КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 14 — Основные концепции C++

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.11.17

Оглавление

Элементы языка C++

Объектно-ориентированная парадигма

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

ООП — подход к программированию как к моделированию информационных объектов, позволяющий в ряде случаев упростить управление данными и упорядочить их организацию при реализации крупных проектов.

Основные принципы ООП

- класс;
- объект;
- абстракция данных;
- инкапсуляция;
- наследование;
- полиморфизм подтипов.

Терминология

Класс — универсальный, агрегатный тип данных, состоящий из тематически единого набора членов в виде переменных более элементарных типов, а также процедур и функций для работы с этими полями. В некоторых ОО языках эти функции называются «методы».

Класс является моделью информационной сущности с состоянием, а также с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями членов).

Объединяет состояние и поведение.

Объект — сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. ООП рассматривает лишь абстракцию данных, подразумевая под этим набор наиболее значимых характеристик объекта.

Инкапсуляция — свойство системы, позволяющее объединить данные и методы (процедурыи функции), работающие с ними, в виде единой сущности — класса.

Наследование — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется базовым, родительским или суперклассом.

Новый класс — потомком, наследником, дочерним или производным классом.

Полиморфизм подтипов (в ООП называемый просто «полиморфизмом») — свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Другой вид полиморфизма — параметрический — в ООП называют обобщённым программированием (шаблоны).

Детали общего

ОО — нечеткая концепция, которая имеет как хорошие, так и плохие стороны. Прежде всего, он часто реализуется неполным, неполноценным способом (C++, Java), что подрывает ее полезность.

Чтобы воспользоваться преимуществами ОО-программирования, язык, помимо того, что реализовано в C++, должен обеспечить:

- распределение всех переменных в куче;
- доступность сопрограмм;
- правильную реализацию обработки исключений;
- сборку мусора.

Это очень дорого с точки зрения реализации на существующих вычислительных архитектурах.

Тем не менее, ОО-модель включает в себя несколько хороших идей:

- идея равноценности между процедурами и структурами данных;
- иерархическое разбиение пространства имен переменных;
- использование кучи вместо стека для размещения переменных;
- группирования связанных процедур, работающих на одних и тех же данных;
- специальная процедура инициализации, называемая конструктором.

Идея равноценности между процедурами и структурами данных

Эта идея первоначально появилась в языках моделирования и вошла в языки программирования общего назначения под влиянием Simula 67.

Иерархическое разбиение пространства имен переменных

Иерархическое разбиение пространства имен переменных на деревья с «наследованием» представляет собой довольно аккуратный метод доступа к данным более низкого уровня из абстракций более высокого уровня.

Это чем-то напоминает структурирование файловой системы Unix. Это то, что делает создание больших, сложных систем проще, или даже просто возможным.

Без пространств имен разработка больших программ требует железной дисциплины именования переменных, на которую была способна только IBM, но это уже в прошлом.

Но это не единственный способ реализовать пространства имен.

Очень полезны неиерархические пространства имен (общий блок в фортране и даже в ассемблере nasm).

Использование кучи вместо стека для размещения переменных

Первым широко используемым языком с распределением переменных в куче был Java.

Ни C, ни C++ не используют кучу для автоматического распределения памяти, что вместе с отсутствием проверки индексов приводит к уязвимости переполнения буфера.

Более безопасные языки для разработки приложений стали доступны где-то с 1990 года благодаря огромному прогрессу в аппаратном обеспечении.

C++ стал попыткой исправить некоторые недостатки C, как языка разработки приложений, заимствуя некоторые идеи из Simula67.

Специальная процедура инициализации, называемая конструктором

Идея типа распространена на структуры и предоставляется механизм создания нового экземпляра структуры с автоматическим заполнением необходимых элементов, для чего используется специальная процедура инициализации — конструктор.

Модель памяти C++

Фундаментальной единицей памяти в модели памяти C ++ является байт.

Байт имеет достаточно большой размер, чтобы содержать любой элемент основного набора символов выполнения и восьмибитных кодовых единиц формата кодирования Unicode UTF-8.

Байт состоит из непрерывной последовательности битов, число которых определяется реализацией.

Младший значащий бит называется младшим битом, самый значимый бит называется старшим битом.

Память, доступная для программы на C ++, состоит из одной или нескольких последовательностей непрерывных байтов. Каждый байт имеет уникальный адрес.

Область памяти — это либо объект скалярного типа, либо последовательность смежных битовых полей, имеющих ненулевую ширину.

Различные функциональные элементы языка, такие как ссылки и виртуальные функции, могут включать дополнительные области памяти, которые не доступны для программ и управляются реализацией.

Два потока выполнения могут обновлять и получать доступ к разным ячейкам памяти, не мешая друг другу.

Битовое поле и смежное небитовое поле находятся в отдельных ячейках памяти и поэтому могут одновременно обновляться двумя потоками исполнения без помех.

Пример:
Структура, объявленная как

```
struct {
    char a;
    int b:5,
        c:11,
         :0, // заполнитель до конца адерсуемого блока
        d:8;
    struct {
        int ee:8;
    } e;
}
```

содержит четыре отдельных ячейки памяти — поле **a** и битовые поля **d** и **e.ee** являются отдельными ячейками памяти и могут изменяться одновременно, не мешая друг другу.

Битовые поля **b** и **c** вместе составляют четвертую ячейку памяти.

Битовые поля **b** и **c** не могут быть одновременно изменены, но, например, могут независимо изменяться **b** и **a**.

Объектная модель C++

Конструкции в программе на языке C++ создают *объекты*, уничтожают их, обращаются к ним и управляют ими.

Объект — это область памяти.

Функция не является объектом, независимо от того, занимает ли она память так же, как и объекты.

а) Объект создается с помощью

- определения (definition);
- выражения **new**;
- реализации (компилятором, исполняющей системой, ...) при необходимости.

б) Свойства объекта определяются при его создании.

в) Объект может иметь имя.

г) Срок существования памяти, выделенной для объекта, влияет на его время жизни.

д) Объект имеет тип. Термин *тип объекта* относится к типу, с которым объект создается.

е) Некоторые объекты *полиморфны* — компилятор генерирует информацию, связанную с каждым таким объектом. Эта информация позволяет определить тип этого объекта во время выполнения программы.

Для других объектов интерпретация найденных в них значений определяется типом выражений, используемых для доступа к ним.

ж) Объекты могут содержать другие объекты, называемые под- или суб-объектами. Подобъект может быть

- подобъектом-членом;
- подобъектом базового класса;
- элементом массива.

Пример:

```
static const char test1 = 'x';
static const char test2 = 'x';
const bool b = &test1 != &test2; // always true
```

С ++ предоставляет множество фундаментальных типов и несколько способов составления новых типов из существующих типов.

Лексические соглашения

- Раздельная трансляция — здесь все, как и в C.

- Фазы трансляции — в целом аналогичны тем, что мы имеем в случае языка C.

- Наборы символов. Основной исходный набор символов состоит из 96 символов: пробел, управляющие символы HT, VT, FF и символ новой строки, графический символ.

- Для именования других символов используется конструкция *универсального-символьного-имениISO/IEC 10646*[1] (universal-character-name):

hex-quad:

hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:

\u *hex-quad*

\U *hex-quad hex-quad*

Лексемы

Существует пять видов лексем — идентификаторы, ключевые слова, литералы[2], операторы и другие разделители. Пробелы, горизонтальные и вертикальные табуляции, новые строки, пеервод формата и комментарии (вместе составляющие «пробельный материал»), игнорируются, за исключением случаев, когда они служат для разделения лексем.

---

1 Universal Coded Character Set (UCS) или Unicode

2 (18) Литералы включают строки и символьные и числовые литералы.

Комментарии

Символы /* начинают комментарий, который заканчивается символами */.

Эти комментарии не могут быть вложенными.

Символы // начинают комментарий, который заканчивается следующим символом новой строки.

Если в таком комментарии есть символ FF или VT, между ним и новой строкой, заканчивающей комментарий, должны появляться только символы пробела.

Символы комментария //, /* и */ не имеют специального значения внутри комментария // и обрабатываются так же, как и другие символы.

Аналогично, символы комментария // и /* не имеют специального значения в комментарии /*.

Имена заголовков

header-name:

< *h-char-sequence* **>**

" *q-char-sequence* **"**

Лексемы препроцессора имени заголовка должны появляться только в директиве **#include**.

Идентификаторы

Идентификатор — это последовательность букв и цифр произвольной длины.

Два идентификатора — **`override`** и **`final`** могут иметь особое значение при появлении в определенном контексте. Кроме того, некоторые идентификаторы зарезервированы для использования компиляторами C++ и стандартными библиотеками, в связи с чем они не должны использоваться вне контекста определения.

identifier:
    identifier-nondigit
    identifier identifier-nondigit
    identifier digit
identifier-nondigit:
    nondigit
    universal-character-name
    other implementation-defined characters
nondigit: одна из
    `a b c d e f g h i j k l m n o p q r s t u v w x y z`
    `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _`
digit: одна из
    `0 1 2 3 4 5 6 7 8 9`

Ключевые слова

Идентификаторы, представленные ниже, зарезервированы в качестве ключевых слов. Ключевое слово **export** не используется[3] — оно зарезервировано для будущего.

alignas *	default	int	sizeof	using
alignof *	delete	long	static	virtual
asm	do	mutable	static_assert *	void
auto	double	namespace	static_cast	volatile
bool *	dynamic_cast	new	struct	wchar_t
break	else	noexcept	switch	while
case	enum	nullptr	template	_Alignas
catch	explicit	operator	this	_Alignof
char	export	private	thread_local *	_Bool
char16_t	extern	protected	throw	_Complex
char32_t	false	public	true	_Generic
class	float	register	try	_Imaginary
const	for	reinterpret_cast	typedef	_Noreturn
constexpr	friend	restrict	typeid	_Static_assert
const_cast	goto	return	typename	_Thread_local
continue	if	short	union	
decltype	inline	signed	unsigned	

Ключевые слова, выделенные красным, присутствуют в C, но отсутствуют в C++

* — измененные ключевые слова

3 В частности, в ISO/IEC 14882-2017.

1998 – 2011 – 2017

```
alignas          continue         friend           register         true
alignof          decltype         goto             reinterpret_cast try
asm              default          if               return           typedef
auto             delete           inline           short            typeid
bool             do               int              signed           typename
break            double           long             sizeof           union
case             dynamic_cast     mutable          static           unsigned
catch            else             namespace        static_assert    using
char             enum             new              static_cast      virtual
char16_t         explicit         noexcept         struct           void
char32_t         export           nullptr          switch           volatile
class            extern           operator         template         wchar_t
const            false            private          this             while
constexpr        float            protected        thread_local
const_cast       for              public           throw
```

Символы препроцессора, операторы и разделители

Лексическое представление программ на C++ включает в себя ряд лексем препроцессора, которые используются в синтаксисе препроцессора или преобразуются в лексемы операторов и разделителей:

```
[    ]    (    )    {    }    .    ->
++   --   &    *    +    -    ~    !
/    %    <<   >>   <    >    <=   >=   ==   !=   ^    |    &&   ||
?    :    ;    ...  =    *=   /=   %=   +=   -=   >>=  <<=  &=   ^=   |=
,    #    ##

->*  ::   .*   new  delete
```

Каждое из вышеприведенных сочетаний *preprocessing-op-or-punc* преобразуется в одну лексему в фазе трансляции.

Литералы

То, что в C называется «константа», в стандарте C++ называется «литерал».

C++	**C**
literal:	*constant:*
integer-literal	*integer-constant*
character-literal	*character-constant*
floating-literal	*floating-constant (+шеснадцатиричная константа с плавающей точкой)*
string-literal	*string-literal*
boolean-literal	
pointer-literal	
user-defined-literal	
	enumeration-constant

Строковые литералы

string-literal:

[encoding-prefix] **"** *[s-char-sequence]* **"**

[encoding-prefix] **R** *raw-string*

encoding-prefix:

u8 – стрковый литерал UTF-8

u – широкий строковый литерал **char16_t**

U – широкий строковый литерал **char32_t**

L – широкий строковый литерал **wchar_t**

s-char-sequence: s-char ...

raw-string:

" *[d-char-sequence]* **(** *[r-char-sequence]* **)** *[d-char-sequence]* **"**

r-char-sequence: r-char ...

r-char: любой элемент исходного набора символов, кроме правой круглой скобки),
за которым следует начальная *d-char-sequence* (которая может быть пустой),
за которой следует двойная кавычка ".

d-char-sequence: d-char ...

d-char: любой элемент базового исходного набора символов, кроме:

пробел,
левая скобка (,
правая скобка),
обратная косая черта \ ,
управляющие символы HT, VT, FF и LF.

Строковый литерал с префиксом **R** является «сырым» строковым литералом.
Разделителем служит *d-char-sequence*.

Булев литерал

```
boolean-literal : false
                | true
```

Булевы литералы — это ключевые слова **`false`** и **`true`**. Такие литералы являются *prvalues* и имеют тип **`bool`**.

Литерал-указатель

```
pointer-literal : nullptr
```

Литерал указателя — это ключевое слово **`nullptr`**.

Срок хранения

Срок хранения — это свойство объекта, которое определяет минимальный потенциальный срок сохранения памяти, содержащей объект.

Срок хранения определяется конструкцией, используемой для создания объекта, и является одним из следующих:

- статическая продолжительность хранения;
- продолжительность хранения потока;
- продолжительность автоматического хранения;
- динамическая продолжительность хранения.

Статические, потоковые и автоматические сроки хранения связаны с объектами, которые представлены объявлениями и неявно созданными реализацией.

Динамическая продолжительность хранения связана с объектами, созданными с помощью оператора **new**.

Категории продолжительности хранения применяются и к *ссылкам*.

Основные концепции

C++ – это язык программирования общего назначения, основанный на языке программирования C. В дополнение к возможностям, предоставляемым языком C, C++ обеспечивает:

- дополнительные типы данных;
- классы;
- шаблоны;
- исключения;
- пространства имен;
- перегрузку операторов;
- перегрузку имен функций;
- ссылки;
- операторы управления свободной памятью;
- дополнительные возможности библиотек.

Сущность – это значение, объект, ссылка, функция, перечислитель, тип, член класса, шаблон, специализация шаблона, пространство имен, пакет параметров и т.д.

Имя[4] – это использование идентификатора, идентификатора оператора-функции, идентификатора оператора-литерала, идентификатора функции преобразования или идентификатора шаблона, который обозначает сущность или метку.

Каждое имя, которое обозначает сущность, вводится *объявлением* (декларация).

Каждое имя, которое обозначает метку, вводится либо оператором **`goto`**, либо оператором с меткой.

Имя переменной обозначает ссылку или объект.

Некоторые имена обозначают типы или шаблоны.

---

4 В общем, всякий раз, когда встречается имя, необходимо определить, обозначает ли это имя одну из этих сущностей, прежде чем продолжить анализ программы, которая его содержит.

Объявления и определения

Объявление (declaration – декларация, объявление) может вводить одно или несколько имен в единице трансляции или переопределять имена, введенные предыдущими объявлениями.

Декларация определяет интерпретацию и атрибуты этих имен.

Объявление может одновременно являться и определением.

Объявление не является определением, если оно:

- объявляет функцию без указания тела функции;
- содержит спецификатор **`extern`** или спецификацию типа связывания, но не содержит инициализатора;
- объявляет невстроенный (non-inline) статический член данных в определении класса;
- объявляет статический член данных вне определения класса;
- объявление имени класса;
- декларация в перечислении;
- параметр шаблона;
- объявление параметра в объявлении функции, которое не является определением;
- объявление имнир типа **`typedef`**;
- **`alias`**-объявление;
- **`using`**-объявление или **`using`**-директива;
- объявление **`static_assert`**;
- объявление атрибута;
- пустое объявление;
- явное объявление экземпляра.

Пример: все, кроме одного – определения

```
int a;                          // определение a
extern const int c = 1;         // определение c
int f(int x) { return x+a; }    // определение f and defines x
struct S { int a; int b; };     // определение S, S::a, and S::b
struct X {                      // определение X
    int x;                      // определение non-static data member x
    static int y;               // объявление static члена данных y
    X(): x(0) { }               // определение a constructor of X
};
int X::y = 1;                   // определение X::y
enum { up, down };              // определение up и down
namespace N { int d; }          // определение N и N::d
namespace N1 = N;               // определение N1
X anX;                          // определение anX
```

А это объявления:

```
extern int a;                   // объявление a
extern const int c;             // объявление c
int f(int);                     // объявление f
struct S;                       // объявление S
typedef int Int;                // объявление Int
extern X anotherX;              // объявление anotherX
using N::d;                     // объявление d
```

Правило одного определения

Единица трансляции не должна содержать более одного определения любой переменной, функции, типа класса, типа перечисления или шаблона.

Область видимости

Декларативные области и области видимости

Каждое имя вводится в некоторой части текста программы, называемой декларативной областью (областью определения). Эта область является самой большой частью программы, в которой это имя является действительным, то есть в котором это имя может использоваться для ссылки на ту и ту же сущность.

Каждое конкретное имя допустимо только в некоторой, возможно, непрерывной части текста программы, называемой областью действия (областью видимости) имени.

Декларативная область может быть потенциальной и фактической.

Пример:

```
int j = 24;

int main( ) {

    int i = j, j;
    j = 42;
}
```

идентификатор **j** объявляется дважды как имя (и дважды используется).

Потенциальная область первого **j** – весь пример.

Фактическая заканчивается в точке объявления второго **j.**

Область видимости блока

Имя, объявленное в блоке, является локальным для этого блока – это имя имеет область видимости блока.

Его потенциальная область действия начинается в точке объявления и заканчивается в конце его блока. Переменная, объявленная в области видимости блока, является локальной переменной.

Область видимости прототипа функции

В объявлении функции или в любом декларатоpе функции за исключеним декларатора определения функции, имена параметров (если они указаны) имеют область действия прототипа функции, которая заканчивается в конце завершения декларатора.

Область действия функции

Метки имеют область действия функции и могут использоваться в любом месте функции, в которой они объявлены.

Область видимости пространства имен

Декларативная область определений пространств имен – это его тело.

Потенциальная область действия, обозначаемая некоторым пространством имен, представляет собой объединение областей, которые установлены каждым из определений пространств имен в одной и той же декларативной области с этим именем.

Пример

```
namespace N {
    int i;                  // объявление одновременно является и определением
    int g(int a) { return a; }
    int j() ;
    void q() ;
}

namespace { int l = 1; } // потенциальная область видимости l простирается
                         // от точки объявления до конца единицы трансляции
namespace N {
    int g(char a) {      // перегружает функцию N::g(int)
        return l + a;    // l здесь из неименованного пространства имен
    }
    int i;               // error: дублирующее определение
    int j() ;            // OK: дублирующее объявление функции
    int j() {            // OK: определение N::j()
        return g(i) ;    // вызывает N::g(int)
    }
    int q() ;            // error: другой тип возврата
}
```

Область видимости класса

Следующие правила описывают область видимости имен, объявленных в классах.

1) потенциальная область действия имени, объявленного в классе, состоит не только из декларативной области, следующей за точкой объявления имени, но также из всех тел функций, инициализаторов в фигурных скобках или следующих за символом «=» *нестатических* членов данных и аргументов по умолчанию в данном классе, а также во вложенных классах.

2) имя, объявленное в функции-члене, скрывает объявление с тем же именем, область которого распространяется до или после конца класса функции-члена.

Имя члена класса должно использоваться *только* следующим образом:

- в области своего класса или класса, производного от его класса;
- после оператора `.` применяется к выражению типа своего класса или класса, производного от его класса;
- после оператора `->` применяется к указателю на объект своего класса или класс, производный от его класса;
- после оператора разрешения области видимости `::` применяется к имени своего класса или класса, производного от его класса.

Область видимости перечисления

Имя перечислителя с областью действия имеет область видимости перечисления. Его потенциальная область действия начинается в точке объявления и заканчивается в конце спецификатора перечисления.

Область видимости параметров шаблона

Декларативная область имени параметра шаблона – наименьший из списков параметров шаблона, в котором было введено это имя.

Сокрытие имен (самостоятельно)

1) имя может быть скрыто явным объявлением того же имени во вложенной декларативной области или производном классе.

2) имя класса или имя перечисления может быть скрыто именем переменной, именем члена данных, функции или перечислителем, объявленными в той же области.

3) объявление имени в области видимости блока в определении функции-члена скрывает объявление члена класса с тем же именем.

4) объявление члена в производном классе скрывает объявление члена базового класса с тем же именем.

Если имя находится в области видимости и не скрыто, оно считается видимым.

Программа и связывание (самостоятельно)

Программа состоит из одного или нескольких единиц трансляции, которые связываются (компонуются) вместе.

Единица трансляции состоит из последовательности объявлений.

Говорят, что имя *имеет связь*, когда оно может обозначать тот же объект, ссылку, функцию, тип, шаблон, пространство имен или значение, что и имя, введенное объявлением в другой области видимости:

- когда имя имеет *внешний тип связывания*, на обозначаемую им сущность можно ссылаться с помощью имен из областей других единиц трансляции или из других областей той же единицы трансляции.
- когда имя имеет *внутренний тип связывания*, на обозначаемую им сущность можно ссылаться с помощью имен из областей той же единицы трансляции.
- когда имя *не имеет связывания*, на обозначаемую им сущность нельзя ссылаться с помощью имен из других областей.

Имя, имеющее *область видимости пространства имен*, может иметь как внутренний тип связывания, так и внешний.

Имя с областью видимости пространства имен имеет внутренний тип связывания, если это имя является:

- переменной, функцией или шаблоном функции, которые явно объявлены статическими;
- переменной, которая явно объявлена как **`const`** или **`constexpr`** и которая не была явно объявлена как **`extern`**;
- член данных анонимного объединения;
- непоименованное (безымянное) пространство имен;
- пространство имен, объявленное прямо или косвенно в безымянном пространстве имен.

Все остальные пространства имен имеют внешнюю связь.

Старт и завершение программы

Функция `main`

Программа должна содержать глобальную функцию, называемую **`main`**, которая является назначенным началом программы.

Запуск содержит выполнение конструкторов для объектов области именного пространства со статической длительностью хранения.

Завершение содержит выполнение деструкторов для объектов со статической длительностью хранения.

Переопределение функции **`main`** не допускается.

Функция **`main`** должна иметь тип возврата, эквивалентный типу **`int`**.

Существует два определения **`main`**:

```
int main() {
    /* … */
}
```

и

```
int main(int argc, char* argv[]) {
    /* ... */
}
```