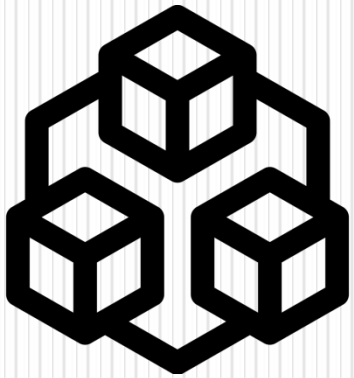
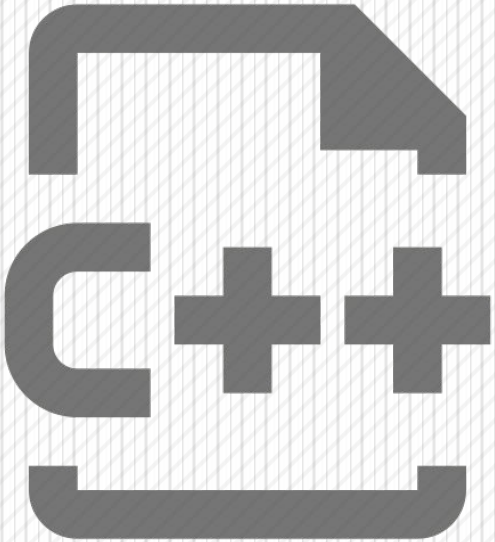
Лекция 8

Работа с типами и структурами данных





Содержание

Типы данных, определяемые пользователем

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе простых типов данных, массивов и указателей.

Язык C++ позволяет программисту определять свои типы данных и правила работы с ними. Исторически для таких типов сложилось наименование типы данных, определяемые пользователем, хотя правильнее было бы назвать их типами, определяемыми программистом.

Переименование типов (typedef)

Помимо явного объявления типа в C++ предусмотрены дополнительные средства описания имён типов. Таким средством является typedef-объявление.

Typedef-объявление - это инструмент объявления. Средство ввода новых имён в программу, средство замены громоздких последовательностей имён в объявлениях новыми именами.

Другими словами, для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова typedef:

typedef тип новое_имя [размерность];

В данном случае квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать.

С помощью typedef в программу можно ввести новые имена, которые затем можно использовать для обозначения производных и основных типов.

Далее рассмотрены примеры работы с typedef-объявлением.

```
typedef unsigned int UINT;
typedef char Msg[100];
typedef struct{
char fio[30];
int date, code;
double salary;}
Worker;
```

Введенное таким образом имя можно использовать также образом, как и имена стандартных типов:

```
UINT i, j;                    // две переменных типа unsigned int
Msg str[10];                  // массив из 10 строк по 100 символов
Worker stuff[100];            // массив из 100 структур
```

Это объявление начинается спецификатором typedef, содержит спецификатор объявления int и список описателей, в который входит два элемента: имя Step и имя pInteger, перед которым стоит символ ptr Операции *.

Объявление эквивалентно паре typedef-объявлений следующего вида:

typedef int Step;

typedef int *pInteger;

В соответствии с typedef-объявлениями, транслятор производит серию подстановок, суть которых становится понятной из анализа примера, в котором пара операторов объявления

Step StepVal;

extern pInteger pVal;

Заменяется следующими объявлениями:

int StepVal;

extern int *pVal;

Синтаксис typedef

Таким образом, typedef - объявление является объявлением, которое начинается спецификатором typedef и состоит из последовательностей разнообразных спецификаторов объявления и описателей. Список описателей (элементы списка разделяются запятыми) может содержать языковые конструкции разнообразной конфигурации. В него могут входить описатели с символами ptrОпераций (* и &), описатели, заключённые в круглые скобки, описатели в сопровождении заключённых в скобки списков объявлений параметров, описателей const и volatile, а также заключённых в квадратные скобки константных выражений.

Пример typedef-объявлениия:

typedef int Step, *pInteger;

Алгоритм подстановки

- после возможного этапа декомпозиции списка описателей typedef-объявления, в результате которого может появиться новая серия typedef-объявлений, транслятор переходит к анализу операторов объявлений;
- в очередном операторе объявления выделяется идентификатор, стоящий на месте спецификатора объявления;
- среди typedef-объявлений производится поиск соответствующего объявления, содержащего вхождение этого идентификатора в список описателей. Таким образом, транслятор находит соответствующий контекст для подстановки. Этот контекст называется контекстом замены. Контекст замены оказывается в поле зрения транслятора вместе с оператором объявления, в котором транслятор различает спецификатор объявления и описатель;
- оператор объявления заменяется контекстом замены, в котором совпадающий со спецификатором объявления идентификатор заменяется соответствующим описателем.

Если в программе присутствует typedef-объявление

typedef char* (*PPFF) (int,int,int*,float);

то компактное объявление функции

PPFF ReturnerF(int, int);

преобразуется при трансляции в сложное, но абсолютно корректное объявление:

char* (*ReturnerF(int, int))(int,int,int*,float);

При этом по идентификатору PPFF в прототипе функции находится контекст замены char* (*PPFF) (int,int,int*,float), в котором замещаемый описатель PPFF заменяется замещающим описателем ReturnerF(int, int).

То же самое typedef-объявление позволяет построить следующее объявление функции:

void MyFun (int, int, int*, float, PPFF);

Далее представлен еще один пример.

typedef long double NewType;

/*

Используется спецификатор для ввода в программу нового имени типа.

***/**

:::::

NewType MyFloatVal;

Правила

Использование спецификатора typedef подчиняется следующим правилам:

1. Спецификатор typedef может переопределять имя как имя типа, даже если это имя само уже было ранее введено typedef спецификатором:

```
typedef int I;
typedef I I;
```

2. Спецификатор typedef не может переопределять имя типа, объявленное в одной и той же области действия, и замещающее имя другого типа.

```
typedef int I;
typedef float I; // Ошибка: повторное описание
```

3. На имена, введённые в программу с помощью спецификатора typedef, распространяются правила области действия, за исключением разрешения на многократное использование имени.

Перечисления (enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант.

Перечисления представляют собой список идентификаторов, введенных пользователем:

enum name_list {name1,name2,...};

За каждым таким именем по умолчанию закрепляются целочисленные константы:

имени name1 соответствует константа 0;

имени name2 соответствует константа 1;

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант.

Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;
...
switch (error){
case ERR_READ:    /* операторы */ break;
case ERR_WRITE:   /* операторы */ break;
case ERR_CONVERT: /* операторы */ break;
}
```

Константам ERR_READ, ERR_WRITE, ERR_CONVERT присваиваются значения 0, 1 и 2 соответственно.

Преимущество применения перечисления перед описанием именованных констант и директивой #define состоит в том, что связанные константы нагляднее; кроме того, компилятор при инициализации констант может выполнять проверку типов. При выполнении арифметических операций перечисления преобразуются в целые. Поскольку перечисления являются типами, определяемыми пользователем, для них можно вводить собственные операции.

Для инициализации значений нумератора не с 0, а с другого целочисленного значения, следует присвоить это значение первому элементу списка значений перечислимого типа.

```
// Создание перечисления
enum eDay{sn, mn, ts, wd, th, fr, st} day1;
// переменная day1 будет принимать
// значения в диапазоне от 0 до 6
day1=st;
// day1 - переменная перечисляемого типа
int i1=sn;
// i1 будет равно 0
day1= eDay(0);
// eDay(0) равно значению sn
enum(color1=255);
// Объявление перечисления, определяющего
// именованную целую константу color1
int icolor=color1;
enum eDay2{sn=1, mn, ts, wd, th, fr, st} day2;
// переменная day2 будет принимать
// значения в диапазоне от 1 до 7
```

Для перечисляемого типа существует понятие диапазона значений, определяемого как диапазон целочисленных значений, которые может принимать переменная данного перечисляемого типа. Для перечислимого типа можно создавать указатели.

Перечисление также может иметь следующее формальное описание:

enum имя_типа {список_значений}

список_объявляемых_переменных;

enum имя_типа список_объявляемых_переменных;

enum (список_элемент=значение);

Перечислимый тип описывает множество, состоящее из элементов-констант, иногда называемых нумераторами или именованными константами.

Значение каждого нумератора определяется как значение типа int. По умолчанию первый нумератор определяется значением 0, второй - значением 1 и т.д. Для инициализации значений нумератора не с 0, а с другого целочисленного значения, следует присвоить это значение первому элементу списка значений перечислимого типа.

Перечисления очень широко используются многими системными программами, особенно графическими:

```
enum line_style{SOLID_LINE,      //сплошная линия
                DOTTED_LINE,     //пунктирная линия
                CENTER_LINE,     //штрих-пунктирная линия
                DASHED_LINE,     //штриховая линия
                USERBIT_LINE};   //линия, определяемая пользователем
enum COLORS {BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN,...};
```

Системный набор операций над переменными типа перечислений довольно ограниченный: им можно присваивать значения из объявленного списка, сравнивать значения однотипных переменных, передавать в качестве параметров другим функциям.

Структуры (struct)

Структура – это объединенное в единое целое множество поименованных элементов (компонентов) данных разных типов. Компонентами структуры могут быть: переменные любых типов, массивы, другие структуры, и все должны иметь различные имена.

Формат определения структурного типа имеет вид:

struct имя структурного типа
{определения элементов};

struct – спецификатор структурного типа;

имя структурного типа – любой идентификатор;

определения элементов – совокупность одного или более описаний объектов.

Предыдущую запись можно представить как:

```
struct [ имя_типа ] {
тип_1 элемент_1;
тип_2 элемент_2;
...
тип_n элемент_n;
} [ список_описателей ];
```

Далее представлено использование различных структур:

```
struct tovar {
char name[15];  /* Наименование */
int price; /* Оптовая цена */
float percent;     /* Наценка в % */
int vol;     /* Объем партии */
char date [9]; }   /* Дата поставки */
```


Инициализация структуры

При инициализации структур непосредственно в определении конкретной структуры после ее имени и знака “=” в фигурных скобках размещается список начальных значений элементов.

Например:

struct
tovar coat = { “пиджак черный”, 4000, 7.5, 220, “12.11.99” } ;
tovar tea = { “чай зеленый”, 2500, 6, 100, “25.02.00” } ;
students student_1 = { “Игнатьев”, “Олег”, 1 } ;

Для структур одного типа допустимо следующее присваивание:

tea=coat;

Для структур не определены операции сравнения. Сравнивать структуры можно только поэлементно.

Доступ к элементам структур обеспечивается с помощью уточненных имен. Уточненное имя – это выражение с двумя операндами операцией доступа к элементам структуры (‘точка’) между ними. Уточненное имя используется для выбора правого операнда операции ‘точка’ из структуры, задаваемой левым операндом, следующим образом:

имя структуры. имя элемента

Элементы структуры называются полями структуры и могут иметь любой тип. Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов. В этом случае описание структуры служит определением элементов этого списка:

// Определение массива структур и указателя на структуру:

```
struct {
char fio[30];
int date, code;
double salary;
}stuff[100], * ps;
```


Битовые поля

Битовые поля - это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет». Минимальная адресуемая ячейка памяти - 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа):

```
struct Options{
bool centerX:1;
bool centerY:1;
unsigned int shadow:2;
unsigned int palette:4;
};
```

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом - по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры.

Объединения (union)

Объединение (union) представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у структуры, только вместо ключевого слова struct используется слово union.

Объединение позволяет размещать в одном месте памяти данные, доступ к которым реализуется через переменные разных типов. Использование объединений значительно экономит память, выделяемую под объекты.

При создании переменной типа объединение память под все элементы объединения выделяется исходя из размера наибольшего его элемента. В каждый отдельный момент времени объединение используется для доступа только к одному элементу данных, входящих в объединение.

Инициализировать объединение при его объявлении можно только заданием значения первого элемента объединения.

Например:

```
union unionA {
   char ch1;
  float f1;} a1={ 'M' };
```

Доступ к элементам объединения, аналогично доступу к элементам структур, выполняется с помощью операторов . и ->.

Например:

```
union TypeNum
{   int   i;
   long  l;
   float  f;
};
   union TypeNum vNum = { 1 };
   // Инициализация первого элемента объединения i = 1
   cout<< vNum.i;
   vNum.f = 4.13;
   cout<< vNum.f;
```

Элементы объединения не могут иметь модификаторов доступа и всегда реализуются как общедоступные (public).

Длина объединения равна наибольшей из длин его полей. В каждый момент времени в переменной типа объединение хранится только одно значение, и ответственность за его правильное использование лежит на программисте.

Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется:

```
#include "pch.h"
#include <iostream>
using namespace std;
int main(){
enum paytype {CARD, CHECK};
paytype ptype;
union payment{
char card[25];
long check;
} info;
/* присваивание значений info и ptype */
switch (ptype){
case CARD: cout << «Оплата по карте: « << info.card; break;
case CHECK: cout << «Оплата чеком: « << info.check; break;
}
return 0;
}
```

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Имя объединения можно не указывать, что позволяет обращаться к его полям непосредственно:

```
#include "pch.h"
#include <iostream>
using namespace std;
int main(){
enum paytype {CARD, CHECK};
struct{
paytype ptype;
union{
char card[25];
long check;
};
} info;
... /* присваивание значения info */
switch (info.ptype){
case CARD: cout << «Оплата по карте: « << info.card; break;
case CHECK: cout << «Оплата чеком: « << info.check; break;
}
return 0;
}
```

Объединения применяются также для разной интерпретации одного и того же битового представления (но, как правило, в этом случае лучше использовать явные операции преобразования типов).

В качестве примера рассмотрена работа со структурой, содержащей битовые поля:

```
struct Options{
bool centerX:1;
bool centerY:1;
unsigned int shadow:2;
unsigned int palette:4;
};
union{
unsigned char ch;
Options bit;
}option = {0xC4};
cout << option.bit.palette;
option.ch &= 0xF0; // наложение маски
```

Ограничения объединения

По сравнению со структурами на объединения налагаются некоторые ограничения:

- объединение может инициализироваться только значением его первого элемента;
- объединение не может содержать битовые поля;
- объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;
- объединение не может входить в иерархию классов.

Контрольные вопросы

1. Как называются элементы структуры?
2. Как инициализируется структуры?
3. С помощью, какой операции выполняется доступ к полям структуру?
4. Дайте определение битовым полям?
5. Что представляет частный случай структуры?
6. Когда применяют объединение?

Список литературы

- Павловская Т.А. С/С++. Программирование на языке высокого уровня / Т. А. Павловская. - СПб.: Питер, 2004. - 461 с.: ил.
- Павловская Т.А. С/С ++. Структурное программирование: Практикум / Т.А. Павловская, Ю.А. Щупак. СПб.: Питер, 2007. - 239 с.: ил.
- Павловская Т. А., Щупак Ю. А. С++. Объектно-ориентированное программирование: Практикум. - СПб.: Питер, 2006. - 265 с: ил.
- Кольцов Д.М. 100 примеров на Си. - СПб.: “Наука и техника”, 2017 - 256 с.
- 5 Доусон М. Изучаем С++ через программирование игр. - СПб.: “Питер”, 2016. - 352.
- Седжвик Р. Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ. Роберт Седжвик. - К.: Издательство “Диасофт”, 2001. - 688с.
- Сиддкхартха Р. Освой самостоятельно С++ за 21 день. - М.: SAMS, 2013. - 651 с.
- Стивен, П. Язык программирования С++. Лекции и упражнения, 6-е изд. Пер. с англ. - М.: ООО "И.Д. Вильямс", 2012. - 1248 с.
- Черносвитов, А. Visual C++: руководство по практическому изучению / А. Черносвитов . - СПб. : Питер, 2002. - 528 с. : ил.

Список литературы

- Страуструп Б. Дизайн и эволюция языка C++. - М.: ДМК, 2000. - 448 с.
- Мейерс С. Эффективное использование C++. - М.: ДМК, 2000. - 240 с.
- Бадд Т. Объектно-ориентированное программирование в действии. - СПб: Питер, 1997. - 464 с.
- Лаптев В.В. С ++. Объектно-ориентированное программирование: Учебное пособие.- СПб.: Питер, 2008. - 464 с.: ил.
- Страуструп Б. Язык программирования C++. Режим доступа: http://8361.ru/6sem/books/Straustrup-Yazyk_programmirovaniya_c.pdf.
- Керниган Б., Ритчи Д. Язык программирования Си. Режим доступа: http://cpp.com.ru/kr_cbook/index.html.
- Герберт Шилдт: C++ базовый курс. Режим доступа: https://www.bsuir.by/m/12_100229_1_98220.pdf,
- Богуславский А.А., Соколов С.М. Основы программирования на языке Си++. Режим доступа: http://www.ict.edu.ru/ft/004246/cpp_p1.pdf.
- Линский, Е. Основы C++. Режим доступа: https://www.lektorium.tv/lecture/13373.
- Конова Е. А., Поллак Г. А. Алгоритмы и программы. Язык C++: Учебное пособие. Режим доступа: https://vk.com/doc7608079_489807856?hash=e279524206b2efd567&dl=f85cf2703018eeaa2