02.2020).

Работа с динамическим разделением памяти в Си

В Си работать с динамической памятью можно при помощи соответствующих функций распределения памяти (calloc, malloc, free), для чего необходимо подключить библиотеку malloc.h

C++ использует новые методы работы с динамической памятью при помощи операторов new и delete

https://prog-cpp.ru/cpp-newdelete/

МЕТОДЫ РАБОТЫ С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ ПРИ ПОМОЩИ ОПЕРАТОРОВ NEW И DELETE

C++ использует новые методы работы с динамической памятью при помощи операторов `new` и `delete`:

- `new` — для выделения памяти;
- `delete` — для освобождения памяти.

Оператор `new` используется в следующих формах:

- **`new тип;`** — для переменных
- **`new тип[размер];`** — для массивов

Память может быть распределена для одного объекта или для массива любого типа, в том числе типа, определенного пользователем. Результатом выполнения операции `new` будет указатель на отведенную память, или нулевой указатель в случае ошибки.

```
int *ptr_i;
double *ptr_d;
struct person *human;
…
ptr_i = new int;
ptr_d = new double[10];
human = new struct person;
```

Память, отведенная в результате выполнения `new`, будет считаться распределенной до тех пор, пока не будет выполнена операция `delete`.

Освобождение памяти связано с тем, как выделялась память – для одного элемента или для нескольких. В соответствии с этим существует и две формы применения `delete`:

- **`delete указатель;`** — для одного элемента
- **`delete[] указатель;`** — для массивов

Например, для приведенного выше случая, освободить память необходимо следующим образом:

```
delete ptr_i;
delete[] ptr_d;
delete human;
```

Освобождаться с помощью `delete` может только память, выделенная оператором `new`.

Указатель `dan` – базовый адрес динамически распределяемого массива, число элементов которого равно `size`. Операцией `delete` освобождается память, распределенная при помощи `new`.

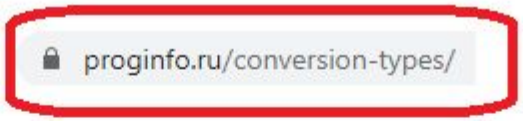
Пример Создание динамического массива

```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {
5     int size;
6     int *dan;
7     system("chcp 1251");
8     system("cls");
9     cout << "Ввести размерность массива : ";
10    cin >> size;
11    dan = new int[size];
12    for (int i = 0; i<size; i++) {
13      cout << "dan[" << i << "] = ";
14      cin >> dan[i];
15    }
16    for (int i = 0; i<size; i++)
17      cout << dan[i] << " ";
18    delete[] dan;
19    cin.get(); cin.get();
20    return 0;
21  }
```

Результат выполнения

```
C:\MyProgram\Program\Debug\Program.exe
Ввести размерность массива : 5
dan[0] = 4
dan[1] = 3
dan[2] = 7
dan[3] = 9
dan[4] = 6
4 3 7 9 6
```

Операции преобразования типа Си в язык C++



Преобразование типов

Преобразование типа в языке C++ в целом аналогично такому преобразованию в других языках. Но есть несколько тонкостей, которые выделяют C++ в этом плане.

https://proginfo.ru/conversion-types/

Как известно, есть 2 вида преобразования типа – неявное и явное. Первый вариант означает, что компилятор при считывании кода изменяет тип самостоятельно. Для примера

```
1 int i=10
2 float j=5;
3 float a;
4
5 A=i+j;
```

Программа сама меняет тип i на float и суммирует 2 числа. Тип меняется только для этой функции.

Второй вариант подразумевает, что преобразование идет с помощью специальной функции. Ниже приведен пример подобного

```
int m=15;

A=float(15);
```

Так же преобразования возможны с помощью функций atoi, atof, atol, из массива символов в числа разных форматов.

```
int a = atoi(const char* str)
long b = atol(const char* str)
double c = atof(const char* str)
```

Для обратного перевода из числа в строку явно заданных функций в стандартных библиотеках нет. Но можно воспользоваться функцией

```
1 sprintf(char* buffer, const char* format [, argument] ... );
```

которая напоминает функцию printf, но печатает не на экран, а в строку buffer, или встроенными функциями компилятора.

Для примера, в компиляторе Borland перевод из int в char* имеет вид char* itoa(int value, char* string, int radix). Параметры этой функции: value – исходное число, string – возвращаемая строка, radix – система счисления (для десятичной системы счисления radix = 10).

Все виды преобразований не слишком надежны и зачастую дают неточности, так что использование их в коде сопряжено с трудностями. Рекомендуется делать проверки результатов преобразований и задавать граничные значения.

Приведем пример программы, преобразующей целое число в строку, строку в вещественное число и вещественное число в строку.

```
#include <iostream.h> // подключаем функцию cout
#include <stdio.h>  // подключаем функцию sprintf

int main(int argc, char* argv[])
{
   int A;
   float B;
   char C[10];

   A = 10;
   sprintf(C, "%d", A);
   cout<<C<<" ";
   B=atof(C);
   cout<<B<<" ";
   sprintf(C, "%f", B);
   cout<<C<<" ";
   return 0;
}
```

ОПЕРАЦИИ

static_cast

dynamic_cast

const_cast

reinterpret_cast

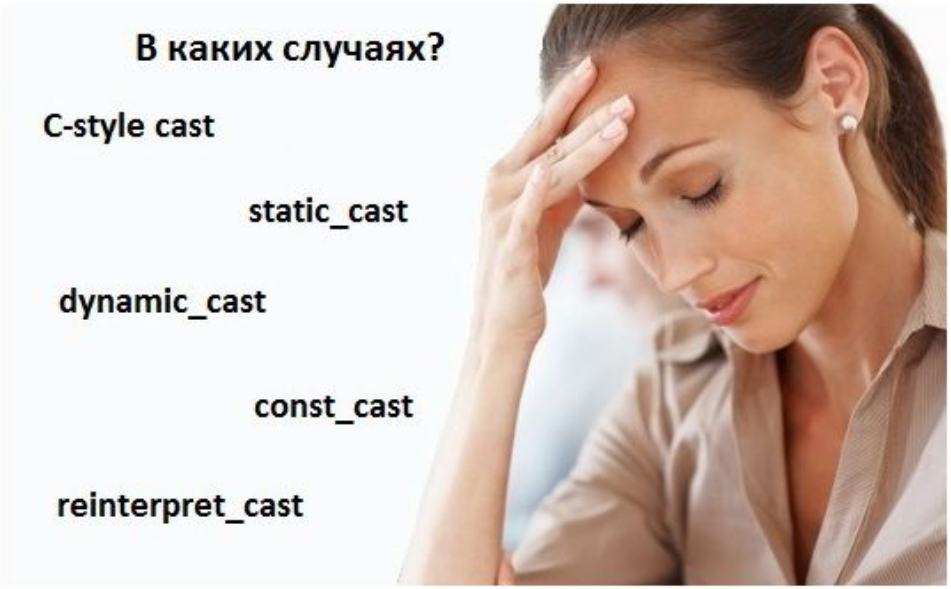
ОПЕРАЦИИ static_cast, dynamic_cast, const_cast, reinterpret_cast

https://habr.com/ru/post/266747/

Еще раз про приведение типов в языке C++ или расстановка всех точек над cast

C++

Tutorial



Этот пост попытка кратко оформить все, что я читал или слышал из разных источников про операторы приведения типов в языке C++. Информация ориентирована в основном на тех, кто изучает C++ относительно недолго и, как мне кажется, должна помочь понять специфику применения данных операторов. Старожилы и гуру C++ возможно помогут дополнить или скорректировать описанную мной картину. Всех интересующихся приглашаю под кат.

ОПЕРАЦИИ static_cast, dynamic_cast, const_cast, reinterpret_cast

https://habr.com/ru/post/266747/

Приведение типов в стиле языка C (C-style cast)

Приведение типов в стиле языка C может привести выражение любого типа к любому другому типу данных (исключение это приведение пользовательских типов по значению, если не определены правила их приведения, а также приведение вещественного типа к указателю или наоборот). К примеру, unsigned int может быть преобразован к указателю на double. Данный метод приведения типов может быть использован в языке C++. Однако, метод приведения типов в стиле языка C не делает проверки типов на совместимость, как это могут сделать static_cast и dynamic_cast на этапе компиляции и на этапе выполнения соответственно. При этом все, что умеют const_cast и reinterpret_cast данный метод приведения типов делать может.

Общий вид приведения:

(new_type)exp

, где *new_type* – новый тип, к которому приводим, а *exp* – выражение, которое приводится к новому типу.

Т.к. данный оператор не имеет зарезервированного ключевого слова (например, static_cast) найти все места приведения типов в тексте программы будет не очень удобно, если это потребуется.

```cpp
#include <iostream>
//Пустые классы только
//для теста приведения
struct AAA{
};
struct BBB{
};
//Наследники BBB
struct BBB_X:BBB{
};
struct BBB_Y:BBB{
};

int main()
{
        //Переменные простых типовы и указатели на переменные простых типов
        int    i = 5;
        double d = 111.222;
        char   c = 'a';
        int*   pi = &i;
        double * pd = &d;
        const int* cpi = &i;
        void*  v = NULL;
        //Объекты классов
        AAA A;
        BBB B;
        BBB_X BX;
        BBB_Y BY;
```

```
        //Указатели на объекты классов
        AAA* pA = &A;
        BBB* pB = &B;
        BBB_X* pBX = &BX;
        BBB_Y* pBY = &BY;
        //Приводим явно double к int
        i = (int)d;
        //и наоборот
        d = (double)i;
        //указатель на int к char
        c = (char)pi;
        //char к указателю на void
        v = (void*)c;
        //указатель на void к указателю на int
        pi = (int*)v;
        //Снимаем константность const int*
        pi = (int *) cpi;
        //Приводим указатель на объект AAA к указателю на объект BBB
        //из разных иерархий
        pA = (AAA*) pB;
        //Приводим указатель на double к double
        d = (double)pd;//Ошибка!!!
        //А если наоборот?
        pd = (double*)d;//Ошибка!!!
        //Перемещение из одной иерархии наследования в другую
        pB = (BBB*)pBX;
        pBY = (BBB_Y*) pB;
        return 0;
}
```

const_cast

Оператор приведения const_cast удаляет или добавляет квалификаторы const и volatile с исходного типа данных (простые типы, пользовательские типы, указатели, ссылки). Например, был const int, а после преобразования стал int или наоборот. Квалификаторы const и volatile называют cv-квалификаторы (cv-qualifiers). Данные квалификаторы указываются перед именами типов. Как ни трудно догадаться квалификатор const задает константность, т.е. защищает переменную от изменения. Квалификатор volatile говорит о том, что значение переменной может меняться без явного выполнения присваивания. Это обеспечивает защиту от оптимизации компилятором операций с данной переменной.

Общий вид приведения:

const_cast<new_type>(exp)

```cpp
#include <iostream>
//Снятие константности
void test_func_X(const int* in1, const int& in2)
{
        int *p;
        //Сняли константность и записали 33
        p = const_cast<int*>(in1);
        *p = 33;
        //Сняли константность и записали 55
        const_cast<int&>(in2) = 55;
}
//Добавление константности
void test_func_Y(int* in1, int& in2)
{
        const int *p;
        //Добавили константность
        //и пытаемся записать 33
        p = const_cast<const int*>(in1);
        *p = 33;//Ошибка !!!
        //Добавили константность константность
        //и пытаемся записалть 33
        const_cast<const int&>(in2) = 55;//Ошибка!!!
}
//Снятие volatile
```

```
//Снятие volatile
void test_func_Z(volatile int* in1, volatile int& in2)
{
        int *p;
        //Сняли volatile и записали 33
        p = const_cast<int*>(in1);
        *p = 33;
        //Сняли volatile и записали 55
        const_cast<int&>(in2) = 55;
}
//Добавление volatile
void test_func_A(int* in1,  int& in2)
{
        volatile int *p;
        //Добавили volatile и записали 33
        p = const_cast<volatile int*>(in1);
        *p = 33;
        //Добавили volatile и записали 55
        const_cast<volatile int&>(in2) = 55;
}
```

```
int main()
{
        int x=3,y=5;
        std::cout<<x<<" "<<y<<std::endl;
        //Снимаем константность
        test_func_X(&x,y);
        std::cout<<x<<" "<<y<<std::endl;
        x=3;
        y=5;
        //Добавляем константность
        test_func_Y(&x,y);//Ошибка!!!
        std::cout<<x<<" "<<y<<std::endl;
        //Снимаем volatile
        test_func_Z(&x,y);
        std::cout<<x<<" "<<y<<std::endl;
        x=3;
        y=5;
        std::cout<<x<<" "<<y<<std::endl;
        //Добавляем volatile
        test_func_A(&x,y);
        std::cout<<x<<" "<<y<<std::endl;
        system("pause");
        return 0;
}
```

Второй пример

```
#include <iostream>

using namespace std;

void f(int *x)
{
    cout << __PRETTY_FUNCTION__ << endl;
}

void f(const int *x)
{
    cout << __PRETTY_FUNCTION__ << endl;
}

int main()
{
    int x = 5;
    int *px = &x;

    f(px);
    f(const_cast<const int*>(px));

    return 0;
}
```

Квалификаторы const и volatile можно удалить или добавить только с помощью оператора приведения const_cast и приведения типов в стиле языка C. Другие операторы приведения типов не влияют на квалификаторы const и volatile (reinterpret_cast, static_cast, dynamic_cast).

---

reinterpret_cast

Оператор приведения reinterpret_cast используется для приведения несовместимых типов. Может приводить целое число к указателю, указатель к целому числу, указатель к указателю (это же касается и ссылок). Является функционально усеченным аналогом приведения типов в стиле языка C. Отличие состоит в том, что reinterpret_cast не может снимать квалификаторы const и volatile, а также не может делать небезопасное приведение типов не через указатели, а напрямую по значению. Например, переменную типа int к переменной типа double привести при помощи reinterpret_cast нельзя.

Общий вид приведения:

reinterpret_cast<new_type>(exp)

```cpp
#include <iostream>
//Пустые классы только
//для теста приведения
struct AAA{
};
struct BBB{
};
//Наследники BBB
struct BBB_X:BBB{
};
struct BBB_Y:BBB{
};

int main()
{
        //Переменные простых типовы и указатели на переменные простых типов
        int    i = 5;
        double d = 111.222;
        char   c = 'a';
        int*   pi = &i;
        double * pd = &d;
        const int* cpi = &i;
        void*  v = NULL;
        //Объекты классов
        AAA A;
        BBB B;
        BBB_X BX;
        BBB_Y BY;
        //Указатели на объекты классов
```

```
        //Указатели на объекты классов
        AAA* pA = &A;
        BBB* pB = &B;
        BBB_X* pBX = &BX;
        BBB_Y* pBY = &BY;
        //Приводим явно double к int
        i = reinterpret_cast<int>(d);//Ошибка!!!
       //и наоборот
        /d = reinterpret_cast<int>(i);//Ошибка!!!
        //указатель на int к char
        c = reinterpret_cast<char>(pi);
        //char к указателю на void
        v = reinterpret_cast<void*>(c);
        //указатель на void к указателю на int
        pi = reinterpret_cast<int*>(v);
        //Снимаем константность const int*
        pi = reinterpret_cast<int *>(cpi);//Ошибка!!!
        //Приводим указатель на объект AAA к указателю на объект BBB
        //из разных иерархий
        pA = reinterpret_cast<AAA*>(pB);
        //Приводим указатель на double к double
        d = reinterpret_cast<double>(pd);//Ошибка!!!
        //А если наоборот?
        pd = reinterpret_cast<double*>(d0;//Ошибка!!!
        //Перемещение из одной иерархии наследования в другую
        pB = reinterpret_cast<BBB*>(pBX);
        pBY = reinterpret_cast<BBB_Y*>(pB);
        return 0;
}
```

static_cast

Оператор приведения static_cast применяется для неполиморфного приведения типов на этапе компиляции программы. Отличие static_cast от приведения типов в стиле языка C состоит в том, что данный оператор приведения может отслеживать недопустимые преобразования, такие как приведение указателя к значению или наоборот (unsigned int к указателю на double не приведет), а также приведение указателей и ссылок разных типов считается корректным только, если это приведение вверх или вниз по одной иерархии наследования классов, либо это указатель на void. В случае фиксации отклонения от данных ограничений будет выдана ошибка при компиляции программы. При множественном наследовании static_cast может вернуть указатель не на исходный объект, а на его подобъект.

Общий вид приведения:

static _cast<new_type>(exp)

```cpp
#include <iostream>
//Пустые классы только
//для теста приведения
struct AAA{
};
struct BBB{
};
//Наследники BBB
struct BBB_X:BBB{
};
struct BBB_Y:BBB{
};

int main()
{
        //Переменные простых типовы и указатели на переменные простых типов
        int    i = 5;
        double d = 111.222;
        char   c = 'a';
        int*   pi = &i;
        double * pd = &d;
        const int* cpi = &i;
        void*  v = NULL;
        //Объекты классов
        AAA A;
        BBB B;
        BBB_X BX;
        BBB_Y BY;
        //Указатели на объекты классов
```

```
        //Указатели на объекты классов
        AAA* pA = &A;
        BBB* pB = &B;
        BBB_X* pBX = &BX;
        BBB_Y* pBY = &BY;
        //Приводим явно double к int
        i = static_cast<int>(d);
        //и наоборот
        d = static_cast<int>(i);
        //указатель на int к char
        c = static_cast<char>(pi);//Ошибка!!!
        //char к указателю на void
        v = static_cast<void*>(c);//Ошибка!!!
        //указатель на void к указателю на int
        pi = static_cast<int*>(v);
        //Снимаем константность const int*
        pi = static_cast<int *>(cpi);//Ошибка!!!
        //Приводим указатель на объект AAA к указателю на объект BBB
        //из разных иерархий
        pA = static_cast<AAA*>(pB);//Ошибка!!!
        //Приводим указатель на double к double
        d = static_cast<double>(pd);//Ошибка!!!
        //А если наоборот?
        pd = static_cast<double*>(d0);//Ошибка!!!
        //Перемещение из одной иерархии наследования в другую
        pB = static_cast<BBB*>(pBX);
        pBY = static_cast<BBB_Y*>(pB);
        return 0;
}
```

dynamic_cast

Оператор приведения dynamic_cast применяется для полиморфного приведения типов на этапе выполнения программы (класс считается полиморфным, если в нем есть хотя бы одна виртуальная функция). Если указатель, подлежащий приведению, ссылается на объект результирующего класса или объект класса производный от результирующего то приведение считается успешным. То же самое для ссылок. Если приведение невозможно, то на этапе выполнения программы будет возвращен NULL, если приводятся указатели. Если приведение производится над ссылками, то будет сгенерировано исключение std::bad_cast. Несмотря на то, что dynamic_cast предназначен для приведения полиморфных типов по иерархии наследования, он может быть использован и для обычных неполиморфных типов вверх по иерахии. В этом случае ошибка будет получена на этапе компиляции. Оператор приведения dynamic_cast приводить к указателю на void, но не может приводить указатель на void к другому типу. Способность dynamic_cast приводить полиморфные типы обеспечивается системой RTTI (Run-Time Type Identification), которая позволяет идентифицировать тип объекта в процессе выполнения программы. При множественном наследовании dynamic_cast может вернуть указатель не на исходный объект, а на его подобъект.

Общий вид приведения:

dynamic_cast <new_type>(exp)

```cpp
#include <iostream>
//Пустые классы только
//для теста приведения
struct AAA{
        //Сделали полиморфным
        virtual void do_some(){};
};
struct BBB{
        //Сделали полиморфным
        virtual void do_some(){};
};
//Наследники BBB
struct BBB_X:BBB{
};
struct BBB_Y:BBB{
};
int main()
{
        //Переменные простых типовы и указатели на переменные простых типов
        void*  v = NULL;
        //Объекты классов
        AAA A;
        BBB B;
        BBB_X BX;
        BBB_Y BY;
        //Указатели на объекты классов
```

```
        //Указатели на объекты классов
        AAA* pA = &A;
        BBB* pB = &B;
        BBB_X* pBX = &BX;
        BBB_Y* pBY = &BY;
        //Приводим указатель на объект AAA к указателю на объект BBB
        //из разных иерархий
        pA = dynamic_cast<AAA*>(pB);
        if (pA == NULL)
        {
                std::cout<<"FAIL"<<std::endl;//Ошибка на этапе выполнения!!!
        }
        //Приводим указатель на void к указателю на объект BBB
        pB = dynamic_cast<AAA*>(v); //Ошибка на этапе компиляции!!!
        //Приводим указатель на BBB к указателю на void
        v = dynamic_cast<void*>(pB);
        //Перемещение из одной иерархии наследования в другую
        pB = dynamic_cast<BBB*>(pBX);
        pBY = dynamic_cast<BBB_Y*>(pB);
        if (pBY == NULL)
        {
                std::cout<<"FAIL"<<std::endl;//Ошибка на этапе выполнения!!!
        }
        system("pause");
        return 0;
}
```

Дополнительно рекомендую посмотреть

https://www.lektorium.tv/lecture/13733

Видеолекция Евгения Линского с проекта Лекториум

Вопросы ?

Вы можете задать вопросы и получить на них ответ на форуме

https://moodle.voenmeh.ru/mod/forum/discuss.php?d=35