КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 15 — Типы. Пространства имен.

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

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

Кафедра ЭВМ, 2021

2021.11.24

Оглавление

Типы

Есть два типа типов:
- фундаментальные типы;
- составные (агрегатные) типы.

Типы описывают объекты, *ссылки* или функции.

Типы могут быть cv-квалифицированными и cv-неквалифицированными.

CV-квалификаторы

Каждый cv-неквалифицированный тип имеет три cv-квалифицированные версии своего типа:

- версия с квалификацией **`const`**;
- версия с квалификацией **`volatile`**;
- версия с квалификацией **`const volatile`**.

Составной тип не квалифицируется cv-квалификаторами типов, из которых он составлен.

Любые cv-квалификаторы, применяемые к типу массива, влияют на тип элемента массива, но не на тип массива.

Существует частичное упорядочение cv-квалификаторов, так что тип можно назвать более квалифицированным по cv, чем другой.

без cv-квалификатора	<	**`const`**
без cv-квалификатора	<	**`voaltile`**
без cv-квалификатора	<	**`const voaltile`**
`const`	<	**`const voaltile`**
`voaltile`	<	**`const voaltile`**

Тип объекта – это, возможно cv-квалифицированный, тип, который *не является*:
- типом функции;
- ссылочным типом;
- пустым типом.

Арифметические типы, типы перечисления, типы указателей, указатели на типы элементов, **`std::nullptr_t`** и *cv*-квалифицированные версии этих типов совместно называются ***скалярными типами***.

POD-типы (2011)

POD расшифровывается как Plain Old Data – то есть класс без конструкторов, деструкторов и функций виртуальных членов, вне зависимости о того, определяется ли он с помощью ключевого слова **`struct`** или ключевого слова **`class`**.

Простая старая структура данных из C в C++ – это агрегатный класс, который содержит только PODы в качестве членов, не имеет пользовательского деструктора, пользовательского оператора копирования и нестатических членов типа указатель на член.

Скалярные типы, классы POD, массивы таких типов и cv-квалифицированные версии этих типов вместе называются *POD-типами*.

Скалярные типы, тривиально копируемые типы классов, массивы таких типов и cv-квалифицированные версии этих типов вместе называются *тривиально копируемыми типами*.

Скалярные типы, тривиальные типы классов, массивы таких типов и cv-квалифицированные версии этих типов вместе называются *тривиальными типами*.

Скалярные типы, типы классов стандартной компоновки, массивы таких типов и cv-квалифицированные версии этих типов вместе называются *типами стандартной компоновки*.

Для любого объекта тривиально копируемого типа **T**, лежащие в его основе байты, составляющие объект, могут быть скопированы в массив **char** или **unsigned char**.

Если содержимое массива **char** или **unsigned char** копируется обратно в объект, объект сохраняет свое первоначальное значение.

Пример тривиальности

```
#define N sizeof(T)
char buf[N];
T obj;
std::memcpy(buf, &obj, N);
/* Здесь объект obj тривиального типа может быть изменен */
std::memcpy(&obj, buf, N);
```

Между двумя вызовами **std::memcpy**, объект **obj** может быть изменен, но после второго вызова каждый подобъект **obj** скалярного типа содержит свое первоначальное значение.

Для любого тривиально копируемого типа **T**, если два указателя на **T** указывают на разные объекты **T obj1** и **obj2**, и ни **obj1**, ни **obj2** не являются подобъектом базового класса, если байты, составляющие **obj1**, копируются **в obj2**, **obj2** будет содержать то же значение, что и у **obj1**.

Пример

```
T* t1p;
T* t2p;
std::memcpy(t1p, t2p, sizeof(T));
```

Фундаментальные типы (как в C)

Объекты, объявленные как символы (**char**), являются достаточно большими для хранения любого члена базового набора символов.

Если символ из этого набора хранится в символьном объекте, целочисленное значение этого символьного объекта равно литеральному значению этого символа.

Символы могут быть явно объявлены как **unsigned** или как **signed**.

Обычный **char**, **signed char** и **unsigned char** – это три различных типа.

Они трое занимают одинаковый объем памяти и имеют одинаковые требования к выравниванию, то есть они имеют одинаковое объектное представление.

Для символьных типов все биты представления объекта участвуют в представлении значения.

Для типов **unsigned char** все возможные битовые комбинации представления значения представляют собой числа.

Однако эти требования не распространяются на другие символьные типы. Простой объект типа **char** может принимать либо те же значения, что и **signed char**, либо **unsigned char** – это определяется реализацией.

Существует пять стандартных типов целых чисел со знаком: «**signed char**», «**short int**», «**int**», «**long int**» и «**long long int**».

В этом списке каждый тип обеспечивает как минимум столько же памяти, сколько предшествующий ему в списке. Размер их достаточно большой, чтобы содержать любое значение в диапазоне **INT_MIN** и **INT_MAX**, как определено в заголовке **<climits>**.

Для каждого из стандартных целочисленных типов со знаком существует соответствующий, но отличающийся от него, стандартный целочисленный тип без знака: «**unsigned char**», «**unsigned short int**», «**unsigned int**», «**unsigned long int**» и «**unsigned long long int**”.

Каждый из них занимает тот же объем памяти и имеет те же требования выравнивания, что и соответствующий целый тип со знаком, то есть каждый целочисленный тип со знаком имеет то же представление объекта, что и соответствующий ему целочисленный тип без знака.

Стандартные и расширенные целочисленные типы без знака вместе называются целочисленными типами без знака.

Стандартные целочисленные типы со знаком и стандартные целочисленные типы без знака вместе называются стандартными целочисленными типами.

Целые числа без знака, объявленные как **unsigned**, подчиняются законам арифметики по модулю 2^n, где n – количество битов в представлении значения этого конкретного размера целого числа.

Это подразумевает, что арифметика без знака не переполняется, потому что результат, который не может быть представлен результирующим целочисленным типом без знака, уменьшается по модулю на число, которое на единицу больше наибольшего значения, которое может быть представлено результирующим целочисленным типом без знака.

Тип **wchar_t** – это отдельный тип, значения которого могут представлять различные коды для всех членов самого большого расширенного набора символов среди поддерживаемых локалей.

Тип **wchar_t** должен иметь те же требования к размеру, знвковости и выравниванию, что и один из интегральных типов, называемый его лежащим в основе типом.

Типы **char16_t** и **char32_t** обозначают различные типы с тем же размером, знаковостью и выравниванием, что и **uint_least16_t** и **uint_least32_t**, соответственно, из **<stdint.h>**.

Значения типа **bool** имеют значение **true** или **false**.. Типы или значения bool со знаком, без знака, с короткой или длинной строкой отсутствуют. Значения типа **bool** могут участвовать в целочисленных операциях.

Типы **bool**, **char**, **char16_t**, **char32_t**, **wchar_t**, а также целочисленные типы со знаком и без знака вместе называются целочисленными типами.

Существует три типа с плавающей точкой – **float**, **double** и **long double**.

Тип **double** обеспечивает, по крайней мере, такую же точность, как и **float**, а тип **long double** обеспечивает, по крайней мере, такую же точность, что и **double**.

Набор значений типа **float** является подмножеством набора значений типа **double**; набор значений типа **double** является подмножеством набора значений типа **long double**.

Представление значений типов с плавающей запятой определяется реализацией[1].

Интегральные и плавающие типы вместе называются *арифметическими типами*.

Максимальное и минимальное значения каждого арифметического типа определяется стандартным шаблоном (template) **std::numeric_limits**.

---

1 IEEE-754

Тип **void** имеет пустой набор значений.

Тип **void** является неполным типом, который в принципе не может быть завершен.

Он используется в качестве типа возврата для функций, которые не возвращают значение.

Любое выражение может быть явно преобразовано в тип **void**.

Выражение типа **void** должно использоваться только как

- операторное выражение;
- операнд выражения-запятая;
- второй или третий операнд `? :`;
- операнд **typeid** или **decltype**;
- выражение в операторе возврата для функции с возвращаемым типом **void**;
- операнд явного преобразования в тип **void**.

Значение типа **std::nullptr_t** является константой нулевого указателя. Это значение участвует в преобразованиях указателей и указателей на члены.

sizeof(std::nullptr_t) всегда равен **sizeof(void*)**.

Составные типы

Составные типы могут быть созданы следующими способами:

- массивы объектов данного типа;
- функции, которые имеют параметры заданных типов и возвращают **`void`** или ссылки или объекты данного типа;
- указатели на **`void`** или объекты или функции (*включая статические члены классов*) данного типа;
- ссылки на объекты или функции данного типа. Существует два типа ссылок – *lvalue* ссылки и *rvalue* ссылки.
- классы, содержащие последовательность объектов различных типов, набор типов, перечисления и функции для манипулирования этими объектами, а также набор ограничений на доступ к этим объектам;
- объединения, которые являются классами, способными содержать объекты разных типов в разное время;
- перечисления, которые содержат набор именованных постоянных значений. Каждое отдельное перечисление составляет отдельный перечислимый тип;
- указатели на нестатические члены класса[2], которые идентифицируют членов данного типа в *объектах данного класса*.

Тип указателя на **`void`** или указателя на тип объекта называется типом *указателя объекта*.

Однако указатель на **`void`** не является типом *указатель на объект*, потому что **`void`** не является типом объекта.

---

2 Статические члены класса являются объектами или функциями, а указатели на них являются обычными указателями на объекты или функции.

Тип указателя, который может обозначать функцию, называется типом *указателя на функцию*.

Указатели на неполные типы разрешаются, но существуют ограничения на то, что можно с ними сделать.

Каждое значение типа указателя является одним из следующих:

- указатель на объект или функцию (указатель указывает на объект или функцию);
- указатель за концом объекта;
- значение нулевого указателя для этого типа;
- неверное значение указателя.

Допустимое значение типа указателя объекта представляет собой адрес байта в памяти или нулевой указатель.

Если объект типа **T** находится по адресу **A**, указатель типа **T** *, значением которого является адрес **A**, указывает на этот объект, независимо от того, как было получено значение.

Например, считается, что адрес за концом массива указывает на несвязанный объект типа элемента массива, который может быть расположен по этому адресу.

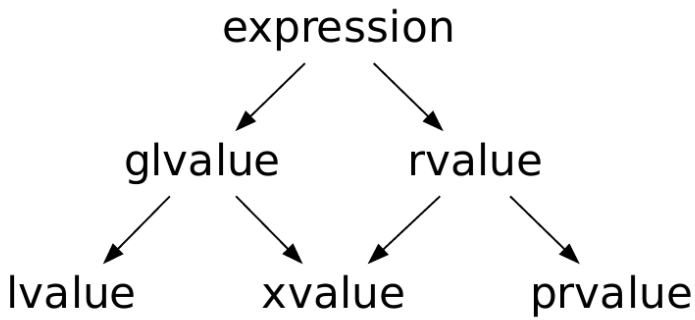
Существуют дополнительные ограничения на указатели на объекты с динамической продолжительностью существования.

Указатели на выровненные типы не имеют специального представления, но их диапазон допустимых значений ограничен требованиями к выравниванию этих типов.

Существует только два стандартных способа получения выровненного указателя – получение адреса действительного объекта с выровненным типом и использование одной из функций выравнивания указателя во время выполнения.

Lvalues и rvalues

Выражения классифицируются в соответствии со следующей таксономией



lvalue (исторически так называемое *lvalue*, которое может появляться в левой части выражения присваивания) обозначает функцию или объект.

Например, если **E** является выражением типа указателя, то ***E** является lvalue-выражением, ссылающимся на объект или функцию, на которые указывает **E**.

В качестве другого примера, результатом вызова функции, возвращаемый тип которой является ссылкой на lvalue, является lvalue.

xvalue (значение «eXpiring») относится к объекту, обычно ближе к концу его времени жизни (например, его ресурсы могут быть перемещены).

Xvalue — это результат некоторых видов выражений, включающих ссылки на *rvalue*.

Например, результатом вызова функции, возвращаемый тип которой является ссылкой на *rvalue*, является значение *xvalue*.

glvalue («обобщенное» lvalue) – это *lvalue* или *xvalue*. glvalue выражение – это выражение, вычисление которого определяет индивидуальность объекта, битового поля или функции.

rvalue – (исторически так называемое, потому что rvalue-значения могут появляться в правой части выражения присваивания) представляет собой *xvalue*, временный объект или подобъект или значение, которое вообще не связано с объектом.

prvalue («чистое» *rvalue*) – это *rvalue*, которое не является *xvalue*.

Например, результатом вызова функции, возвращаемый тип которой не является ссылкой, является *prvalue*-значение. Значение литерала, такого как **`12`**, **`7.3e5`** или **`true`**, также является prvalue-значением.

Каждое выражение относится к одной из основных классификаций в этой таксономии – lvalue, xvalue или prvalue.

Это свойство выражения называется его **категорией значений**.

Например, встроенные операторы присваивания ожидают, что левый операнд является **`lvalue`**, а правый операнд является **`prvalue`** и в результате этот оператор выдает **`lvalue`**.

Определяемые пользователем операторы являются функциями, а категории значений, которые они ожидают и получают, определяются их параметрами и типами возвращаемых данных.

Чтобы модифицировать объект, необходимо, чтобы выражение было lvalue.

Есть исключение – rvalue типа **`class`** при определенных обстоятельствах может использоваться для изменения объекта, с ним связанного.

Например, функция-член, вызываемая для объекта, может модифицировать этот объект.

Пространства имен

Пространство имен является произвольно поименованной декларативной областью.

Имя пространства имен может использоваться для доступа к объектам, объявленным в этом пространстве имен, иными словами, они являются членами данного пространства имен.

В отличие от других декларативных областей, определение пространства имен может быть разбито на несколько частей в пределах одной или нескольких единиц трансляции.

Пространство имен является самой внешней декларативной областью единицы трансляции.

Доступ к именам, объявленным внутри пространства имен можно получить используя *оператор разрешения области видимости* `::`.

Пространства имен не имеют никаких дополнительных функций.

Основная цель пространства имен состоит в том, чтобы добавить дополнительный идентификатор (имя пространства имен) к имени.

У пространства имен может быть имя — это его идентификатор или псевдоним (alias)

Пространство имен может быть:

- именованным;
- неименованным;
- вложенным.

Именованое

[**inline**] **namespace** [<список спецификаторов атрибутов>] <идентификатор> **{** <тело> **}**

Неименованное

[**inline**] **namespace** [<список спецификаторов атрибутов>] **{** <тело> **}**

Вложенное

namespace <спецификатор включающего ПИ> **::** <идентификатор> **{** <тело> **}**

<спецификатор включающего ПИ> :
    <идентификатор>
    <спецификатор включающего ПИ> **::** <идентификатор>

<тело пространства имен> — это последовательность деклараций и функций.

Каждое определение пространства имен должно появляться в глобальной области или в области пространства имен (вложенность).

В именованном-пространстве-имен-идентификаторе идентификатор является именем пространства имен.

Именованное пространство имен

[inline] namespace [<список спецификаторов атрибутов>] <идентификатор> **{** <тело> **}**

В самом простом случае определение пространства имен выглядит так:

```
namespace <идентификатор> {
    тело пространства имен
}
```

Если идентификатор является именем пространства имен (но не к псевдониму пространства имен (alias)), которое уже было введено в пространство имен, новое определение пространства имен расширяет ранее объявленное пространство имен.

Определение пространства имен может содержать в своем теле объявления пространства имен, т.е. определения пространств имен могут быть вложенными.

Пример вложенного пространства имен

```
namespace Outer {
    int i;
    namespace Inner {
        void f() { i++; } // Outer::i
        int i;
        void g() { i++; } // Inner::i
    }
}
```

Включающие пространства имен — это те пространства имен, в которых появляется объявление, но за исключением случаев переобъявления члена пространства имен вне его исходного пространства имен. Такое переобъявление имеет такое же окружающее пространство имен, что и исходное объявление.

Пример

```
namespace Q {
    namespace V {
        void f(); // включающие пространства имен --  глобальное, Q и Q::V
        class C {
           void m();
        };
    }
    void V::f() { // включающие пространства имен --  глобальное, Q и Q::V
        extern void h(); // объявление Q::V::h
    }
    void V::C::m() { // включающие пространства имен --  глобальное, Q и Q::V
    }
}
```

Если в определении некоторого пространства имен появляется необязательное начальное ключевое слово **inline**, то это пространство имен объявляется *встроенным* пространством имен.

inline namespace [<список спецификаторов атрибутов>] <идентификатор> **{**
    namespace-body
}

Необязательный <список спецификаторов атрибутов> в определении именованного пространства имен относится к тому пространству имен, которое определяется или расширяется.

Члены встроенного пространства имен могут использоваться в большинстве случаев так, как если бы они были членами включающего пространства имен.

C++17

Определение вложенного пространства имен с идентификатором **E**, содержащим пространство имен с идентификатором **I** и телом пространства имен **B** эквивалентно следующему

```
namespace E {
    namespace I {
        B
    }
}
```

Пример

```
namespace A::B::C {
    int i;
}
```

Вышеприведенное имеет тот же эффект, что и

```
namespace A {
    namespace B {
        namespace C {
            int i;
        }
    }
}
```

Безымянное пространство имен

namespace [<список спецификаторов атрибутов>] **{** <тело пространства имен> **}**
inline namespace [<список спецификаторов атрибутов>] **{** <тело пространства имен> **}**

Определение безымянного пространства имен ведет себя так, как если бы вместо него было следующее

```
[inline] namespace unique { /* empty body */ }
using[3] namespace unique ;
namespace unique {
    тело пространства имен
}
```

где все вхождения **unique** в единице трансляции заменены одним и тем же идентификатором, и этот идентификатор отличается от всех других идентификаторов в этой единице трансляции.

3 Директива **using** ...

Пример

```
namespace {
    int i;              // unique::i
}
void f() { i++; }       // unique::i++

namespace A {
    namespace {
        int i;          // A::unique::i
        int j;          // A::unique::j
    }
    void g() { i++; }   // A::unique::i++
}

using namespace A;
void h() {
    i++;          // error: unique::i or A::unique::i
    A::i++;       // A::unique::i
    j++;          // A::unique::j
}
```

```
ns.cc: В функции «void h()»:
ns.cc:20:5: ошибка: ссылка на «i» противоречива
   20 |     i++;          // error: unique::i or A::unique::i
      |     ^
```

Определение членов пространства имен

Объявление в пространстве имен **N**, за исключением объявлений во вложенных областях, объявляет (или объявляет повторно) свой идентификатор (возможно неквалифицированный), как член пространства имен **N**.

Пример

```
namespace X {
    void f() { /* ... */ }      // OK: объявляет X::f()
    namespace M {
        void g();               // OK: объявляет X::M::g()
    }
    using M::g;
    void g();                   // ошибка: конфликт с X::M::g()
}
```

Члены именованного пространства имен также могут быть *определены* вне этого пространства имен путем явной квалификации определяемого имени при условии, что определяемая сущность уже была *объявлена* в пространстве имен, и определение появляется после точки ее объявления в пространстве имен, которое охватывает данное пространство имен.

Пример

```
namespace Q {
    namespace V {
        void f();
    }
    void V::f() { /* ... */ }   // определение OK
    void V::g() { /* ... */ }   // error: g() еще не является членом V
    namespace V {
        void g();
    }
}

namespace R {
    void Q::V::g() { /* ... */ }  // error: R не включает Q
}
```

Алиасы (псевдонимы) пространств имен

Определение псевдонима пространства имен объявляет альтернативное имя для данного пространства имен в соответствии со следующей грамматикой:

namespace <идентификатор> **=** <квалифицированное имя пространства имен> **;**

где
<квалифицированное имя пространства имен> :
[<спецификатор имени вложенного пространства имен>] <имя пространства имен>

Идентификатор <идентификатор> в определении псевдонима пространства имен *является синонимом* имени пространства имен, обозначенного квалифицированным спецификатором пространства имен, и становится псевдонимом пространства имен.

Пример

```
namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name; // OK: duplicate
namespace CWVLN = CWVLN;
```

Директива using

using namespace <имя пространства имен> **;**

Директива **using** указывает, что в декларативной области после ее появления можно использовать имена из указанного пространстве имен в их неквалифицированном виде.

Директива **using** не может появляться в области видимости класса, но может появляться в области имен или в области блоков.

Пример

```
namespace A {
    int i;
    namespace B {
        namespace C {
            int i;
        }
        using namespace A::B::C;
        void f1() {
            i = 5; // OK, C::i visible in B and hides A::i
        }
    }
    namespace D {
        using namespace B;
        using namespace C;
        void f2() {
            i = 5; // двусмысленность, B::C::i or A::i?
        }
    }
    void f3() {
        i = 5; // uses A::i
    }
}
void f4() {
    i = 5; // ill-formed; neither i is visible
}
```