[18. Структуры и другие типы данных.](#_Toc515770787)

[18.1. Структуры С и C++.](#_Toc515770788)

[18.1.1. Структуры С и C++: синтаксис и правила.](#_Toc515770789)

[18.1.2. Создание простой структуры. Типовая задача 2 – инвентаризация книг.](#_Toc515770790)

[18.1.3. Создание простой структуры. Типовая задача 2 – учет катеров.](#_Toc515770791)

[18.1.4. Структуры C++: расширения синтаксиса и правил.](#_Toc515770792)

[18.1.5. Доступ к элементам структуры.](#_Toc515770793)

[18.1.6. Инициализация структуры.](#_Toc515770794)

[18.1.7. Передача структур в функции.](#_Toc515770795)

[18.2. Массив структур.](#_Toc515770796)

[18.2.1. Описание массива структур.](#_Toc515770797)

[18.2.2. Определение элементов массива структур.](#_Toc515770798)

[18.2.3. Масив структур. Типовая задача 2.](#_Toc515770799)

[18.3. Вложенные структуры.](#_Toc515770800)

[18.3.1. Структуры внутри структур. Типовая задача 2.](#_Toc515770801)

[18.4. Использование указателей на структуры.](#_Toc515770802)

[18.4.1. Описание и инициализация указателя на структуру.](#_Toc515770803)

[18.4.2. Доступ к элементу структуры при помощи указателя.](#_Toc515770804)

[18.4.3. Использование указателя. Типовая задача 2:](#_Toc515770805)

[18.5. Структуры и функции.](#_Toc515770806)

[18.5.1. Передача структур по значению.](#_Toc515770807)

[18.5.2. Передача структур посредством указателей.](#_Toc515770808)

[18.5.3. Передача структур посредством ссылок.](#_Toc515770809)

[18.5.4. Передача элементов структуры.](#_Toc515770810)

[18.5.5. Передача массива структур.](#_Toc515770811)

[18.5.5.1. Передача информации о структурах функциям. Использование массива. Типовая задача 2 на С.](#_Toc515770812)

[18.5.5.2. Передача информации о структурах функциям. Использование массива. Типовая задача 2 на C++.](#_Toc515770813)

[18.5.6. Функция – член структуры](#_Toc515770814)

[18.5.6.1. Особенность возврата const.](#_Toc515770815)

[18.5.7. Возврат структуры из функции](#_Toc515770816)

[18.6. Структуры и битовые поля.](#_Toc515770817)

[18.7. Структуры. Их дальнейшее использование.](#_Toc515770818)

[18.8. Объединения.](#_Toc515770819)

[18.8.1. Объединения: синтаксис и правила.](#_Toc515770820)

[18.8.2. Создание простого объединения.](#_Toc515770821)

[18.9. Вспомогательные средства.](#_Toc515770822)

[18.9.1. Использование typedef.](#_Toc515770823)

[18.9.2. Использование enum.](#_Toc515770824)

[18.10. Сложные формы данных.](#_Toc515770825)

[18.11. Функции работы с датой и временем.](#_Toc515770826)

1. Структуры и другие типы данных.

Успех программы часто зависит от удачного выбора способа представления данных, с которыми она должна работать. В этом отношении языку Си очень повезло (и не случайно), так как он обладает очень мощными средствами представления сложных данных. Этот тип данных, называемых «структурой», не только достаточно гибок для представления разнообразных данных, но, кроме того, он позволяет пользователю создавать новые типы. Если вы знакомы с «записями» языка Паскаль, вам должны быть удобны структуры.

Здесь вы познакомитесь с несколькими сложными типами С и C++, такими как структуры, объединения и битовые поля, а также с некоторыми другими темами. Вы узнаете, как создавать и использовать структуры в программах, как передавать структуры в функции при помощи указателей на структуры, как создавать и использовать объединения; также вы познакомитесь с другими важными средствами, такими как typedef и перечисляемые типы (enum).

Основное внимание здесь уделено двум важным средствам, общим для С и C++: структурам и объединениям. Структура С или C++ аналогична массиву или вектору тесно связанных атрибутов. Однако, в отличие от массива или вектора структура позволяет иметь смешанные атрибуты различных типов данных.

Структуры очень важны в С и C++, они являются предшественниками более совершенного типа C++, называемого классом. После детального знакомства со структурами намного проще разбираться с классами C++. Это объясняется тем, что классы C++ содержат и дополняют многие свойства структур.

Еще один сложный тип, объединения, позволяет хранить различные типы данных в одной области памяти. Эти сложные типы данных служат основой для большинства программ электронных таблиц и баз данных.

В последующем вы узнаете, как создавать простые структуры и массивы структур, как передавать структуры и массивы структур в функции и как обращаться к элементам структуры при помощи указателей.

## Структуры С и C++.

Структурированные данные весьма распространены в повседневной жизни. Картотека, содержащая адреса знакомых, номера телефонов и прочую информацию, является структурой связанных атрибутов. Список любимых компакт-дисков или пластинок также является структурой; каталог диска — структура. В этих примерах используются структуры, но что же это такое? Структуру буквально можно рассматривать как набор переменных различных типов, образующих единый объект. Этот отдельный объект и является структурой.

### Структуры С и C++: синтаксис и правила.

В языках С или C++ структура создается при помощи ключевого слова struct, за которым следуют необязательное поле тега (tag) и список элементов структуры. Поле тега используется для создания других переменных данного структурного типа. Синтаксис структуры с необязательным полем тега выглядит следующим образом:

struct поле\_тега {

тип\_элемента элемент 1;

тип\_элемента элемент 2;

тип\_элемента элемент З;

...

...

...

тип\_элемента\_элемент N;

};

Описание структуры заканчивается точкой с запятой, поскольку на самом деле оно является оператором С и C++.

struct stuff {

int numb;

char code [4];

float cost;

};

15_01

Долее в примерах будет использоваться две структуры (упрощенная типовая задача 1) и подобная следующей (типовая задача 2):

struct stboat { // катер

char sztype [iSTRING15 + iNULL\_CHAR]; //тип

char szmodel[iSTRING15 + iNULL\_CHAR]; //модель

char sztitle[iSTRING20 + iNULL\_CHAR]; //бортовой номер

int iyear; // год выпуска

long int lmotor\_hours; // ресурс двигателя

float fsaleprice; // продажная стоимость

};

Данная структура создана при помощи ключевого слова struct, за которым следует поле тега или тип структуры. В этом примере stboat является теговым полем структуры.

Объявление этой структуры содержит несколько элементов: sztype, szmodel и sztitle являются строками заданной длины, заканчивающимися null-символом. За ними следуют целое iyear, длинное целое lmotor\_hours и число с плавающей точкой fsaleprice. Эта структура используется для хранения коммерческой информации о катере или лодке.

Таким образом описан новый гипотетический структурный тип stboat. Однако, с этой структурой не связана никакая переменная; для этого в программе можно использовать следующий оператор:

Типовая задача 1

struct book libry;

Типовая задача 2

struct stboat stused\_boat;

Этот оператор описывает переменную stused\_boat, имеющую тип struct stboat. Обратите внимание на то, что в объявлении нужно использовать поле тега структуры. Если данный оператор располагается в некоторой функции, то структура stused\_boat имеет локальную область действия внутри этой функции. Если оператор находится в программе вне всех функций, то структура имеет глобальную область действия. Также можно объявить структурную переменную, используя следующий синтаксис:

Типовая задача 1

struct book {

char title [MAXTIT];

char author [MAXAUT];

float value;

} libry; /\* присоединяет имя переменной к шаблону \*/

Типовая задача 2

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int lmotor\_hours;

float fsaleprice;

} stused\_boat;

Объявление переменной заключено между фигурной скобкой, закрывающей структуру (}), и обязательной точкой с запятой. В обоих примерах переменная stused\_boat объявляется как структура типа stboat. На практике, если со структурным типом связывается только одна переменная, поле тега можно убрать, поэтому предыдущий пример можно записать и так:

Типовая задача 1

struct { /\* без имени типа структуры \*/

char title [MAXTIT];

char author [MAXAUT];

float value;

} libry;

Типовая задача 2

struct {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int lmotor\_hours;

float fsaleprice;

} stused\_boat;

Заметьте, что в этом объявлении структуры отсутствует поле тега и создается так называемый анонимный структурный тип. Поскольку оператор описывает единственную переменную, stused\_boat, то нигде в приложении нельзя создать другую переменную такого же типа. Если поле тега структуры отсутствует, то такой синтаксис не позволяет ссылаться на этот новый тип. Однако, с одним структурным типом можно связать несколько переменных, не указывая поле тега; это показано в следующем примере:

Типовая задача 1

struct { /\* без имени типа структуры \*/

char title [MAXTIT];

char author [MAXAUT];

float value;

} libry1, libry2, libry3;

Типовая задача 2

struct {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int lmotor\_hours;

float fsaleprice;

} stboat1, stboat2, stboat3;

Форма с именем типа структуры удобнее, если вы используете структурный шаблон более одного раза.

Так же, как и для любых других переменных, компиляторы С и C++ выделяют необходимую память для элементов структуры. Чтобы определить, нужно ли поле тега в объявлении структуры, нужно ответить на следующие вопросы: "Нужно ли где-нибудь в программе создавать другие переменные такого структурного типа?" и "Нужно ли передавать структурный тип в функции?" Если на какой-нибудь из вопросов имеется положительный ответ, то поле тега — необходимо.

Как правило, объекты, представляемые в программе, обладают рядом разнообразных и, что очень важно, разнотипных свойств. Конструирование таких объектов на С++ предусматривает использование структур и объединений.

Структура может быть представлена как некоторый набор разнотипных и/или однотипных данных, совокупность которых рассматривается как совершенно новый, пользовательский тип данных. Структура объявляется с помощью ключевого слова struct, за которым следует необязательное имя тела для создания нового типа данных и указываемый в фигурных скобках шаблон, по которому будут создаваться переменные структурного типа. Шаблон содержит указываемые через точку с запятой объявления полей или членов структуры. Объявление поля состоит из указания типа и имени переменной:

struct NewType

{

type1 Name1;

type2 Name2;

…

typeN NameN;

};

Синтаксис описания структуры заканчивается символом точка с запятой (;).

Использование структурированных данных в теле программы возможно в том случае, если будет объявлен какой-нибудь объект вновь созданного типа. Например, для приведенного выше синтаксиса можно указать:

NewType Variable;

Таким образом, будет создан структурированный объект Variable типа NewType. Кроме того, объект Variable мог быть создан непосредственно при объявлении структуры:

struct NewType

{

type1 Name1;

type2 Name2;

typeN NameN;

} Variable;

Инициализация элементов структуры может быть произведена непосредственно при объявлении. При этом присваиваемые значения указываются через запятую в фигурных скобках, например:

struct MyStruct

{

int iVariable;

long iValue;

char Str[10];

} mystruct = {10, 300L, "Hello"};

Поясним вышесказанное на примере. Предположим, в программе создается база данных (подробно базы данных рассмотрены в книге "Базы данных. Учебный курс", изд. "Фолио"), содержащая информацию о жилых домах микрорайона. Изначально известно, что для дальнейшего использования потребуются данные о номере микрорайона, названии улицы, номере дома, количестве этажей, числе квартир, наличии прилегающей стоянки. Разработаем структуру, отвечающую указанным требованиям. Пусть под номер микрорайона отводится беззнаковое короткое целое, название улицы может быть закодировано строкой из 50-ти символов, номер дома представим как строку из пяти символов (на случай, если номер дома дополнительно содержит букву), количество этажей и число квартир - беззнаковое короткое целое, а информация о наличии стоянки - логическая переменная. В конечном итоге получится что-то вроде:

struct HOUSE

{

unsigned short RegNum;

char Street[51]; // с учетом '\0'

char HouseNum[6];

unsigned short MaxFloorNum;

unsigned short MaxFlatNum;

bool Parking;

};

Для использования полученного типа HOUSE объявим соответствующую переменную House:

HOUSE House;

Следующий важный момент - доступ к элементам структуры. Чтобы записать или прочитать данные структуры, после имени объекта ставится символ точки (.), за которым следует имя члена структуры. Вся подобная конструкция рассматривается как единая переменная. В качестве примера заполним уже имеющийся объект House:

House.RegNum = 524;

strcpy(MyHouse.Street, "ул. Гоголя");

strcpy(MyHouse.HouseNum, "2-a");

House.MaxFloorNum = 7;

House.MaxFlatNum = 84;

House.Parking = true;

Как видно из примера, целочисленным и логическим данным производится обычное присвоение, а заполнение строковых членов структуры осуществлено с помощью функции работы со строками strcpy ().

Для определения размера структурированного объекта в памяти к нему применяют оператор (или функцию) sizeof. Таким образом, отдельный экземпляр структуры HOUSE будет занимать, например, 64 байта:

int i = sizeof(HOUSE);

Однако очень часто структуры помогают экономить память благодаря использованию так называемых битовых полей. В этом случае объявление поля структуры имеет вид:

объявление поля : константное\_выражение;

где

объявление поля - объявление типа и имени поля структуры;

константное выражение определяет длину поля в битах.

Тип поля должен быть целочисленным (int, long, unsigned, char) объявление\_поля может отсутствовать. В этом случае в шаблоне структуры пропускается указанное после двоеточия чисто битов. Таким образом, если разработчик знает наверняка, что элемент структуры может принимать, скажем, всего два значения (0 или 1), для него можно отвести один бит. Дальнейшая работа с таким элементом структуры ведется с использованием поразрядных логических операций. Реализация битовых полей тесно связана с аппаратной платформой, на которой функционирует компилятор. Поэтому, детали использования битовых полей следует уточнить в документации, поставляемой с вашим компилятором.

Рассмотрим небольшой пример. Допустим, необходимо создать структуру, содержащую информацию о дате и времени некоторых событий. Этого можно добиться следующим путем:

struct DATETIME

{

unsigned short Year; // год

unsigned short Month; // месяц

unsigned short Date; // дата

unsigned short Hour; // часы

unsigned short Minute; // минуты

unsigned short Second; // секунды

}

Таким образом, объект типа DATETIME в памяти будет занимать 6(элементов) х 2(байта) = 12 байт. Нетрудно заметить, что в описании такой структуры присутствует значительная избыточность, так как год может принимать значения от 0 до 99 (задействуется всего 7 бит), месяц - от 1 до 12 (4 бита), дата - от 1 до 31 (5 бит), часы, минуты и секунды - от 0 до 59 (по 6 бит на каждый элемент). Применяя битовые структуры, приведенная выше структура примет вид:

struct DATETIME2

{

unsigned Year: 7 // год

unsigned Month: 4 // месяц

unsigned Date:5 // дата

unsigned Hour: 6 // часы

unsigned Minute: 6 // минуты

unsigned Second: 6 // секунды

}

Экземпляр модифицированного типа DATETIME2 будет занимать не 64, а 5 байт (так как 34 бита могут быть размещены только в пяти байтах).

### Создание простой структуры. Типовая задача 2 – инвентаризация книг.

Гвен Гленн хочет напечатать опись своих книг. Она хотела бы занести в нее различную информацию о каждой книге: ее название, фамилию автора, издательство, год издания, число страниц, тираж и цену. Теперь некоторые из этих элементов, такие, как название, можно записать в массив строк. Другие элементы требуют массив целого типа или массив типа float. Если работать с семью различными массивами и следить за всей содержащейся в них информацией, можно сойти с ума, особенно если Гвен желает иметь несколько списков — список, упорядоченный по названиям, список, упорядоченный по авторам, по цене и т. д. Гораздо лучше было бы использовать один массив, в котором каждый элемент содержал бы всю информацию о книге.

Но какой тип данных может содержать строки и числа одновременно и как-то хранить эту информацию раздельно? Ответом должна быть, конечно, тема этого материала — структура. Чтобы посмотреть, как создается структура и как она работает, начнем с небольшого примера. Для упрощения задачи введем два ограничения: первое — мы включим в опись только название книги, фамилию автора и цену; второе — ограничим опись до одной книги. Если у вас больше книг, не беспокойтесь; мы покажем, как расширить эту программу.

Сначала посмотрите на программу и ее результат, а потом мы рассмотрим основные вопросы.

/\* инвентаризация одной книги \*/

#include <stdio.h>

#define MAXTIT 41 /\* максимальная длина названия +1 \*/

#define MAXAUT 31 /\* максимальная длина фамилии автора +1 \*/

/\* шаблон первой структуры: book является именем типа структуры \*/

struct book {

char title [MAXTIT]; /\* символьный массив для названия \*/

char author [MAXAUT]; /\* символьный массив для фамилии автора \*/

float value; /\* переменная для хранения цены книги \*/

}; /\* конец шаблона структуры \*/

void main()

{

struct book libry; /\* описание переменной типа book \*/

printf(" Enter the name of a book, please.\n");

gets(libry.title); /\* доступ к элементу title \*/

printf(" Now enter surname of the author.\n");

gets(libry.author);

printf(" And now, enter the price\n");

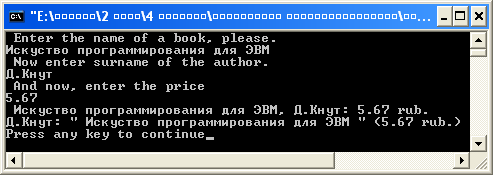
scanf(" %f", &libry.value);

printf(" %s, %s: %.2f rub.\n", libry.title, libry.author, libry.value);

printf("%s: \" %s \" \(%.2f rub.\) \n", libry.author, libry.title, libry.value);

}

Вот образец работы программы:



Созданная нами структура состоит из трех частей: одна для названия, другая для фамилии автора и третья для цены.

### Создание простой структуры. Типовая задача 2 – учет катеров.

В следующем примере используется структура, аналогичная описанной выше структуре stboat. Для того чтобы определить, понимаете ли вы, как в программе обрабатываются различные элементы структуры, проанализируйте следующий листинг программы:

/\*Программа на С, иллюстрирующая создание структуры.

Программа запоминает информацию о вашем катере в некоторой структуре С \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int Imotor\_hours;

float fsaleprice;

} stused\_boat;

int main(void)

{

printf ("\nPlease enter the type of the boat: "); /\* Введите, пожалуйста, тип катера: \*/

gets (stused\_boat.sztype);

printf ("\nPlease enter the model of the boat: "); /\* Введите, пожалуйста, модель катера: \*/

gets(stused\_boat.szmodel);

printf("\nPlease enter the title number for the boat: "); /\* Введите, пожалуйста, бортовой номер катера: \*/

gets(stused\_boat.sztitle);

printf("\nPlease enter the model year for the boat: "); /\* Введите, пожалуйста, год выпуска модели катера: \*/

scanf("%d",&stused\_boat.iyear);

printf("\nPlease enter the current hours on "); /\* Введите, пожалуйста, количество часов, отработанных двигателем катера: \*/

printf("the motor for the boat: ");

scanf("%d",&stused\_boat.Imotor\_hours);

printf("\nPlease enter the purchase price of the boat: "); /\* Введите, покупную стоимость катера \*/

scanf("%f",&stused\_boat.fsaleprice);

printf("\n\n\n");

printf("A %d %s %s with title number #%s\n", stused\_boat.iyear, stused\_boat.sztype,

stused\_boat.szmodel,stused\_boat.sztitle); /\* Катер ... с бортовьм номером .\*/

printf("currently has %d motor hours", stused\_boat.Imotor\_hours); /\* в настоящий момент отработал ... часов \*/

printf(" and was purchased for $%8.2f\n",stused\_boat.fsaleprice); /\* и был куплен за ...\*/

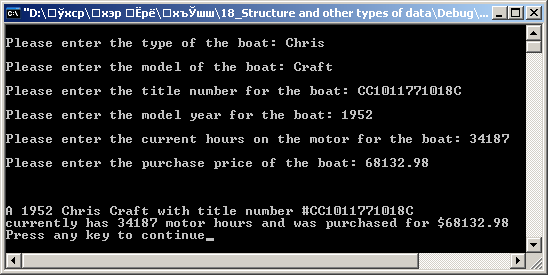
return (0);

}

Результат работы программы показывает, как информация обрабатывается в структуре:

А 1952 Chris Craft with title number #CC1011771018C

currently has 34187 motor hours and was purchased for $68132.98



Вы можете заметить, что переменная stused\_boat имеет глобальную область действия в файле, поскольку она была объявлена вне всяких функций.

### Структуры C++: расширения синтаксиса и правил.

Во многих случаях С можно считать подклассом языка C++. В общем, это означает, что все программы на С будут работать в среде C++.

**ПРИМЕЧАНИЕ**. Если использовать методы проектирования С в программах на C++, то зачастую можно упустить более развитые и простые возможности C++.

Описанные выше примеры объявлений структуры работают с компиляторами С и C++. В C++, однако, имеется еще один способ объявления переменной некоторого структурного типа. В этой краткой записи, имеющейся только в C++, отсутствует необходимость повторять ключевое слово struct. Это тонкое различие отражено в следующем примере:

/\* Объявление структуры, допустимое в С и C++ \*/

struct stboat stused\_boat;

// Объявление структуры, возможное только в C++

stboat stused\_boat;

### Доступ к элементам структуры.

Для обращения к отдельным элементам структуры можно использовать "точку" — операцию обращения к члену структуры (.). Синтаксис операции:

stname.mname

В этой записи stname является переменной, связанной со структурным типом, а mname — это имя любой переменной-члена структуры.

Например, в С можно записать информацию в элемент szmodel при помощи следующего оператора:

gets(stused\_boat.szmodel);

В этом операторе stused\_boat — имя структуры, a szmodel — переменная-член структуры. Аналогичным образом для печати элемента структуры можно использовать функцию printf():

printf("%1d",stused\_boat.lmotor\_hours);

Доступ к элементам структуры в C++ по сути такой же:

cin >> stused\_boat.sztype;

Этот оператор выполняет считывание типа катера stused\_boat в символьный массив; следующий оператор выводит на экран продажную стоимость stused\_boat

cout << stused\_boat.fsaleprice;

Элементы структуры обрабатываются так же, как и любые другие переменные С или C++; необходимо только всегда использовать операцию "точка".

### Инициализация структуры.

Мы видели, как инициализируются переменные и массивы:

int count = 0;

static int fibo [] = {0, 1, 1, 2, 3, 5, 8};

Можно ли инициализировать и структурную переменную? Да, если структурная переменная будет внешней или статической. Здесь следует иметь в виду, что принадлежность структурной переменной к внешнему типу зависит от того, где определена переменная, а не где определен шаблон. В нашем примере (1) шаблон book является внешним, а переменная libry — внутренней, так как она определена внутри функции и по умолчанию располагается в классе автоматической памяти. Предположим, мы создали такое описание:

static struct book libry; (создание структурной переменной)

В этом случае используется статическая память, и можно инициализировать структуру следующим способом:

static struct book libry = {

"Пират и девица",

"Рене Вивот",

1р.95 // Работает студенческая соображалка?

};

Чтобы сделать ассоциации более явными, мы дали каждому элементу свою собственную строку для инициализации, хотя компилятору требуются только запятые, чтобы отделить инициализацию одного элемента от инициализации следующего.

### Передача структур в функции.

Очень часто необходимо передать в функцию информацию о структуре; при этом информация передается по значению. Поскольку передается только копия информации, функция не может изменить содержимое исходной структуры. Структуру можно передать в функцию при помощи, следующей синтаксической конструкции:

fname(stvariable);

Если переменная stused\_boat описана как локальная для main() и ее объявление передается в функцию, то эту переменную можно передать в функцию с именем vprint\_data() при помощи оператора:

vprint\_data(stused\_boat);

Как можно догадаться, в прототипе функции vprint\_data() нужно объявить тип принимаемой структуры:

/\* Объявление структуры, допустимое в С и C++ \*/

void vprint\_data(struct stboat stany\_boat);

// Объявление структуры, возможное только в C++//

void vprint\_data(stboat stany\_boat);

Если передавать полные копии структур в функции, то нередко можно снизить эффективность программы. В приложениях, критичных ко времени, лучше использовать указатели. Если стоит вопрос об экономии памяти при обработке связанных списков, то вместо статически выделяемой памяти зачастую используется функция malloc(), динамически выделяющая память для структур. Долее вы увидите, как это делается. В следующем примере показано, как структура передается в функцию целиком. Обратите внимание на то, что это — простая модификация предыдущего примера. В следующих четырех примерах используется та же самая идея. В каждой программе модифицируется только та часть алгоритма, которая необходима для прояснения конкретного вопроса. Такой подход позволит вам легко анализировать текст программ и изменения синтаксиса, необходимые для реализации конкретных возможностей языка. Просмотрите листинг программы и обратите внимание на то, как структура stused\_boat передается в функцию vprint\_data().

/\* Программа на С, показывающая передачу структуры в функцию \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int lmotor\_hours;

float fsaleprice;

};

void vprint\_data(struct stboat stany\_boat);

int main(void)

{

struct stboat stused\_boat;

printf("\nPlease enter the type of the boat: ");

gets(stused\_boat.sztype);

printf("\nPlease enter the model of the boat: ");

gets(stused\_boat.szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(stused\_boat.sztitle);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&stused\_boat.iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%ld",&stused\_boat.lmotor\_hours);

printf("\nPlease enter the purchase price of the boat: ");

scanf("%f",&stused\_boat.fsaleprice);

vprint\_data(stused\_boat);

return (0);

}

void vprint\_data(struct stboat stany\_boat)

{

printf("\n\n");

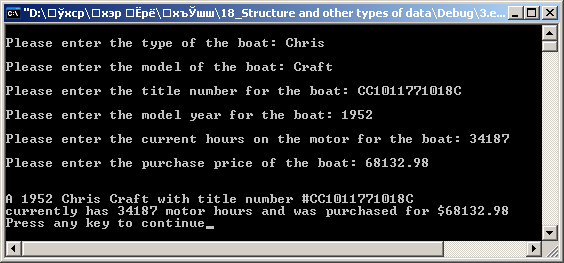
printf("A %d %s %s with title number #%s\n",stany\_boat.iyear,

stany\_boat.sztype,stany\_boat.szmodel,stany\_boat.sztitle);

printf("currently has %ld motor hours", stany\_boat.lmotor\_hours);

printf(" and was purchased for $%8.2f\n", stany\_boat.fsaleprice);

}



В приведенном примере структура целиком пересылается (вызов по значению) в функцию. Вызывающая процедура просто обращается к функции и передает ей в качестве параметра структурную переменную stused\_boat. Обратите внимание на то, что в прототипе и при объявлении функции vprint\_data() необходимо поле тега структуры stboat. Как вы увидите ниже, можно также передавать в функцию (вызов по значению) отдельные элементы структуры. Результат на выходе программы аналогичен тому, который был в предыдущем примере.

## Массив структур.

Структуру можно рассматривать как аналогию отдельной карточки в картотеке. Реальные достоинства структур проявляются тогда, когда используются набор структур, называемый массивом структур. Массив структур подобен целой картотеке, содержащей большое количество отдельных картотек. На основе такого массива можно разработать информационную базу данных, содержащую самые разные объекты.

Сама по себе одна единственная запись типа структуры в большинстве случаев вряд ли может вызывать повышенный интерес. Однако в случае, когда структурированные данные объединяются в массивы, речь идет уже не о единичном объекте, а о целой базе данных.

Объявление массива структур отличается от объявления обычных массивов лишь тем, что в качестве типа создаваемого массива указывается вновь образованный тип структуры. Таким образом, для создания базы данных, содержащей информацию о тридцати домах микрорайона, можно объявить следующий массив:

HOUSE mDistr[30];

Доступ к элементам такого массива осуществляется обычным способом, например, по индексу (индексация ведется, начиная с нуля). Следующий фрагмент кода осуществляет в цикле упорядоченное заполнение номеров домов структуры HOUSE, а затем выводит записанные данные в столбик на экран:

HOUSE mDistr[30];

for(int i=0; i<30; i++)

itoa(i+l,mDistr[i], HouseNum,10);

for(int i=0; i<30; i++)

cout << mDistr[i].HouseNum << ' \n ';

Напомним, что для работы функции преобразования целого числа в строку символов itoa() необходимо подключить модуль stdlib. h.

Поскольку запись структуры очень часто имеет внушительный размер (может включать в себя массивы элементов, другие структуры и т.д.), следует принимать во внимание ограничения по памяти при выборе соответствующей модели.

### Описание массива структур.

Процесс описания массива структур совершенно аналогичен описанию любого другого типа массива:

struct book libry [MAXBKS];

Этот оператор объявляет libry массивом, состоящим из MAXBKS-элементов. Каждый элемент массива представляет собой структуру типа book. Таким образом, libry[0] является book-структурой, libry[1] — второй book-структурой и т. д. Имя libry само по себе не является именем структуры; это имя массива, содержащего структуры.

15_02

### Определение элементов массива структур.

При определении элементов массива структур мы применяем те же самые правила, которые используются для отдельных структур: сопровождаем имя структуры операцией получения элемента и именем элемента:

libry [0].value value — первый элемент массива

libry [4].title title — пятый элемент массива

Заметим, что индекс массива присоединяется к libry, а не к концу имени:

libry.value[2] /\* неправильно \*/

libry[2].value /\* правильно \*/

Мы используем libry[2].value, потому что libry[2] является именем структурной переменной точно так же, как libry[1] является именем другой структурной переменной.

Между прочим, что бы это значило?

libry[2].title[4]

Это был бы пятый элемент элемента title (т. е. title [4]) структуры типа book, описанный третьей структурой (т. е. libry[2]). Это означает, что индексы, находящиеся справа от операции ".", относятся к отдельным элементам, в то время как индексы, расположенные слева от операции, относятся к массивам структур.

Настроим нашу программу инвентаризации книг на обработку, если потребуется, двух или трех (или, может быть, даже большего числа) книг. Очевидно, каждую книгу можно описать структурной переменной типа book. Для описания двух книг нам нужны две такие переменные и т. д. Для обработки нескольких книг потребуется массив таких структур, и мы его создали в программе

/\* инвентаризация большого количества книг \*/

#include <stdio.h>

#include <string.h>

#define MAXTIT 40

#define MAXAUT 40

#define MAXBKS 100 /\* максимальное количество книг \*/

#define STOP " " /\* нулевая строка прекращает ввод \*/

struct book { /\* создание шаблона типа book \*/

char title [MAXTIT];

char author [MAXAUT];

float value;

};

void main ()

{

struct book libry[MAXBKS]; /\* массив структур типа book \*/

int count = 0;

int index;

printf (" Please, enter the name of a book\n");

printf (" Push the buttom [enter] at the beginning of the string for stop.\n");

while (strcmp(gets(libry[count].title), STOP)!=0 && count<MAXBKS)

{

printf (" Enter now surname of the author\n");

gets (libry [count].author);

printf (" Enter now the price\n");

scanf ("%f", &libry[count++].value);

while (getchar() != '\n'); /\* очистите строку ввода \*/

if (count < MAXBKS)

printf (" Enter the title of the next book.\n");

}

printf (" There is the list of your books:\n");

for (index = 0; index < count; index++)

printf (" %s, %s: %.2f rub.\n" , libry [index]. title, libry[index] .author, libry[index] .value);

}

Вот пример работы программы:

Введите, пожалуйста, название книги.

Нажмите клавишу [ввод] в начале строки для останова.

Искусство программирования

Введите теперь фамилию автора

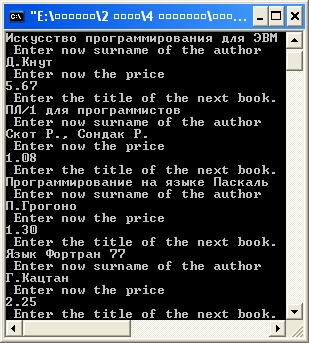
Д.Кнут

Введите теперь цену.

5р.67

Введите название следующей книги

... еще вводы ...



Вот список ваших книг:

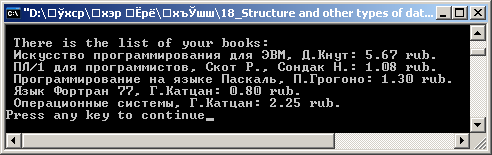
Искусство программирования для ЭВМ, Д Кнут: 5р.б7

ПЛ/1 для программистов, Скот Р., Сондак Н.: 1р 08

Программирование на языке Паскаль, П.Грогоно: 1р.30

Язык Фортран 77, Г Катцан: 0р.80

Операционные системы, Г.Катцан: 2р. 25



Следует обратить внимание на два важных момента, относящихся к массивам структур, — как описывать и как обращаться к отдельным их элементам.

Главное отличие ее от нашей первой программы заключается в том, что теперь создается цикл для считывания названий книг. Мы начинаем цикл с while-условия:

while (strcmp (gets (libry [count].title), STOP) != 0 && count < MAXBKS)

Выражение gets (libry [count].title) считывает вводимую строку, содержащую название книги. Функция strcmp () сравнивает эту строку со STOP, которая является " ", т. е. пустой строкой. Если пользователь нажмет клавишу [ввод] в начале строки, то перепишется пустая строка и цикл закончится. Мы также должны проверять, не превысило ли число считанных на текущий момент книг предельного размера массива.

В программе есть странная строка

while (getchar () ! = '\n'); /\* очистить строку ввода \*/

Она включена для того, чтобы использовать особенность функции scanf(), которая игнорирует символы пробела и новой строки. Когда вы отвечаете на запрос об элементе value в структуре book, то вводите что-нибудь вроде

12.50 [ввод]

При этом передается последовательность символов

12.50\n

Функция scanf() собирает символы 1, 2, . , 5, 0, но опускает символ \n, стоящий там, и ожидает, что следом придет еще какой-нибудь оператор чтения. Если пропустить нашу странную строку, то следующим оператором чтения будет gets (libry [count].title) в операторе управления циклом. Он прочел бы оставшийся символ новой строки как первый символ, и программа решила бы, что мы послали сигнал останова. Поэтому мы и вставили такую странную строку. Если вы поняли это, то увидите, что она «проглатывает» символы до тех пор, пока не найдет и не получит символ новой строки. Функция ничего не делает с данным символом, только принимает его от устройства ввода. Это приводит к тому, что функция gets() снова начинает работать. Вернемся теперь к изучению структур.

### Масив структур. Типовая задача 2.

Массив структур мог бы, к примеру, содержать информацию о всех категориях местного флота. Такой файл оказался бы полезным для дилера, занимающегося катерами, который смог бы выбрать из базы данных и выставить на продажу все катера не дороже 45000 долларов или все катера, имеющие минимум одну каюту. Изучите следующий пример и обратите внимание на изменения кода программы по сравнению с предыдущими примерами:

/\* Программа на С, использующая массив структур.

В этом примере создается "реестр подержанных катеров" для фирмы

Nineveh Boat Sales. \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment [80];

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

};

int main(void)

{

int i, iinstock;

struct stboat astNineveh[iMAX\_BOATS];

printf("How many boats in inventory? "); /\* Сколько катеров в реестре \*/

scanf("%d",&iinstock);

for (i=0; i<iinstock; i++) {

flushall(); /\* очистить буфер клавиатуры \*/

printf("\nPlease enter the type of the boat: ");

gets(astNineveh[i].sztype);

printf("\nPlease enter the model of the boat: ");

gets(astNineveh[i].szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(astNineveh[i].sztitle);

printf("\nPlease enter a one line comment about the boat: "); /\* Введите, пожалуйста, строку комментария: \*/

gets(astNineveh[i].szcomment);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&astNineveh[i].iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%ld",&astNineveh[i].lmotor\_hours);

printf("\nPlease enter the retail price of the boat :"); /\* Введите, пожалуйста, розничную цену катера: \*/

scanf("%f",&astNineveh[i].fretail);

printf("\nPlease enter the wholesale price of the boat :"); /\* Введите, пожалуйста, оптовую цену катера: \*/

scanf("%f",&astNineveh[i].fwholesale);

}

printf("\n\n\n");

for (i=0; i<iinstock; i++) {

printf("A %d %s %s beauty with %1d low hours.\n",

astNineveh[i].iyear,astNineveh[i].sztype, astNineveh[i].szmodel,

astNineveh[i].lmotor\_hours);

printf("%s\n", astNineveh[i].szcomment);

printf ("Grab the deal by asking your Nineveh salesperson for\n");

/\* Поторопитесь связаться с дилером фирмы Nineveh насчет \*/

printf (" #%s ONLY! $%8.2f.\n",astNineveh[i] .sztitle, astNineveh[i].fretail);

printf("\n\n");

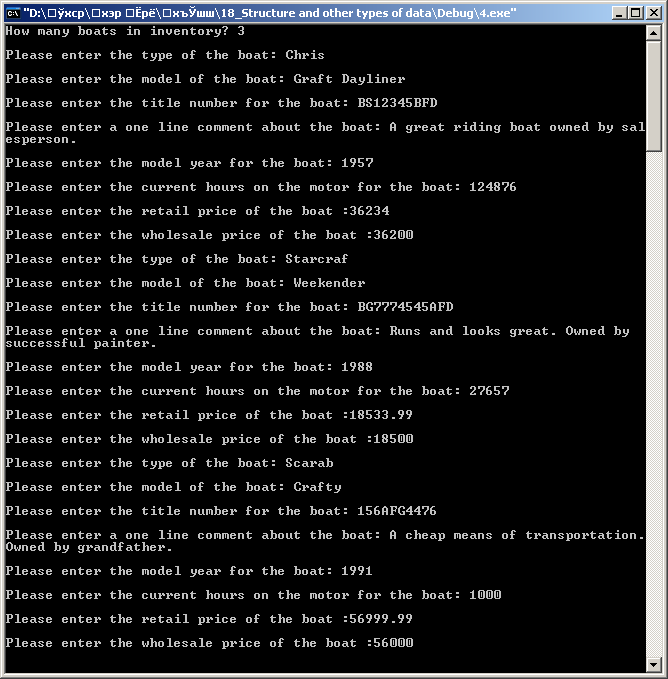
}

return (0);

}

С помощью этой программы фирма Nineveh Boat Sales может создать массив структур, предназначенный для хранения информации о всех катерах, которыми она располагает.

Вводимые значения имеют вид:



Переменная astNineveh[iMAX\_BOATS], связанная со структурой struct stboat, на самом деле является массивом. В данном случае константа iMAX\_BOATS задает максимальный размер массива, равный 50. Это означает, что в массиве структур может храниться информация о 50 катерах. Теперь необходимо занести данные о тех катерах, которые вам нужны. Первый элемент массива имеет нулевой индекс. Следовательно, к информации о первом катере, хранящейся в массиве структур, можно обратиться при помощи такого оператора:

gets(astNineveh[0].sztitle);

Изучая программу, обратите внимание на то, что доступ к элементам массива осуществляется в цикле. В этом случае для доступа к элементам нужно написать следующее:

gets(astNineveh[i].sztitle);

Оператор flushall() в цикле for необходим для того, чтобы удалить символ перевода строки, оставшийся во входном потоке от предыдущих операторов scanf() (оператора, расположенного перед циклом, и последнего оператора scanf() в цикле). Если бы не было flushall(), то оператор gets() был бы пропущен. Напоминаем, что gets() считывает все символы до перевода строки, включая сам этот символ. Операторы scanf() оставляют символы перевода строки во входном потоке. Если flushall() убрать, то оператор gets() будет перехватывать символ перевода строки из входного потока и передавать его следующему выполняемому оператору. На выходе приведенной программы можно увидеть небольшую часть списка катеров, имеющихся в распоряжении фирмы Nineveh Boat Sales. Также можно понять, как организовать вывод информации, хранящейся в структуре:

А 1957 Chris Craft Dayliner 124876 low hours.

A great riding boat owned by a salesperson.

Grab the deal by asking your Nineveh salesperson for

#BS12345BFD ONLY! $36234.00.

A 1988 Starcraft Weekender a beauty with 27657 low hours.

Runs and looks great. Owned by successful painter.

Grab the deal by asking your Nineveh salesperson for

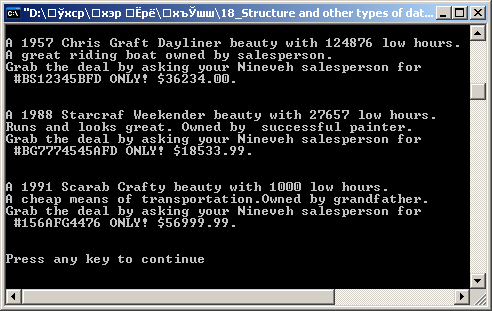
#BG7774545AFD ONLY! $18533.99.

A 1991 Scarab a wower with 1000 low hours.

A cheap means of transportation. Owned by grandfather.

Grab the deal by asking your Nineveh salesperson for

#156AFG4476 ONLY! $56999.99.



При работе с массивами структур необходимо учитывать ограничения по памяти, имеющиеся в системе программирования; при статическом выделении памяти для массивов структур может потребоваться большое количество системной памяти.

## Вложенные структуры.

Иногда бывает удобно, чтобы одна структура содержалась или была «вложена» в другую. Например, Шалала Пироски строит структуру, содержащую информацию о ее друзьях. Один элемент структуры — это, конечно, имя друга. Однако имя можно представить самой структурой с разными элементами для имени и фамилии. Здесь приведен сокращенный пример деятельности Шалалы.

/\* пример вложенной структуры \*/

#include <stdio.h>

#define LEN 20

#define M1 " Thank you for a beautiful evening,"

#define M2 " You, certainly, right, that"

#define M3 "-original guy. We should get together"

#define M4 " to taste very good"

#define M5 " and enjoy a little bit."

struct names { /\*первый структурный шаблон \*/

char first[LEN];

char last[LEN];

};

struct guy

{ /\* второй шаблон \*/

struct names handle; /\* вложенная структура \*/

char favfood[LEN];

char job[LEN];

float income;

};

void main()

{

static struct guy fellow = { /\* инициализация переменной \*/

{"Franco", "Otel"},

" eggplant",

" knitter of doormat",

15435.00

};

printf (" Dear %s, \n \n", fellow.handle.first);

printf (" %s %s.\n", M1, fellow.handle.first);

printf (" %s %s\n", M2, fellow.job);

printf (" %s\n", M3);

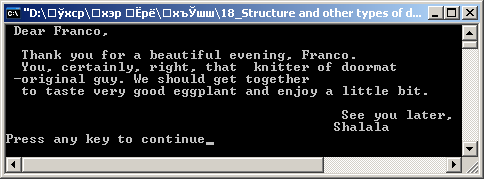
printf (" %s%s%s\n\n", M4, fellow.favfood, M5);

printf (" %40s%s\n" , " " , " See you later," );

printf (" %40s%s\n", " " , "Shalala");

}

Вот результат работы программы:



Во-первых, следует рассказать о том, как поместить вложенную структуру в шаблон. Она просто описывается точно так же, как это делалось бы для переменной типа int:

struct names handle;

Это означает, что handle является переменной типа struct names. Конечно, файл должен также содержать шаблон для структуры типа names.

Во-вторых, следует рассказать, как мы получаем доступ к элементу вложенной структуры. Нужно дважды использовать операцию ".":

fellow.handle.first = = "Франко" ;

Мы интерпретируем эту конструкцию, перемещаясь слева направо:

(fellow.handle).first

То есть первым находим элемент fellow, далее элемент handle структуры fellow, а затем его элемент first.

### Структуры внутри структур. Типовая задача 2.

Допускаются объявления вложенных структур. Это означает, что структура может содержать один или несколько элементов структурного типа. Предположим, что следующая структура может быть включена в другую:

struct strepair { // ремонт

int ioilchange; // замененное масло

int iplugs; // втулки

int lairfilter; // воздушный фильтр

int ibarnacle\_cleaning; // ветошь для чистки

};

Структуру strepair можно включить в главную структуру следующим образом:

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

struct strepair strepair\_record;

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

} astNineveh[iMAX\_BOATS] ;

Для доступа к некоторому элементу структуры strepair\_record можно использовать такой оператор:

printf("%d\n",astNineveh[0].strepair\_record.ibarnacle\_cleaning);

## Использование указателей на структуры.

Любители указателей будут рады узнать, что указатели можно использовать и для структур. Это хорошо по крайней мере по трем причинам. Во-первых, точно так же как указатели на массивы, они легче в использовании (скажем, в задаче сортировки), чем сами массивы, а указателями на структуры легче пользоваться, чем самими структурами. Во-вторых, структура не может использоваться в качестве аргумента функции, а указатель на структуру может. В-третьих, многие удобные представления данных являются структурами, содержащими указатели к другим структурам.

### Описание и инициализация указателя на структуру.

Вот самое простое описание, какое только может быть:

struct guy \*him;

Первым стоит ключевое слово struct, затем слово guy, являющееся именем структурного шаблона, далее \* и за нею имя указателя. Синтаксис тот же, как для описаний других указателей, которые мы видели.

Теперь можно создать указатель him для ссылок на любые структуры типа guy. Мы инициализируем him, заставляя его ссылаться на fellow[0]; заметим, что мы используем операцию получения адреса:

him = &fellow[0];

Первые две выведенные строки показывают результат этого присваивания. Сравнивая две строки, мы видим, что him ссылается на fellow [0], a him + 1 — на fellow[1]. Заметим, что добавление 1 к him прибавляет 84 к адресу. Это происходит потому, что каждая guy-структура занимает 84 байта памяти: первое имя — 20, последнее имя — 20, favfood — 20, job — 20 и income — 4 байта (размер элемента типа float в нашей системе).

### Доступ к элементу структуры при помощи указателя.

В предыдущем разделе him ссылается на структуру fellow [0]. Каким образом можно использовать him для получения значения элемента структуры fellow[0]? Третья выведенная строка дает для этого два способа.

Первый способ, наиболее общий, использует новую операцию - >. Она заключается в вводе дефиса (-) и следующего за ним символа «больше чем» (>). Пример помогает понять смысл сказанного:

him - > income — это fellow[0].income, если him = &fellow[0]

Другими словами, структурный указатель, за которым следует операция - >, работает так же, как имя структуры с последующей операцией ".". (Мы не можем сказать him.income, потому что him не является именем структуры.)

Очень важно отметить, что him — указатель, а him - > income — элемент структуры, на которую делается ссылка. Таким образом, в этом случае him - > income является переменной типа float.

Второй способ определения значения элемента структуры вытекает из последовательности:

если him == &fellow[0], то \*him == fellow[0].

Это так, поскольку & и \* — взаимообратные операции. Следовательно, после подстановки

fellow[0]. income = = (\*him). income

Круглые скобки необходимы, поскольку операция "." имеет приоритет выше, чем \*.

Отсюда можно сделать вывод, что если him является указателем на структуру fellow [0], то следующие обозначения эквивалентны:

fellow[0]. income == (\*him).income == him-> income

Следующий короткий пример показывает, как определять указатель на структуру и как использовать его для получения элементов структуры.

/\* указатель на структуру \*/

#include<stdio.h>

#define LEN 20

struct names {

char first [LEN];

char last [LEN];

};

struct guy {

struct names handle;

char favfood [LEN];

char job [LEN];

float income;

};

void main()

{

static struct guy fellow [2] = {

{

{"Franco", "Otel"},

"eggplant",

"knitter of doormat",

15435.00

},

{

{"Rodney", "Svilbely"},

"salmon mousse",

"interior decorator",

35000.00

}

};

struct guy \*him; /\* ЭТО------указатель на структуру \*/

printf ("address #1: %u #2: %u\n", &fellow[0], &fellow[1]);

him = &fellow [0]; /\* сообщает указателю, на что ссылаться \*/

printf ("pointer #1: %u #2: %u\n", him, him+1);

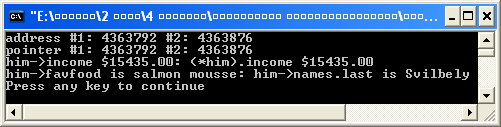
printf ("him->income $%.2f: (\*him).income $%.2f\n", him->income, (\*him).income);

him++; /\* указывает на следующую структуру \*/

printf ("him->favfood is %s: him->names.last is %s\n",him->favfood, him->handle.last);

}

Вот, пожалуйста, ее выход:



Сначала посмотрим, как мы создали указатель на структуру guy. Затем научимся определять отдельные элементы структуры при помощи указателей.

### Использование указателя. Типовая задача 2:

В следующей программе создается массив структур; в этом отношении этот пример аналогичен предыдущему. Для доступа к отдельным элементам структуры в данном примере используется операция "стрелка" (arrow). Эту операцию можно использовать только тогда, когда создан указатель на структуру.

/\* Программа на С, использующая указатель на массив структур.

Опять взят пример реестр катеров фирмы Nineveh Boat Sales \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

int iyear;

long int lmotor\_hours ;

float fretail;

float fwholesale;

};

int main(void)

{

int i, iinstock;

struct stboat astNineveh[iMAX\_BOATS], \*pastNineveh;

pastNineveh=&astNineveh[0];

printf("How many boats in inventory? ");

scanf("%d",&iinstock) ;

for (i=0; i<iinstock; i++) {

flushall(); /\* очистить буфер клавиатуры \*/

printf("\nPlease enter the make of the boat: ");

gets(pastNineveh->sztype);

printf("\nPlease enter the model of the boat: ");

gets(pastNineveh->szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(pastNineveh->sztitle);

printf("\nPlease enter a one line comment about the boat: ");

gets(pastNineveh->szcomment);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&pastNineveh->iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%d",&pastNineveh->lmotor\_hours);

printf("\nPlease enter the retail price of the boat: ");

scanf("%f",&pastNineveh->fretail);

printf("\nPlease enter the wholesale price of the boat: ");

scanf("%f",&pastNineveh->fwholesale);

pastNineveh++;

}

pastNineveh =&astNineveh[0];

printf("\n\n\n") ;

for (i=0; i < iinstock; i++) {

printf("A %d %s %s beauty with %d low hours.\n",pastNineveh->iyear,

pastNineveh->sztype,pastNineveh->szmodel,pastNineveh->lmotor\_hours);

printf("%s\n",pastNineveh->szcomment);

printf("Grab the deal by asking your Nineveh salesperson for:");

printf("\n#%s ONLY! $ %.2f \n",pastNineveh->sztitle, pastNineveh->fretail);

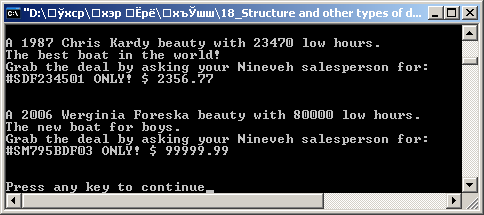
printf("\n\n");

pastNineveh++;

}

return (0);

}



Переменная-массив astNineveh[iMAX\_BOATS] и указатель \*pastNineveh связаны со структурой при помощи следующего оператора:

struct stboat astNineveh[iMAX\_BOATS],\*pastNineveh;

Адрес массива astNineveh копируется в переменную-указатель pastNineveh при помощи оператора

pastNineveh=&astNineveh[0];

Хотя синтаксис и не запрещает обращаться к элементам массива при помощи следующего оператора, однако этот метод не рекомендуется:

gets ((\*pastNineveh).sztype);

С учетом приоритета операций для того, чтобы операция обращения к члену структуры (.) выполнялась после операции разыменования указателя \*pastNineveh, необходимы дополнительные круглые скобки. Лучше использовать операцию "стрелка", делающую весь оператор намного проще:

gets (pastNineveh->sztype);

Хотя этот пример и очень простой, он иллюстрирует использование операции косвенного обращения к члену структуры "стрелка". Данный пример подводит вас к пониманию главного преимущества указателей — их использованию для передачи массива структур в функцию.

## Структуры и функции.

### Передача структур по значению.

Зачастую в тело функции необходимо передать информацию, структурированную по определенному принципу. При этом в качестве параметра в прототипе функции указывается пользовательский тип данных, сформированный при объявлении структуры. Так, если была объявлена структура

struct ALLNUMB

{

int nVar;

long lVar;

short shVar;

unsigned int uiVar;

}

прототип функции будет, например, иметь вид:

void Punc (ALLNUMB);

При этом происходит передача в тело функции параметра типа структуры по значению. Таким образом, оригинал структурированного объекта модификации не подлежит. Функция может также возвращать объект типа структуры:

ALLNUMB Func2 (ALLNUMB);

В этом случае модифицированные данные при выходе из функции не потеряются, а будут переданы в точку вызова функции.

Рассмотрим пример передачи в функцию описанной ранее структуры House для вывода на экран названия улицы и номера дома, содержащихся в ней.

#include <iostream>

using namespace std;

struct HOUSE

{

unsigned short RegNum;

char Street[51]; // с учетом '\0'

char HouseNum[6];

unsigned short MaxFloorNum;

unsigned short MaxFlatNum;

bool Parking;

};

void OutAddress(HOUSE);

int main ()

{

setlocale(LC\_ALL, "Rus");

HOUSE MyHouse;

MyHouse.RegNum = 524;

strcpy(MyHouse.Street, "ул. Гоголя");

strcpy(MyHouse.HouseNum, "2-a");

MyHouse.MaxFloorNum = 7;

MyHouse.MaxFlatNum = 84;

MyHouse.Parking = true;

OutAddress(MyHouse);

getchar(); getchar();

return 0;

}

void OutAddress(HOUSE house)

{

cout << house.Street << ",";

cout << house.HouseNum << '\n';

}

Вызов функции OutAddress (MyHouse) передает в тело сформированную структуру, доступ к членам которой осуществляется в соответствии с описанным выше правилом, через символ 'точка' (.). В результате на экран будет выведено:

ул. Гоголя, 2-а



### Передача структур посредством указателей.

Разработчик ПО на C++ имеет возможность обращаться к элементам структуры через указатели. Для этого должна быть объявлена соответствующая переменная типа указателя на структуру, синтаксис которой может быть представлен в виде:

тип структуры\* идентификатор\_указателя;

Доступ к элементам структуры через указатель осуществляется с использованием не точки, а символа стрелки (->) Например, создадим указатель на структуру house:

HOUSE \*pHouse;

pHouse = &MyHouse;

Теперь запишем в структуру информацию о наличии автостоянки около дома посредством объявленного указателя и выведем записанную информацию:

pHouse->Parking = true;

cout << MyHouse.Parking;

// или так: cout << pHouse->Parking;

Мы будем решать ту же самую задачу, что и прежде, но при этом использовать адрес структуры в качестве аргумента. Это хорошо, поскольку адрес представляет собой только одно число. Так как функция должна работать со структурой funds, она тоже должна использовать шаблон funds.

/\* передача адреса структуры в функцию \*/

#include<stdio.h>

struct funds {

char \*bank;

float bankfund;

char \*save;

float savefund; } stan = {

"Senior tomatoes Bank",

1023.43,

"Savings and lendings of Snoopy",

4239.87

};

float sum(struct funds \*);

void main()

{

float total;

total=sum(&stan);

printf ("Stan has only %.2f dollars.\n", total );

}

float sum(struct funds \*money)

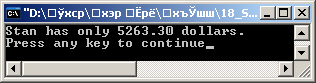
{

return ( money->bankfund + money->savefund);

}

Эта программа тоже выдает

У Стэна всего 5263.30 долл.



Функция sum() имеет указатель (money) на структуру fund. Передача адреса &stan в функцию заставляет указатель money ссылаться на структуру stan. Затем используем операцию - > для получения значений элементов stan .bankfund и stan.savefund.

Эта функция также имеет доступ к названиям учреждений, хотя их не использует.

Заметим, что мы должны применять операцию & для получения адреса структуры. В отличие от имени массива имя структуры само по себе не является синонимом своего адреса.

### Передача структур посредством ссылок.

Помимо использования указателей возможно применение ссылок на структуры. Объявление такой ссылки имеет следующий синтаксис:

тип структуры &имя\_ссылки = имя\_переменной;

Как и ссылка на обычную переменную, ссылка на структуру должна быть инициализирована именем объекта, на который она указывает (в данном случае это имя\_переменной).

Ссылки и указатели на структуры данных могут быть переданы в качестве аргументов в тело функции. При этом значительно снижается время (в сравнении с передачей по значению), за которое данный параметр передается в функцию, а также экономится стековая память.

Синтаксис прототипа функции при передаче структур посредством указателей и ссылок идентичен синтаксису обычной передачи параметров через указатели и ссылки:

// Пример передачи указателя и ссылки на

// целочисленную переменную:

bool Func(int\* ptr, int& ref);

// Передача указателя и ссылки на

// структуру типа HOUSE:

char Func2 (HOUSE\* pMh, HOUSES rMh);

Таким образом, в функцию Func2 будут переданы не сами значения структур, а соответствующие адреса, что в значительной степени экономит стековую память.

### Передача элементов структуры.

Поскольку элемент структуры является переменной с единственным значением (т. е. типа int или одного из его «родственников» — char, float, double или указатель), он может быть передан как аргумент функции. Простая программа финансовых расчетов, которая прибавляет взнос клиента к его счету, иллюстрирует этот способ. Заметим, между прочим, что мы объединили определение шаблона, описание переменной и инициализацию в один оператор.

/\* передача элементов структуры как аргументов функции \*/

#include<stdio.h>

struct funds {

char \*bank;

float bankfund;

char \*save;

float savefund; } stan = {

"Senior tomatoes Bank",

1023.43,

"Savings and lendings of Snoopy",

4239.87

};

float sum(float x, float y);

void main()

{

float total;

extern struct funds stan; /\* необязательное описание \*/

total=sum(stan.bankfund, stan.savefund);

printf ("Stan has only %.2f dollars.\n", total);

}

/\* складывает два числа типа float \*/

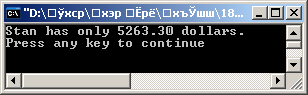
float sum(float x, float y)

{

return x+y;

}

Результат выполнения этой программы:



Вот это да, она работает. Заметим, что функция sum() «не знает», или же ей безразлично, являются ли элементами структуры фактические аргументы; она только «требует», чтобы они имели тип float.

Конечно, если вы хотите, чтобы программа воздействовала на значение элемента в вызывающей программе, можно передать ей адрес этого элемента:

modify(&stan.bankfund);

и тогда получилась бы функция, которая изменила бы банковский счет в структуре Stan.

Второй способ передачи информации о структуре заключается в возможности сообщить суммирующей функции, что она имеет дело со структурой.

### Передача массива структур.

Предположим, у нас есть массив структур. Имя массива является синонимом его адреса, поэтому его можно передать функции. С другой стороны, функции будет необходим доступ к структурному шаблону. Чтобы показать, как такая программа работает, мы расширим нашу программу таким образом, чтобы она обслуживала двух человек, а мы, следовательно, имели бы массив двух структур funds.

/\* передача массива структур в функцию \*/

#include<stdio.h>

struct funds {

char \*bank;

float bankfund;

char \*save;

float savefund; } jones[2] ={

{

"Senior tomatoes Bank",

1023.43,

"Savings and lendings of Snoopy",

4239.87

},

{

"Honest Jack's Bank",

976.57,

"Accumulation by prior plan",

1760.13

}

};

float sum(struct funds \*);

void main()

{

float total;

total=sum(jones);

printf ("Jonsons have only %.2f dollars.\n" , total);

}

float sum (struct funds \*money)

{

float total;

int i;

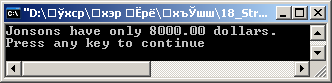
for( i = 0, total = 0; i < 2; i++ , money++)

total+= money->bankfund + money->savefund;

return(total);

}

Программа выдает



(Что за круглая сумма! Можно подумать, что эти цифры взяты с потолка.)

Имя массива Jones является указателем на массив. В частности, оно ссылается на первый элемент массива, который является структурой jones[0]. Таким образом, в начале указатель money задается через

money = &jones[0];

Затем использование операции — > позволяет нам добавить два вклада для первого Джонса. Это действительно очень похоже на последний пример. Далее, цикл for увеличивает указатель money на 1. Теперь он ссылается на следующую структуру, jones[1], и остаток вкладов может быть добавлен к total.

Вот два основных замечания:

1. Имя массива можно использовать для передачи в функцию указателя на первую структуру в массиве.
2. Затем можно использовать арифметическую операцию над указателем, чтобы передвигать его на последующие структуры в массиве. Заметим, что вызов функции

sum(&jones[0])

дал бы тот же самый эффект, что и применение имени массива, так как оба они ссылаются на один и тот же адрес. Использование имени массива является просто косвенным способом передачи адреса структуры.

Передача информации о структурах функциям. Использование массива. Типовая задача 2 на С.

Ранее уже говорилось о том, что передача указателя на структуру выполняется быстрее, чем простое копирование структуры в функцию. Этот факт становится еще очевиднее при частом использовании структур в программе. Ниже показано, как при помощи указателя функция может получить доступ к массиву структур:

/\* Программа на С, иллюстрирующая обращение функции к массиву

структур с использованием указателя

Опять используется реестр катеров фирмы Nineveh Boat Sales! \*/

#include <stdio.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

int iinstock;

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

};

void vprint\_data(struct stboat \*stany\_boatptr);

int main(void)

{

int i;

struct stboat astNineveh[iMAX\_BOATS], \*pastNineveh;

pastNineveh=&astNineveh[0];

printf("How many boats in inventory?\n");

scanf("%d",&iinstock);

for (i=0; i < iinstock; i++) {

flushall(); /\* очистить буфер клавиатуры \*/

printf("\nPlease enter the make of the boat: ");

gets(pastNineveh->sztype);

printf("\nPlease enter the model of the boat: ");

gets(pastNineveh->szmodel);

printf("\nPlease enter the title number for the boat: ");

gets(pastNineveh->sztitle);

printf("\nPlease enter a one line comment about the boat: ");

gets(pastNineveh->szcomment);

printf("\nPlease enter the model year for the boat: ");

scanf("%d",&pastNineveh->iyear);

printf("\nPlease enter the current hours on ");

printf("the motor for the boat: ");

scanf("%d",&pastNineveh->lmotor\_hours);

printf("\nPlease enter the retail price of the boat: ");

scanf("%f",&pastNineveh->fretail);

printf("\nPlease enter the wholesale price of the boat: ");

scanf("%f",&pastNineveh->fwholesale);

pastNineveh++;

}

pastNineveh = &astNineveh[0];

vprint\_data(pastNineveh);

return (0);

}

void vprint\_data(struct stboat \*stany\_boatptr)

{

int i;

printf("\n\n\n");

for (i=0; i < iinstock; i++) {

printf("A %d %s %s beauty with %ld low hours.\n", stany\_boatptr->iyear,

stany\_boatptr->sztype, stany\_boatptr->szmodel, stany\_boatptr->lmotor\_hours);

printf ("%s\n",stany\_boatptr->szcomment);

printf ("Grab the deal by asking your Nineveh salesperson for\n");

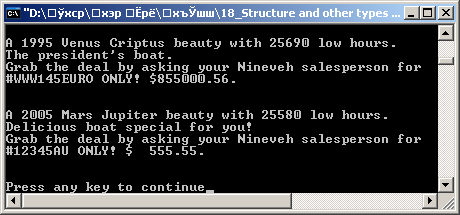
printf ("#%s ONLY! $%8.2f.\n",stany\_boatptr->sztitle, stany\_boatptr->fretail);

printf("\n\n");

stany\_boatptr++;

}

}



Первое отличие данной программы от предыдущей касается прототипа функции vprint\_data():

void vprint\_data(struct stboat \*stany\_boatptr);

Эта функция в качестве параметра получает указатель на описанную выше структуру. В функции main() массив astNineveh[iMAX\_BOATS] и указатель \*pastNineveh связываются со структурой при помощи следующего оператора:

struct stboat astNineveh[iMAX\_BOATS], \*pastNineveh;

После того как информация для реестра катеров введена, она передается в функцию vprint\_data() посредством указателя:

vprint\_data(pastNineveh);

Одним из главных преимуществ от использования указателей при передаче массива структур в функцию является то, что в этом случае массив вызывается по имени или по ссылке. Это означает, что функция обращается к исходному массиву, а не к его копии. При использовании данного метода вызова любые изменения массива структур внутри функции имеют глобальную область действия. Результат работы этой программы такой же, как и для предыдущего примера.

Передача информации о структурах функциям. Использование массива. Типовая задача 2 на C++.

Далее следует программа на C++, похожая на последнюю программу на С. Если говорить о синтаксисе, то оба языка обрабатывают структуры одинаково. Однако, в этом примере используются преимущества более компактных синтаксических конструкций C++:

// Программа на C++, иллюстрирующая использование указателей при

// обращении к данным в структуре из функции.

// Примечание: Строка комментария заканчивается точкой (.)

#include <iostream.h>

#define iSTRING15 15

#define iSTRING20 20

#define iNULL\_CHAR 1

#define iMAX\_BOATS 50

int iinstock;

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

char szcomment[80];

int iyear;

long int lmotor\_hours;

float fretail;

float fwholesale;

};

void vprint\_data(stboat \*stany\_boatptr);

int main(void)

{

int i;

char newline;

stboat astNineveh[iMAX\_BOATS],\*pastNineveh;

pastNineveh=&astNineveh[0];

cout << "How many boats in inventory? ";

cin >> iinstock;

for (i=0; i<iinstock; i++) {

cout << "\nPlease enter the make of the boat: ";

cin >> pastNineveh->sztype;

cout << "\nPlease enter the model of the boat: ";

cin >> pastNineveh->szmodel;

cout << "\n Please enter the title number for the boat: ";

cin >> pastNineveh->sztitle;

cout << "\nPlease enter the model year for the boat: ";

cin >> pastNineveh->iyear;

cout << "\nPlease enter the current hours on "<< "the motor for the boat: ";

cin >> pastNineveh->lmotor\_hours;

cout << "\nPlease enter the retail price of the boat: ";

cin >> pastNineveh->fretail;

cout << "\nPlease enter the wholesale price of the boat: ";

cin >> pastNineveh->fwholesale;

cout << "\nPlease enter a one line comment about the boat: ";

cin.get(newline); // обрабатывается возврат каретки

cin.get(pastNineveh->szcomment,80,'.');

cin.get(newline); // обрабатывается возврат каретки pastNineveh++;

}

pastNineveh=&astNineveh[0];

vprint\_data(pastNineveh);

return (0);

}

void vprint\_data(stboat \*stany\_boatptr)

{

int i;

cout << "\n\n\n";

for (i=0; i<iinstock; i++) {

cout << "A[{|"<< stany\_boatptr->iyear <<"|}] [{|"<< stany\_boatptr->sztype<<"|}] [{I"

<< stany\_boatptr->szmodel <<"I}] beauty with [{|"<< stany\_boatptr->lmotor\_hours <<"|}] low hours.\n";

cout << stany\_boatptr->szcomment << endl;

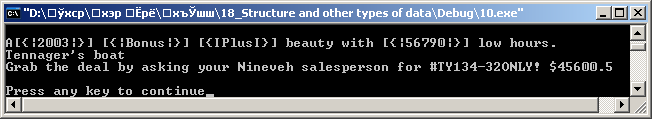
cout << "Grab the deal by asking your Nineveh "<< "salesperson for #";

cout << stany\_boatptr->sztitle << "ONLY! $"<< stany\_boatptr->fretail << "\n\n";

stany\_boatptr++;

}

}



Реальные различия между программами на С и C++ заключаются в том, как они управляют потоковым вводом/выводом. Обычно для замены стандартных функций С printfO и gets() можно использовать потоки C++ cout и cin. Например:

cout << "\nPlease enter the wholesale price of the boat: ";

cin >> pastNineveh->fwholesale;

Один из операторов программы предназначен для запроса комментария пользователя о каждом катере. В операторе ввода C++, необходимом для считывания строки комментария, используется другой метод ввода/вывода. Напомним, что поток cin считывает символьную информацию до первого разделительного символа. В нашем случае таким разделителем служит символ пробела между словами в комментарии. Если использовать cin, то только первое слово строки комментария можно запомнить в элементе структуры szcomment. Поэтому для ввода целой строки текста используется модифицированный вид потока cin:

cout << "\nPlease enter a one line comment about the boat: ";

cin.get(newline); // обрабатывается возврат каретки

cin.get(pastNineveh->szcomment,80, '.') ;

cin.get(newline); // обрабатывается возврат каретки

Сначала используется вызов cin.get(newline); он аналогичен вызову функции flushall() в предыдущих программах на С. В системах, где запросы от клавиатуры буферизируются, зачастую необходимо удалять символ перевода строки из входного буфера. Для этого существуют, конечно, и другие способы, однако они не более эффективны. Оператор cin.get(newline) считывает символ перевода строки и запоминает его в переменной newline, которая служит только для сбора информации и в программе фактически не используется. Строка комментария вводится при помощи следующего оператора:

cin.get(pastNineveh->szcomment,80,'.');

В этом операторе используется указатель на член структуры szcomment, затем указывается максимальная длина комментария (80) и следует символ окончания (.). В этом случае строка комментария заканчивается после ввода (n-1) или (80-1) символов или при вводе точки (последняя, n-ая позиция отводится под символ окончания строки null, \0). Символ точки не запоминается как часть комментария, поэтому он добавляется уже после вывода комментария. Найдите оператор, выполняющий эти действия.

### Функция – член структуры

При описании параметров некоторых объектов, например, человека, достаточно занести только конкретное значение, например, имя, фамилия, рост и так далее. Но некоторые параметры требуют выполнения процедур на момент их запроса, как-то: давление, температура и так далее.

Поэтому, как правило, описав новую структуру, приходится создавать набор функций, работающих с элементами этой структурой. Например, можно описать структуру трехмерной точки в декартовой системе или вектор. Пример.

struct **\_3d {**

double **x, y, z;**

} vect;

Было бы логично написать функцию, которая будет вычислять модуль вектора (длину). Пример.

double **mod (\_3d &V);**

Это традиционный путь для программ как на языке С так и других. Описывается функция, вычисляющая модуль вектора, который передается ей через формальные параметры. Между тем, модуль — характеристика, присущая каждому вектору. Поэтому, будет логично, совместить описание вектора и его длину, то есть вектор будет, как бы, сам возвращать свою длину.

Для того чтобы функция стала членом структуры, достаточно поместить ее описание внутрь описания структуры. Пример.

struct **\_3d {**

double **x, y, z;**

double **mod ();**

} vect;

Здесь функция mod () выступает как член структуры.

При описании реализации функции надо после типа возвращаемого значения указать имя шаблона структуры, членом которого является данная функция, и имя функции, отделив их знаком "::" (два двоеточия). Пример.

double **\_3d::mod()**

{

return **sqrt (x \* x + y \* y + z \* z);**

}

Если текст функции короткий, то реализацию функции можно поместить непосредственно внутрь описания структуры.

Пример.

struct **\_3d {**

double **х, у, z;**

double **mod () {** return **sqrt (x \* x + у \* у + z \* z); }**

} vect;

В этом случае можно опустить имя структуры, а сама функция будет считаться **inline**.

В качестве преимуществ такой структуры отметим, что такая функция обращается к своим переменным (членам структуры) запросто по имени. То есть, если функция не использует посторонние данные (не члены структуры) для достижения результата и не возвращает результат через список параметров, то список параметров может быть пустым.

Помимо этого необходимо отметить, что каждая функция шаблона структуры представлена в единственном экземпляре и получает один скрытый параметр — указатель на ту переменную структуру, для которой она вызвана. К этому указателю можно обратиться непосредственно по его имени – this.

Обращаться к функциям-членам структуры (класса) следует также как и к переменным: указать имя структуры и функции через точку; указать имя указателя на структуру и функции через знак «->».

Пример

**\_3d a, \*b;**

double **lenA, lenB, lenAB;**

**b =** new **\_3d;**

**a.x = 1.;**

**…**

**lenA = a.mod();**

…

**b->x = 3.;**

**…**

**lenB = b->mod();**

**…**

Полный пример, который показывает, как использовать поля и функции переменной типа структуры и указателя на структуру

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

struct **\_3d {**

double **x, y, z;**

double **mod ();**

} vect;

struct **\_3d {**

double **х, у, z;**

double **mod () {** return **sqrt (x \* x + у \* у + z \* z); }**

} vect;

double **\_3d::mod()**

{

return **sqrt (x \* x + y \* y + z \* z);**

}

int **main()**

{

**\_3d a, \*b;**

double **lenA, lenB, lenAB;**

**b =** new **\_3d;**

**a.x = 1.;**

**a.y = 2.;**

**a.z = 3.;**

**lenA = a.mod();**

**printf(**"%f\n",a.x);

**printf(**"%f\n",a.y);

**printf(**"%f\n",a.z);

**printf(**"%f\n\n",lenA);

**b->x = 3.;**

**b->y = 2.;**

**b->z = 1.;**

**lenB = b->mod();**

**printf(**"%f\n",b->x);

**printf(**"%f\n",b->y);

**printf(**"%f\n",b->z);

**printf(**"%f\n\n",lenB);

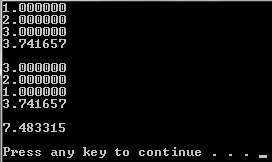
**lenAB = a.mod() + b->mod();**

**printf(**"%f\n\n",lenAB);

**system(**"PAUSE");

return **0;**

}



Особенность возврата const.

**Замечание**

1. Расширение возможностей и/или обход установок выполняемых const можно осуществить с помощью ключевого слова mutable (тема «*Классы С++*» в ООП).

2. Полезно ознакомиться с вопросом «*Сложные формы данных» этой темы*

Немного теории. Прежде всего учитываем то, что

int\* - pointer на int

int const \* - pointer на const int

int \* const - const pointer на int

int const \* const - const pointer на const int

Теперь первый const может быть по обе стороны от типа:

const int \* == int const \*

const int \* const == int const \* const

Если вы хотите сойти с ума, вы можете делать такое:

int \*\* - pointer на pointer на int

int \*\* const - const pointer на a pointer на int

int \* const \* - pointer на const pointer на int

int const \*\* - pointer на pointer на const int

int \* const \* const - const pointer на const pointer на int

...

И чтобы убедиться, что мы ясно понимаем значение const

constint\* foo;

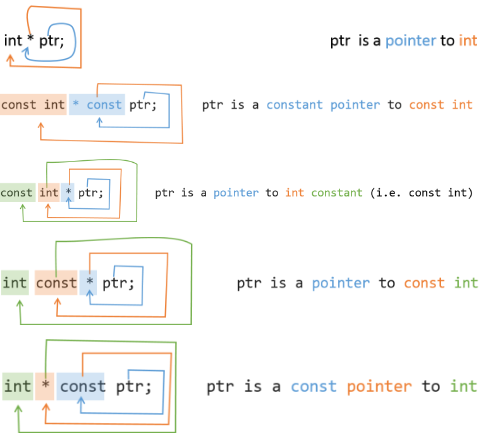
int **\***const **bar;** //обратите внимание, вам действительно нужно установить указатель

//здесь, потому что вы не можете изменить его позже

**foo** - переменный указатель на константу int. Это позволяет вам изменить то, на что вы указываете, но не значение, на которое вы указываете. Чаще всего это наблюдается для cstrings, где у вас есть указатель на const char. Вы можете изменить строку, на которую указываете, но вы не можете изменить содержимое этих строк. Это важно, когда сама строка находится в сегменте данных программы и не должна изменяться.

**bar** является константой или фиксированным указателем на значение, которое можно изменить. Это похоже на ссылку без дополнительного синтаксического излишества. Из-за этого, как правило, вы будете использовать ссылку, где вы будете использовать указатель T \* const, если вам не нужно разрешать нулевые указатели.

Вот визуальная подсказка



1. **Постоянная ссылка:**

Ссылка на переменную (здесь int), которая является постоянной. Мы передаем переменную как ссылку главным образом, потому что ссылки меньше по размеру, чем фактическое значение, но есть побочный эффект, и это потому, что он похож на псевдоним на фактическую переменную. Мы можем случайно изменить основную переменную через наш полный доступ к псевдониму, поэтому мы делаем его постоянным, чтобы предотвратить этот побочный эффект.

int **var0 = 0;**

constint **&ptr1 = var0;**

ptr1 = 8; // Error

var0 = 6; // OK

1. **Константные указатели**

Как только постоянный указатель указывает на переменную, он не может указывать на какую-либо другую переменную.

int **var1 = 1;**

int **var2 = 0;**

int **\***const **ptr2 = &var1;**

ptr2 = &var2; // Error

1. **Указатель на константу**

Указатель, через который нельзя изменить значение переменной, на которую он указывает, называется указателем на константу.

intconst **\* ptr3 = &var2;**

\*ptr3 = 4; // Error

1. **Постоянный указатель на константу**

Постоянным указателем на константу является указатель, который не может ни изменить адрес, на который он указывает, ни изменить значение, сохраненное на этом адресе, и не изменять его.

int **var3 = 0;**

int **var4 = 0;**

constint **\*** const **ptr4 = &var3;**

\*ptr4 = 1; // Error

**ptr4 = &var4;** // Error

С учетом теории, очевидными становятся примеры кода

**Пример1**

#include<iostream>

usingnamespace **std;**

int **main() {**

int **a,b;**

int\* w; // read/write int, read/write pointer

**w= &b;** // OK

**\*w= 1;** // OK

int\* const **x = &a;** // read only pointer, read/write int

// x = &b; // compilation error

**\*x = 0;** // OK

intconst **\* y;** // read/write ptr, read only int

constint **\* y2;** // " " "

**y = &a;** // OK

// \*y = 0; // compilation error

**y2 = &a;** // OK

// \*y2 = 0; // compilation error

intconst **\*** const **z = &a;** // read only ptr and read only int

constint **\*** const **z2 = &b;** // " " " "

// \*z = 0; // compilation error

// z = &a; // compilation error

// \*z2 = 0; // compilation error

// z2 = &a; // compilation error

}

**Пример2**

#include<iostream>

usingnamespace **std;**

class **A**

{

public:

**A():member\_()**

**{**

**}**

int **hashGetter()** const

**{**

**state\_ = 1;**

return **member\_;**

**}**

int **goodGetter()** const

**{**

return **member\_;**

**}**

int **getter()** const

**{**

//member\_ = 2; // error

return **member\_;**

**}**

int **badGetter()**

**{**

return **member\_;**

**}**

private:

mutableint **state\_;**

int **member\_;**

};

int **main()**

{

const **A a1;**

//a1.badGetter(); // doesn't work

**a1.goodGetter();** // OK

**a1.hashGetter();** // OK

**A a2;**

**a2.badGetter();** // OK

**a2.goodGetter();** // OK

**a2.hashGetter();** // OK

**system(**"PAUSE");

}

**Пример3**

#include<iostream>

usingnamespace **std;**

int **main()**

{

int **i1 = 0;**

int **i2 = 10;**

constint **\*p = &i1;**

intconst **\*p2 = &i1;**

constintconst **\*p3 = &i1;**

**p = &i2;**

**p2 = &i2;**

**p3 = &i2;**

**cout << \*p << endl**

**<< \*p2 <<endl**

**<< \*p3 <<endl;**

**system(**"PAUSE");

return **0;**

}



Также имеет значение различия возвращаемых значений типа класса: если вы возвращаете T const, то вызывающий не может вызывать неконстантные функции по возвращаемому значению, например:

class **Test**

{

public:

void **f();**

void **g()** const;

};

Test ff();

Test const **gg();**

ff().f(); // legal

ff().g(); // legal

gg().f(); // \*\*illegal\*\*

gg().g(); // legal

Когда вы добавляете ключевое слово const в метод, то этот указатель будет по существу const, и поэтому вы не можете изменять какие-либо данные элемента. (Если вы не используете mutable, более подробно об этом позже).

Ключевое слово const является частью сигнатуры функций, что означает, что вы можете реализовать два похожих метода: тот, который вызывается, когда объект const, а другой - нет.

#include<iostream>

usingnamespace **std;**

class **MyClass**

{

private:

int **counter;**

public:

void **Foo()**

**{**

**cout <<** "Foo" **<< endl;**

**}**

void **Foo()** const

**{**

**cout <<** "Foo const" **<< endl;**

**}**

};

int **main()**

{

**MyClass cc;**

const **MyClass& ccc = cc;**

**cc.Foo();**

**ccc.Foo();**

**system(**"PAUSE");

}



В методе non-const вы можете изменить члены экземпляра, что вы не можете сделать в версии const. Если вы измените объявление метода в приведенном выше примере на код ниже, вы получите некоторые ошибки.

void **Foo()**

{

**counter++;** //this works

**cout <<** "Foo" **<< endl;**

}

void **Foo()** const

{

**counter++;** //this will not compile

**cout <<** "Foo const" **<< endl;**

}

Это не совсем так, потому что вы можете пометить элемент как измененный, а метод const затем может его изменить. Он в основном используется для внутренних счетчиков и прочее. Решением для этого будет приведенный ниже код.

#include<iostream>

usingnamespace **std;**

class **MyClass**

{

private:

mutableint **counter;**

public:

**MyClass() : counter(0) {}**

void **Foo()**

**{**

**counter++;**

**cout <<** "Foo" **<< endl;**

**}**

void **Foo()** const

**{**

**counter++;**

**cout <<** "Foo const" **<< endl;**

**}**

int **GetInvocations()** const

**{**

return **counter;**

**}**

};

int **main(**void)

{

**MyClass cc;**

const **MyClass& ccc = cc;**

**cc.Foo();**

**ccc.Foo();**

//Вызывается экземпляр MyClass

**cout <<** "The MyClass instance has been invoked "

**<< ccc.GetInvocations() <<** " times" **<< endl;**

**system(**"PAUSE");

}



Становится очевидным, если вы считаете, что компилятор будет генерировать конструктор по умолчанию, конструктор копии по умолчанию и стандартный оператор присваивания копии для вас, если ваш struct/class не содержит ссылочные элементы. Затем подумайте, что стандарт позволяет вам вызывать методы-члены во временных рядах, то есть вы можете вызывать неконстантные члены в неконстантных временных рядах.

См. этот пример:

#include<iostream>

usingnamespace **std;**

struct **MyS {**

int **fn1()** const **{** return **0xFEED; }** // 65261

int **fn2() {** return **0xFEEA; }** // 65258

};

//mys это моя функция типа const MyS

const **MyS mys () {**

return **MyS();** //запускается конструктор MyS() и возвращается объект MyS

}

int **main () {**

**cout <<** "mys().fn1()=" **<< mys().fn1() <<** "\n"; // okay, called const member method

//cout << "mys().fn2()=" << mys().fn2(); // error, called non-const member of const temporary

**system(**"PAUSE");

return **0;**

}



**Пример**

#include<stdio.h>

#include<stdlib.h>

typedefstruct **Kernel {**

int **a;**

} Kernel;

Kernel\* kernel\_new() {

return **(Kernel\*) malloc(**sizeof(Kernel));

}

void **kernel\_delete(Kernel\* kernel) {**

**free(kernel);**

}

void **kernel\_start(Kernel\* kernel) {**

**printf(**"%d\n", kernel->a);

}

int **main(**int **argc,** char\* argv[]) {

**Kernel\* kernel = kernel\_new();**

**kernel->a = 10;**

**kernel\_start(kernel);**

**kernel\_delete(kernel);**

**system(**"PAUSE");

return **0;**

}



### Возврат структуры из функции

https://dev64.wordpress.com/2012/10/11/object-return-in-cpp/

Существует легенда, что возврат структуры по значению из функции в C++ — затратная процедура.

#include<stdio.h>

#include<string.h>

#include<iostream>

struct **Object {**

public:

int **a;**

int **b;**

int **c;**

char **string[50];**

};

Object func() {

**Object o;**

**o.a = 1;**

**o.b = 2;**

**o.c = 3;**

**strcpy(o.string,** "Hello world!");

return **o;**

}

int **main() {**

**Object o = func();**

**printf(**"%d\n",o.a);

**printf(**"%d\n",o.b);

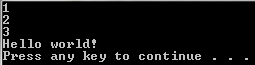
**printf(**"%d\n",o.c);

**printf(**"%s\n",o.string);

**system(**"PAUSE");

return **0;**

}



Что отсюда видно и что следует из ассеблерного кода. Видно что в функции main() резервируется место в стеке под структуру Object o. А потом, как это не странно ссылка на эту выделенную память передается внутрь функции func(). И функция func не производя никакого выделения памяти самостоятельно, работает с полями структуры, предварительно загрузив адрес структуры в регистр eax.

Т.е. это означает, что никаких накладных расходов данный фрагмент не вызывает! Во первых нет ни одной операцией с «кучей». Всё выделяется в стеке вызывающей функции. Во-вторых адрес выделенной памяти передается скрытой ссылкой или указателем. Это как вам больше нравится. В ассеблере просто передается адрес.

Теперь проверим следующий, закономерный вопрос. А что если функция func сама является членом другого класса. Как компилятор поступит в этом случае. Провожу для этого второй эксперимент, аналогичный первому:

#include<stdio.h>

#include<string.h>

#include<iostream>

struct **Object {**

public:

int **a;**

int **b;**

int **c;**

char **s[50];**

};

class **A {**

int **a;**

public:

**Object func();**

};

Object A::func() {

**Object o;**

**o.a=1;**

**o.b=2;**

**o.c=3;**

**strcpy(o.s,**"Hello world");

**a=5;**

return **o;**

}

int **main(**int **argc,** char **\*\* argv) {**

**A a;**

**Object o = a.func();**

**printf(**"%d\n",o.a);

**printf(**"%d\n",o.b);

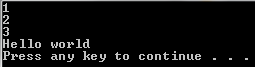
**printf(**"%d\n",o.c);

**printf(**"%s\n",o.s);

**system(**"PAUSE");

return **0;**

}



Этот код показывает, что опять в функцию func() в качестве первого аргумента передается указатель на структуру Object o. Функция в данном случае загружает его адрес в регистр ebx и заполняет поля структуры. Затем загружает в регистр eax второй скрытый параметр, содержащий указатель на экземпляр класса, которому принадлежит функция, т.е. this.

Т.о. вывод при возврате структуры по значению никаких накладных расходов на его выделение не случается. Память выделяется в стеке вызывающей функции. В функцию передается скрытый указатель на выделенную память. В случае, если функция, возвращающая структуру является членом класса, тогда в нее передается не один, а два указателя. Один указатель на выделенную для возвращаемого значения память и второй указатель this (на экземпляр объекта класса, членом которого является функция). Функция по окончании работы в регистре eax возвращает указатель на объект, тот же, что ей передали в скрытом параметре. Это уже лишнее действие по сути. Но присвоение регистра — это не сильно затратная операция.

У вышеописанного поведения есть ограничения. У функции func() должен быть только один return, и др. См. статью. **Если объект будет сложным (у его мемберов будут деструкторы или у объекта будет деструктор), тогда произойдет вызов оператора копирования и вызов деструктора. И операция обойдется дороже.** Однако, если возвращается структура содержащая простые типы без деструкторов, тогда копирования делаться не будет. Из чтения статьи на wiki по описанному вопросу можно почерпнуть дополнительную информацию.

**Еще один пример.**

Очень часто требуется писать функции, которые принимают структуры в качестве аргумента или возвращают структуру. Например, если вам надо написать небольшую космическую армаду, вам может понадобится функция для инициализации нового противника:

struct **EnemySpaceShip {**

int **x\_coordinate;**

int **y\_coordinate;**

int **weapon\_power;**

};

EnemySpaceShip getNewEnemy();

Функция getNewEnemy должна возвращать структуру с инициализированными полями:

EnemySpaceShip getNewEnemy() {

**EnemySpaceShip ship;**

**ship.x\_coordinate = 0;**

**ship.y\_coordinate = 0;**

**ship.weapon\_power = 20;**

return **ship;**

}

На самом деле эта функция вернет копию созданной локальной переменной ship. Это значит, что каждое поле структуры будет скопировано в новую переменную. В нашем случае копирование малого количества полей не заметно, но когда вы работаете с большими объемами данных нужно избегать лишних действий, подробнее об этом поговорим в статье про указатели.

Таким образом, для получения новой переменной будем использовать следующий код:

**EnemySpaceShip enemy = getNewEnemy();**

Теперь эту переменную можно использовать как обычную структуру.

Передавать структуры в функцию можно так:

EnemySpaceShip upgradeWeapons(EnemySpaceShip ship) {

**ship.weapon\_power += 10;**

return **ship;**

}

Когда мы передаем структуру в функцию, она копируется, так же как и при возвращении структуры. Поэтому любые изменения сделанные внутри функции будут потеряны, поэтому мы возвращаем структуру после изменения.

Использование функции:

ship = upgradeWeapons(ship);

Когда вызывается функция, переменная ship копируется и изменяется в функции, а когда переменная возвращается, она снова копируется и перезаписывает поля оргинальной переменной.

И наконец, программа для создания и улучшения одного корабля:

#include<stdio.h>

#include<iostream>

struct **EnemySpaceShip {**

int **x\_coordinate;**

int **y\_coordinate;**

int **weapon\_power;**

};

EnemySpaceShip getNewEnemy() {

**EnemySpaceShip ship;**

**ship.x\_coordinate = 0;**

**ship.y\_coordinate = 0;**

**ship.weapon\_power = 20;**

return **ship;**

}

EnemySpaceShip upgradeWeapons(EnemySpaceShip ship) {

**ship.weapon\_power += 10;**

return **ship;**

}

int **main() {**

**EnemySpaceShip enemy = getNewEnemy();**

**printf(**"%d\n", enemy.x\_coordinate);

**printf(**"%d\n", enemy.y\_coordinate);

**printf(**"%d\n\n", enemy.weapon\_power);

**enemy = upgradeWeapons(enemy);**

**printf(**"%d\n", enemy.x\_coordinate);

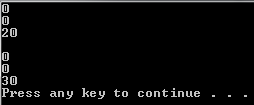
**printf(**"%d\n", enemy.y\_coordinate);

**printf(**"%d\n", enemy.weapon\_power);

**system(**"PAUSE");

return **0;**

}



## Структуры и битовые поля.

При описании полей структуры в явном виде можно указывать размер полей. Минимальный размер поля структуры – один бит. Общий синтаксис объявления структуры с явным указанием размеров полей имеет вид:

struct имя{

тип\_поля1 имя\_поля1: размер1;

тип\_поля2 имя\_поля2: размер2;

...

тип\_поляN имя\_поляN: размерN;

};

При этом для части полей можно указывать размер, а для других – нет. Если поле имеет размер в один бит, в качестве его типа указывается unsigned (у числа, реализованного с помощью одного бита, не может быть знака). Пример использования структуры с явно указанным размером полей приведен в листинге

#include <iostream>

using namespace std;

struct BitFields{

unsigned int state:1;

int n:2;

int m;

} str;

int main(){

cout<<"Enter a number: ";

cin>>str.m;

str.state=str.m%2;

str.n=str.m%4-2;

cout<<"state = "<<str.state<<endl;

cout<<"n ="<<str.n<<endl;

getchar();getchar();

return 0;

}



У структуры BitFields три поля: однобитовое целочисленное поле state, двухбитовое целочисленное поле n и целочисленное поле m (размер поля не указан).

Поле state может принимать всего два значения: 0 или 1. Поле n принимает целочисленные значения в диапазоне от -2 до 1 включительно, т.е. всего 4 возможных значения: напомним, что старший бит определяет знак числа, поэтому двухбитовое число 11 соответствует числу -2, двухбитовое число 10 соответствует числу -1, число 00 соответствует нулю и число 01 соответствует числу 1.

В главном методе программы выводится запрос на ввод пользователем целого числа. Это число записывается в поле m переменной str структуры BitFields. В переменную state заносится остаток от деления этого числа на 2, а в переменную n записывается остаток от деления введенного пользователем числа на 4 минус 2. Значения полей переменной str структуры BitFields выводятся на экран. В результате можем получить нечто наподобие следующего:

Enter a number: 9

state = 1

n = -1

Таким образом, путем явного указания размеров полей структуры удалось добиться экономии системных ресурсов: для записи значений используется минимально необходимое количество бит. Возникает естественный вопрос: а что будет, если присваиваемое битовому полю (т.е. полю, для которого явно указан размер) значение, выходит за допустимые для этого поля пределы? Например, что будет, если полю n значение присваивать с по мощью команды str.n=str.m%4-4? Ответ такой: в указанных случаях происходит автоматическое отбрасывание лишних бит. В частности, если поле m равно 9, а поле n определяется как str.n=str.m%4-4, то этому полю будет присвоено значение 1. Поясним это. Так, остаток от деления 9 на 4 есть 1. Если отнять 4, получим -3. В двоичном представлении число -3 имеет вид (последние три бита, все старшие биты равны 1) 101. Старшие биты отбрасываются, и в результате в двухбитовом представлении получаем 01, что соответствует числу 1.

В С и C++ имеется возможность доступа к отдельным разрядам (битам) в некотором более крупном типе данных, например, в байте. Это полезно, к примеру, для изменения масок, используемых для системных целей и графики. Возможность доступа к разрядам строится на основе структур С и C++.

Рассмотрим для примера информацию, необходимую для изменения регистра статуса клавиатуры компьютера. Этот регистр в компьютерах IBM содержит следующую информацию:

биты регистра

Статус клавиатуры: 76543210

Порт(417h)

где

bit 0 = RIGHT SHIFT не нажат (1)

bit 1 = LEFT SHIFT не нажат (1)

bit 2 = CTRL не нажат (1)

bit 3 = ALT не нажат (1)

bit 4 = SCROLL LOCK активен (1)

bit 5 = NUM LOCK активен (1)

bit 6 = CAPS LOCK активен (1)

bit 7 = INS активен (1)

Для доступа и управления этими данными можно создать структуру следующего вида:

struct stkeybits {

unsigned char

ucrshift: 1, /\* младший разряд \*/

uclshift: 1,

ucctrl: 1,

ucalt: 1,

ucscroll: 1,

ucnumlock: 1,

uccaplock: 1,

ucinsert: 1; /\* старший разряд \*/

} stkey\_register;

Биты в структуре начинаются с младшего и заканчиваются старшим. Можно указывать несколько битов, задавая их количество (вместо 1). В битовых полях можно использовать только целочисленные типы.

Обращение к элементам структуры битовых полей выполняется как обычно.

## Структуры. Их дальнейшее использование.

Мы не будем больше рассказывать о структурах, но хотелось бы отметить одно очень важное использование структур: создание новых типов данных. Пользователи компьютеров разработали новые типы данных, гораздо более эффективные для определенных задач, чем массивы и простые структуры, которые мы описали.

Эти типы имеют такие названия, как очереди, двоичные деревья, неупорядоченные массивы, рандомизированные таблицы и графы. Многие из этих типов создаются из «связанных» структур. Обычно каждая структура будет содержать один или два типа данных плюс один или два указателя на другие структуры такого же типа. Указатели служат для связи одной структуры с другой и для обеспечения пути, позволяющего вам вести поиск по всей структуре. Например, на рисунке показано двоичное дерево, в котором каждая отдельная структура (или «узел») связана с двумя, расположенными ниже.

15_03

Является ли эта разветвленная конструкция более эффективной чем массив? Рассмотрим случай дерева с 10 уровнями узлов. Если вы составите его, то найдете 1023 узла, в которых вы можете запомнить, скажем, 1023 слова. Если слова упорядочены, согласно некоторому разумному плану, вы можете начать с верхнего уровня и находить любое слово в лучшем случае за 9 перемещений, если ваш поиск идет сверху вниз с одного уровня на следующий. Если слова находятся в массиве, вам, может быть, придется перебрать все 1023 элемента, прежде чем вы найдете нужное слово.

Когда вас интересуют более прогрессивные структуры данных, обратитесь к литературе по вычислительной технике. Используя структуры языка Си, вы сможете создавать типы, о которых вы прочитали.

Это наше последнее слово о структурах. Далее мы хотим вкратце ознакомить вас с двумя другими средствами языка Си для работы с данными: объединением и функцией typedef.

## Объединения.

Объединение — еще один тип данных, который можно использовать различными способами. К примеру, некоторое объединение может рассматриваться как целое значение при выполнении одной операции и как число с плавающей точкой или двойной точности — при выполнении другой. По виду объединения напоминают структуры; однако, они сильно отличаются. Объединение, так же как и структура, может содержать несколько различных типов данных. Однако в объединении эти данные занимают одну и ту же область памяти! Таким образом, в отдельный момент времени объединение может хранить информацию только об одном типе данных. Во многих других языках высокого уровня данное языковое средство называется "записью с вариантами" (variant record).

### Объединения: синтаксис и правила.

В C++ имеется особая конструкция, объявление которой напоминает синтаксис объявления структуры, но имеющая совершенно другой физический смысл. Речь идет об объединениях (иногда называются союзами).

Объединение — это средство, позволяющее запоминать данные различных типов в одном и том же месте памяти. Типичным использованием его было бы создание таблицы, содержащей смесь типов в некотором порядке, который не является постоянным и не известен заранее. Объединение позволяет создавать массив, состоящий из элементов одинакового размера, каждый из которых может содержать различные типы данных.

Объединения устанавливаются таким же способом, как и структуры. Есть шаблон объединения и переменные объединения. Они могут определяться одновременно или, если используется имя объединения, последовательно за два шага.

Объединение служит для размещения в одной и той же области памяти (по одному и тому же адресу) данных различных типов. Очевидно, что в отдельный момент времени в памяти может находиться только один из указанных при объявлении объединения типов.

Объединение создается при помощи ключевого слова union и с использованием следующего синтаксиса:

union поле\_тега {

тип поле1;

тип поле2;

тип поле3;

...

…

…

тип полеN;

};

Объявление объединения начинается с ключевого слова union, за которым следует идентификатор и блок описания элементов различного типа, например:

union MyData

{

int iVar1;

unsigned long ulFreq;

char Symb[10];

}

При этом в памяти компилятором резервируется место для размещения самого большого элемента объединения, в данном случае - 10 байт под символьный массив. Чаше всего объединения включают в состав структур, а также служат для преобразования данных одного типа в другой. Доступ к элементам объединения производится с помощью операции 'точка' (.). Ниже приводится пример такого объединения.

**В отличие от структур, переменная типа объединения может быть проинициализирована только значением первого объявленного члена** (в данном случае - целочисленной переменной iVar1):

union MyData

{

int iVar1;

unsigned long ulFreq;

char Symb[10];

} myX = {25};

Обратите внимание на сходство синтаксисов объявлений структур и объединений в следующем примере:

union unmany\_types {

char с;

int ivalue;

float fvalue;

double dvalue;

} unmy\_union;

Объединение описывается при помощи ключевого слова union, за которым следует необязательное поле тега unmany\_types. Поле тега объединения используется так же, как поле тега структуры. Данное объединение содержит несколько элементов: символ, целое число, число с плавающей точкой и число двойной точности. В отдельный момент времени это объединение позволяет хранить в unmany\_types информацию только об одном типе данных.

С объединением связана переменная unmy\_union. Если этот оператор расположен в функции, то область действия данного объединения является локальной для функции. Если оператор расположен за пределами всех функций, то область действия объединения — глобальная.

Подобно структурам, с одним объединением можно связать несколько переменных; можно также обращаться к членам объединения при помощи операции "точка" (.). Синтаксис очень простой:

unname.mname

В этой записи unname является переменной, связанной с типом объединения, а mname — имя любого элемента объединения.

Вот пример шаблона с именем объединения:

union holders {

int digit;

double bigfl;

char letter;

};

А вот пример определения переменных объединения типа holdem:

union holdem fit; /\* переменная объединения типа holdem \*/

union holdem save[10]; /\* массив из 10 переменных объединения \*/

union holdem \*pu; /\* указатель на переменную типа holdem \*/

Первое описание создает одну переменную fit. Компилятор выделяет достаточно памяти для размещения самой большой из описанных переменных. В этом случае наибольшей из возможных является переменная double, для которой требуется в нашей системе 64 разряда или 8 байтов. Массив save имел бы 10 элементов, каждый по 8 байтов.

Вот как используется объединение:

fit.digit = 23; /\* 23 записывается в fit; используется 2 байта \*/

fit.double = 2.0; /\* 23 стирается, 2.0 записывается; используется 8 байтов \*/

fit.letter = 'h' ; /\* 2.0 стирается, h записывается; используется 1 байт \*/

Вы применяете операцию получения элемента, чтобы показать, какие типы данных используются. В каждый момент времени запоминается только одно значение; нельзя записать char и int одновременно, даже если для этого достаточно памяти.

Вы сами должны следить за типом данных, записываемых в данный момент в объединение; приведенная ниже последовательность операторов показывает, что нельзя делать:

fit.letter = 'A';

flnum = 3.02\*fit.double; /\* ОШИБКА ОШИБКА ОШИБКА \*/

Ошибка заключается в том, что записано значение типа char, a следующая строка предполагает, что содержимое fit имеет тип double.

Можно использовать операцию —> с объединениями таким же образом, как это делалось для структур:

pu = &fit;

х = pu->digit; /\* то же, что и х = fit.digit \*/

В качестве примера рассмотрим использование объединений и структур для неявного преобразования данных, представленных в десятичном формате, в двоично-десятичный код. Как известно, двоично-десятичные данные представляют собой разновидность упакованных значений, в которых для представления для представления десятичной цифры используется одна тетрада (четыре бита). Таким образом, двоично-десятичное число может принимать значения от 0х00 до 0х99 или в десятичном виде - от 0 до 153. Для решения поставленной задачи сконструируем объединение BCD, содержащее однобайтный член. В качестве другого элемента объединения создадим однобайтную структуру с применением битовых полей:

#include <iostream>

using namespace std;

union BCD

{

unsigned char data;

struct{

unsigned char lo : 4;

unsigned char hi : 4;

} bin;

} bed;

int main()

{

setlocale(LC\_ALL, "Rus");

bed.data = 152;

cout << "\nHigh: " << (int) bed.bin.hi;

cout << "\nLow : " << (int) bed.bin.lo;

getchar(); getchar();

return 0;

}

В теле функции main () производится инициализация двоично-десятичной переменной bed двоичным значением 152 с последующим ее разложением на старший и младший полубайт.

****

### Создание простого объединения.

Для того чтобы проиллюстрировать некоторые концепции объединений, в следующей программе на C++ создается объединение рассмотренного выше типа. Цель этого примера — показать, что объединение может содержать различные типы данных, но в конкретный момент времени оно содержит значение только для одного типа.

// Программа на C++, иллюстрирующая использование объединения.

// Создается объединение, содержащее несколько типов данных

#include <iostream.h>

union unmany\_types {

char c;

int ivalue;

float fvalue;

double dvalue, j;

} unmy\_union;

int main(void)

{

// valid I/O

unmy\_union.c='b';

cout << unmy\_union.c << "\n";

unmy\_union.ivalue=1990;

cout << unmy\_union.ivalue << "\n";

unmy\_union.fvalue=19.90;

cout << unmy\_union.fvalue << "\n";

unmy\_union.dvalue=987654.32E+13;

cout << unmy\_union.dvalue << "\n";

// неправильный ввод/вывод

cout << unmy\_union.c << "\n";

cout << unmy\_union. ivalue << "\n";

cout << unmy\_union.fvalue << "\n";

cout << unmy\_union.dvalue << "\n";

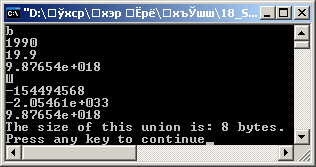
// размер объединения

cout << "The size of this union is: "<< sizeof(unmany\_types) <<" bytes." << "\n";

return (0);

}

Сначала программа просто вводит и распечатывает информацию из объединения. Эта часть работает, поскольку вызываемое объединение хранит в каждый момент времени только один тип данных. Однако во второй части программы делается попытка вывести каждый тип, существующий в объединении. Единственным верным значением будет число двойной точности, поскольку оно было последним значением, записанным в предыдущей части программы.



Объединения занимают память, необходимую для самого большого типа данных, хранящегося в них. Все другие типы данных в объединении используют совместно часть этой памяти или всю эту память.

При помощи встроенного отладчика вы можете посмотреть, что происходит с памятью внутри объединения.

## Вспомогательные средства.

Необходимо упомянуть еще две темы: объявления typedef и перечисляемые типы enum. При правильном использовании оба средства, typedef и enum, упрощают текст программы.

### Использование typedef.

Помимо структур и объединений разработчик программного обеспечения на C++ имеет возможность моделирования новых типов данных на базе уже имеющихся в его распоряжении типов. Формирование пользовательских типов осуществляется с использованием ключевого слова typedef, за которым указывается какой-либо из имеющихся типов данных и следующий за ним идентификатор, назначающий новое имя для выбранного типа. Фактически, таким образом, определяется синоним типа данных. Например, выражение

typedef unsigned char byte;

определяет новый тип данных byte, суть которого такая же, как и типа unsigned char. Следовательно, в программе возможно определение переменных типа byte, которые в памяти будут занимать один байт и смогут принимать значения от 0 до 255, например:

byte nlnput = 0xFF;

byte nOutput = 0;

Синтаксис объявления пользовательского типа для массива выглядит следующим образом:

typedef char Names[27];

В этом случае объявление переменной типа Names

Names name;

будет означать, что такая переменная представляет собой строку из 27-ми символов.

Наиболее часто переопределение типов данных используется совместно со структурами. Благодаря этому становится возможным создание новых типов данных, характерных для сложных объектов, объединяющих разнотипные характеристики. Например, создадим тип данных COORD, объекты которого будут нести информацию о трех координатах в пространстве:

typedef struct

{

double x;

double у;

double z;

} COORD;

Теперь можно объявить объект myPoint типа COORD:

COORD myPoint;

и обращаться к элементам так, если бы он был объявлен как соответствующая структура:

myPoint.x = 5.654;

myPoint.y = 0;

myPoint.z = 3.14;

При помощи typedef вы можете связать новые типы данных с существующими. К примеру, в математической программе может возникнуть необходимость в использовании типов данных fixed, whole, real или complex. Используя typedef, можно связать эти новые типы со стандартными типами С. В следующей программе создаются два новых типа:

/\* Программа на С, иллюстрирующая использование typedef.

Создаются два новых типа, "whole" и "real",

которые можно использовать вместо "int" и "float" \*/

#include <stdio.h>

typedef int whole;

typedef float real;

int main(void)

{

whole wvalue=123;

real rvalue=5.6789;

/\* Целое число \*/

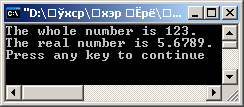
printf("The whole number is %d.\n",wvalue);

/\* Вещественное число \*/

printf("The real number is %.4f.\n",rvalue);

return (0);

}



**Будьте внимательны и не используйте слишком много новых типов; это может вызвать обратный эффект и ухудшить читаемость программы и усложнить ее.** Используйте typedef с осторожностью.

Объявление typedef можно использовать для упрощения объявлений. Взгляните на два следующих примера; сможете ли вы заметить небольшое отличие, вызванное применением ключевого слова typedef:

struct stboat {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int lmotor\_hours;

float fsaleprice;

} stused\_boat;

typedef struct {

char sztype [iSTRING15 + iNULL\_CHAR];

char szmodel[iSTRING15 + iNULL\_CHAR];

char sztitle[iSTRING20 + iNULL\_CHAR];

int iyear;

long int lmotor\_hours;

float fsaleprice;

} STBOAT;

Имеют место три главных отличия:

* Исчезло необязательное поле тега. (Хотя поле тега вместе с typedef можно использовать по-прежнему; просто оно избыточно.)
* Поле тега stboat превратилось в новый тип STBOAT и располагается там, где традиционно описывалась структурная переменная.
* Объявление переменной stused\_boat теперь отсутствует.

Достоинство описаний typedef — в способе их использования. После приведенного описания по всей программе можно описывать переменные типа STBOAT при помощи более простой синтаксической конструкции:

STBOAT STused\_boat;

Синтаксически, использование заглавных букв в компиляторах C/C++ не обязательно; они, однако, подчеркивают важное соглашение, принятое в программе. Во всех возможных файлах, где объявляются идентификаторы, программисты на С традиционно используют заглавные буквы для описаний новых типов, констант, перечисляемых значений и макросов, определяемых обычно в заголовочном файле. Видимое различие между ключевыми словами, записываемыми строчными буквами, и пользовательскими идентификаторами, записываемыми заглавными буквами, позволяет быстрее понимать текст программы, поскольку заглавные буквы обычно означают: "Ищите это объявление в другом файле".

Функция typedef позволяет нам создать свое собственное имя типа. Это напоминает директиву #define, но со следующими тремя изменениями:

1. В отличие от #define функция typedef дает символические имена, но ограничивается только типами данных.
2. Функция typedef выполняется компилятором, а не препроцессором.
3. В своих пределах функция typedef более гибка, чем #define.

Посмотрим, как она работает. Предположим, вы хотите использовать термин real для чисел типа float. Тогда вы определяете термин real, как если бы он был переменной типа float, и перед его определением ставите ключевое слово typedef:

typedef float real;

С этого момента вы можете использовать real для определения переменных:

real х, у[25], \*рг;

Область действия такого определения зависит от расположения оператора typedef. Если определение находится внутри функции, то область действия локальна и ограничена этой функцией. Если определение расположено вне функции, то область действия глобальна.

Часто в этих определениях используются прописные буквы, чтобы напомнить пользователю, что имя типа является на самом деле символической аббревиатурой:

typedef float REAL;

В последнем примере можно было бы применить директиву #define. А здесь это делать нельзя:

typedef char \*STRING;

Без ключевого слова typedef оператор определял бы STRING как указатель на тип char. С ключевым словом оператор делает STRING идентификатором указателей на тип char. Так,

STRING name, sign;

означает

char "name, \*sign;

Мы можем использовать typedef и для структур. Вот пример:

typedef struct COMPLEX {

float real;

float imag;

}

Кроме того, можно использовать тип COMPLEX для представления комплексных чисел.

Одна из причин использования typedef заключается в создании удобных, распознаваемых имен для часто встречающихся типов. Например, многие пользователи предпочитают применять STRING или его эквивалент, как это мы делали выше.

Вторая причина: имена typedef часто используются для сложных типов. Например, описание

typedef char \*FRPTC () [5];

приводит к тому, что FRPTC объявляет тип, являющийся функцией, которая возвращает указатель на пятиэлементный массив типа Char.

Третья причина использования typedef заключается в том, чтобы сделать программы более мобильными. Предположим, например, что вашей программе нужно использовать 16-разрядные числа. В некоторых системах это был бы тип short, в других же он может быть типом int. Если вы использовали в ваших описаниях short или int, то должны изменить все описания, когда перейдете от одной системы к другой. Вместо этого сделайте следующее. В файле директивы #include есть такое определение:

typedef short TWOBYTE;

Используйте TWOBYTE в ваших программах для переменных типа short, которые должны быть 16-разрядными. Тогда если вы перемещаете программу туда, где необходимо использовать тип int, то следует только изменить одно определение в вашем файле директивы #include:

typedef int TWOBYTE;

Это пример того, что делает язык Си столь мобильным. При использовании typedef следует иметь в виду, что он не создает новых типов, он только создает удобные метки.

### Использование enum.

Перечисляемый тип данных enum имеет единственное назначение — сделать текст программы более читаемым. В других компьютерных языках этот тип называется пользовательским, есть определяемым пользователем. В общем, объявление перечисляемого типа выглядит так:

enum поле\_тега { значение1,. . .значениеN } переменная;

Как вы уже, наверное, догадались — необязательное поле тега (op\_tag\_field) аналогично тому, что используется при объявлении структуры. Если это поле опускается, то после закрывающей фигурной скобки необходимо указать переменную (или переменные). При использовании поля тега можно объявлять в программе другие переменные такого же типа; при этом в C++ необязательно повторять ключевое слово enum.

Перечисляемые типы данных позволяют связать в единый тип легко понятные мнемонические идентификаторы: например, Понедельник, Вторник, Среда и так далее. Это помогает созданию самодокументирующегося текста программы. К примеру, вместо цикла, пробегающего значения от 0 до 4, можно написать цикл, имеющий диапазон от Понедельника до Пятницы (Monday — Friday):

enum eweekdays { Monday, Tuesday, Wednesday, Thursday, Friday };

/\* Объявление переменной типа enum в С \*/

enum eweekdays ewToday;

/\* To же объявление в C++ \*/

eweekdays ewToday;

/\* Без перечисляемого типа \*/

for(i = 0; i <= 4; i++)

…

…

…

/\* С использованием перечисляемого типа \*/

for(ewToday = Monday; ewToday <= Friday; ewToday++)

Как исторически сложилось, компиляторы С не делают различия между типами int и enum. Это означает, что программа может присвоить целое значение перечисляемому типу. При отсутствии явного приведения типа в C++ компилятор в подобной ситуации генерирует предупреждение:

/\* Допустимо в С; недопустимо в C++ \*/

ewToday = 1;

/\* Решение проблемы в C++ \*/

ewToday = (eweekdays)1;

Тип enum очень часто используется в программировании, когда информацию можно представить списком целых значений, подобных номерам месяцев в году или номерам дней недели. Такой список уже является перечислением.

В следующем примере имеется список всех месяцев в году. Они представляют собой перечисляемый список с тегом emonths. Co списком связана переменная emcompleted. Перечисляемые списки всегда начинаются с нуля (если не задано другого целого значения). В приведенном примере Январь (January) — первый месяц в году.

/\* Программа на С, иллюстрирующая использование перечисляемых типов.

Программа определяет количество прошедших и остающихся месяцев

в году при помощи типа enum \*/

#include <stdio.h>

enum emonths {

January=1,

February,

March,

April,

May,

June,

July,

August,

September,

October,

November,

December

} emcompleted;

int main(void)

{

int ipresent\_month;

int isum,idiff;

/\* Введите, пожалуйста, текущий месяц (от 1 до 12): \*/

printf("\nPlease enter the present month (1 to 12): ");

scanf("%d",&ipresent\_month);

emcompleted = December;

isum = ipresent\_month;

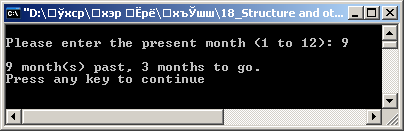
idiff = (int)emcompleted - ipresent\_month;

/\* ... месяцев прошло, ... месяцев остается. \*/

printf("\n%d month(s) past, %d months to go.\n",isum,idiff);

return (0);

}



В данной программе перечисляемый список — это список целых значений от 1 до 12. Поскольку имена эквивалентны последовательным целым значениям, то с ними можно выполнять целочисленные арифметические операции. Фактически, когда перечисляемой переменной emcompleted присваивается значение December, она становится равной 12.

Приведенная программа выполняет некоторые простые арифметические действия и выводит на экран следующий результат:

Please enter the current month (1 to 12) : 4

4 month(s) past, 8 months to go.

## Сложные формы данных.

Язык Си позволяет вам создавать сложные формы данных. Обычно мы придерживаемся более простых форм, но считаем своим долгом указать на потенциальные возможности языка. При создании описания мы используем имя (или «идентификатор»), которое можно изменять при помощи модификатора:

***модификатор значение***

\* указатель

() функция

[] массив

Язык Си позволяет использовать одновременно более одного модификатора, что дает возможность создавать множество типов:

int board[8] [8]; /\* массив массивов типа int \*/

int \*\*ptr; /\* указатель на указатель на тип int \*/

int \*risks[10]; /\* 10-элементный массив указателей на тип int \*/

int (\*wisks) [10]; /\* указатель на 10-элементный массив типа int \*/

int \*oof[3] [4]; /\* 3-элементный массив указателей на 4-элементный массив типа int \*/

int (\*uuf) [3] [4]; /\* указатель на массив 3x4 типа int \*/

Для распутывания этих описаний нужно понять, в каком порядке следует применять модификаторы. Три правила помогут вам справиться с этим.

1. Чем ближе модификатор стоит к идентификатору, тем выше его приоритет
2. Модификаторы [] и () имеют приоритет выше, чем \*.
3. Круглые скобки используются для объединения частей выражения, имеющих самый высокий приоритет.

Давайте применим эти правила к описанию

int \*oof[3] [4];

\* и [3] примыкают к oof и имеют более высокий приоритет, чем [4] (правило 1). [3] имеет приоритет более высокий, чем \* (правило 2). Следовательно, oof является 3-элементным массивом (первый модификатор) указателей (второй модификатор) на 4-элементный массив (третий модификатор) типа int (описание типа).

В описании

int (\*uuf)[3] [4];

скобки говорят, что модификатор \* должен иметь первый приоритет, а это делает uuf указателем, как показано в предыдущем описании.

Эти правила создают также следующие типы:

char \*fump(); /\* функция, возвращающая указатель на тип char \*/

char (\*frump)(); /\* указатель на функцию, возвращающую тип char \*/

char \*flump() [3] /\* функция, возвращающая указатель на 3-элементный массив типа char \*/

char \*flimp[3] () /\* 3-элементный массив указателей на функцию, которая возвращает тип char \*/

Если вы примените структуры к этим примерам, то увидите, что возможности для описаний действительно растут причудливо. А применение ... так и быть, мы оставим для более опытных программистов.

Язык Си со структурами, объединениями и typedef дает нам средства для эффективной и мобильной обработки данных.

По окончанию этого материала вы готовы приступить к изучению основ объектно-ориентированного программирования.

## Отличие stract от class

Отличий class от struct несколько (основных всего два):

* ***Member access control [class.access]***

Члены класса, определенного с помощью ключевого слова class, по умолчанию являются private. Члены класса, определенного с помощью ключевого слова struct или union, по умолчанию являются public.

C имеет структуры, у него нет понятия инкапсуляции, поэтому все открыто. Публикация по умолчанию обычно считается плохой идеей при использовании объектно-ориентированного подхода, поэтому при создании формы C, которая естественным образом способствует OOP (вы можете делать OO на C, но это вам не поможет) которая была идеей в С++ (первоначально "C With Classes" ), имеет смысл сделать члены закрытыми по умолчанию.

Например, программа 1 не работает в компиляции, и программа 2 работает нормально.

// Program 1

#include <stdio.h>

class Test {

int x; // x is private

};

int main()

{

Test t;

t.x = 20; // compiler error because x is private

getchar();

return 0;

}

//Run on IDE

// Program 2

#include <stdio.h>

struct Test {

int x; // x is public

};

int main()

{

Test t;

t.x = 20; // works fine because x is public

getchar();

return 0;

}

В дополнение еще один пример

class A{

public:

int i;

};

class A2:A{};

struct A3:A{};

struct abc{

int i;

};

struct abc2:abc{};

class abc3:abc{};

int main()

{

abc2 objabc;

objabc.i = 10;

A3 ob;

ob.i = 10;

//A2 obja; //privately inherited

//obja.i = 10;

//abc3 obss;

//obss.i = 10;

}

* ***Accessibility of base classes and base class members [class.access.base]***

При отсутствии спецификатора доступа (т.е. private/protected/public) у базового класса, базовый класс будет public если класс определен с помощью struct и private если класс определен с помощью class.

(Хотя union и является классом, он имеет много других отличий от class и struct.)

// Program 3

#include <stdio.h>

class Base {

public:

int x;

};

class Derived : Base { }; // is equilalent to class Derived : private Base {}

int main()

{

Derived d;

d.x = 20; // compiler error becuase inheritance is private

getchar();

return 0;

}

//Run on IDE

// Program 4

#include <stdio.h>

class Base {

public:

int x;

};

struct Derived : Base { }; // is equilalent to struct Derived : public Base {}

int main()

{

Derived d;

d.x = 20; // works fine becuase inheritance is public

getchar();

return 0;

}

* ***отличие между структурами и классами, связанное с инициализацией по умолчанию.***

struct Foo {

int a;

};

class Bar {

int a;

};

class Tester {

Foo m\_Foo = Foo();

Bar m\_Bar = Bar();

public:

Tester() {}

};

int main() {

auto myTester = Tester();

}

Запустите этот код и проверьте myTester. Вы обнаружите, что для m\_Foo структура, m\_Foo.a была инициализирована на 0, но для m\_Bar класс m\_Bar.a не инициализирован. Таким образом, существует разница в том, что делает конструктор по умолчанию для struct vs. class. Я вижу это с помощью Visual Studio.

**Итак**. Существует 3 основных различия между структурой и классом.

1St-память зарезервирована для структуры в стеке памяти (которая близка к языку программирования), независимо от того, зарезервированы ли для памяти класса в стеке только референт, а фактическая память зарезервирована в кучевой памяти.

2Nd. По умолчанию структура рассматривается как публикация, рассматриваемая классом как частная.

3Rd- не может повторно использовать код в структуре, но в классе мы можем повторно использовать один и тот же код за много времени, называемый inhertence

**Использование**

Есть популярное мнение, что для "расово правильных ООП классов" надо использовать только class, а для структур данных - только struct, и еще у struct не должно быть методов. Это не более чем вкусовщина, и подобные вещи должны быть закреплены в руководствах по стилю кодирования, как например это сделано в [Google C++ Style Guide](http://google-styleguide.googlecode.com/svn/trunk/cppguide.html#Structs_vs._Classes).

STRUCT - это тип абстрактного типа данных, который делит данный кусок памяти в соответствии со спецификацией структуры. Структуры особенно полезны при сериализации файлов/десериализации, поскольку структуру часто можно записать в файл дословно. (т.е. получить указатель на структуру, использовать макрос SIZE для вычисления количества байтов для копирования, а затем переместить данные в или из структуры.)

Классы - это другой тип абстрактных типов данных, которые пытаются обеспечить скрытие информации. Внутри могут быть различные махинации, методы, временные переменные, переменные состояния. и т.д., которые все используются для представления согласованного API для любого кода, который хочет использовать класс.

По сути, структуры относятся к данным, классы относятся к коду.

Однако вам нужно понять, что это просто абстракции. Совершенно возможно создавать структуры, которые очень похожи на классы и классы, которые очень похожи на структуры. На самом деле, самые ранние компиляторы С++ были просто предварительными компиляторами, которые переводили код С++ в C. Таким образом, эти абстракции являются преимуществом для логического мышления, не обязательно являющегося активом для самого компьютера.

Помимо того, что каждый из них представляет собой другой тип абстракции, классы обеспечивают решения головоломки с именованием кода C. Поскольку у вас не может быть более одной функции, открытой с тем же именем, разработчики использовали шаблон \_(). например mathlibextreme\_max(). Сгруппировав API в классы, подобные функции (здесь мы называем их "методами" ) могут быть сгруппированы вместе и защищены от именования методов в других классах. Это позволяет программисту лучше организовать свой код и увеличить повторное использование кода. Теоретически, по крайней мере.

## Функции работы с датой и временем.

Функции и типы данных, необходимые для работы с датой и временем объявлены в заголовочном файле time.h. В частности, этот файл содержит определения типа данных time\_t:

typedef long time\_t;

и структуры tm, которая объявлена следующим образом:

struct tm

{

int tm\_sec; // секунды

int tm\_min; // минуты

int tm\_hour; // часы (0-23)

int tm\_mday; // дата (1-31)

int tm mon; // месяц (0-11)

// год (текущий год минус 1900)

int tm\_year;

// день недели (0-6; Воскр = 0)

int tm wday;

int tm\_yday; // день в году (0-365)

int tm\_isdst; // 0 если зимнее время

}

Ниже приводится таблица 9.1, обобщающая часть объявленных функций с кратким описанием их работы.

**Таблица 9.1.**

**Функции работы с датой и временем**

|  |  |
| --- | --- |
| **Наименование** | **Краткое описание** |
| asctime | преобразует время и дату из формата структуры типа tm в символьную строку |
| clock | возвращает 'гасло "тиков" процессора, прошедших от начала запущенного процесса |
| ctime | преобразует время и дату из формата time t в символьную строку |
| difftime | вычисляет интервал между двумя заданными временными параметрами |
| gmtime | преобразует дату и время из формата time t в формат структуры tm по Гринвичу (GMT) |
| localtime | преобразует дату и время из формата time t в формат структуры tm |
| mktime | преобразует дату и время в календарный формат |
| strdate | преобразует текущую дату в символьную строку в формате mm/dd/yy |
| strftime | форматирует время для последующего вывода |
| strtime | возвращает текущее системное время в виде символьной строки |
| time | возвращает время в секундах, прошедшее с полуночи (0 часов 0 минут 0 секунд) 1 января 1970 г. по Гринвичу |
| tzset | устанавливает значения глобальных переменных daylight, \_timezone, и tzname |

В качестве обобщающего примера рассмотрим программу работы с датой и временем:

#include <time.h>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

time\_t tt;

tm \*pMyTime;

tt = time(NULL);

pMyTime = localtime(&tt);

cout << "Текущее время: ";

cout << asctime(pMyTime);

getchar(); getchar();

return 0;

}

В рассматриваемом примере используется функция time (), имеющая прототип

time\_t time(time\_t \*timer);



возвращающая число секунд, прошедших с 00:00:00 1 января 1970 г. Параметр timer также принимает возвращаемое функцией значение. Для более удобной работы с датой и временем полученные данные преобразуются функцией local time() к формату структуры tm. Применение функции asctime () позволяет вывести полученную таким образом информацию в виде строки символов.

**Что вы должны были узнать.**

* Что такое структурный шаблон, и как его определять
* Что такое имя структуры и как оно используется
* Как определить структурную переменную: struct car honda;
* Как обратиться к элементу структуры: honda.mpg
* Как обратиться к указателю на структуру: struct car\*ptcar;
* Как обратиться к элементу при помощи указателя: ptcar— >mpg
* Как передать в функцию элемент структуры: eval (honda .mpg)
* Как сообщить функции о структуре: rate (&honda)
* Как создать вложенную структуру
* Как обратиться к элементу вложенной структуры: honda .civic .cost
* Как создавать и использовать массивы структур: struct car gm [5];
* Как создать объединение: подобно структуре
* Как использовать typedef: typedef struct car CRATE;

**Вопросы и ответы.**

**Вопросы.**

1. Что неправильно в этом шаблоне?

structure {

char itible;

int num [20];

char \*togs;

};

1. Вот фрагмент программы; что она напечатает?

struct house {

float sqft;

int rooms;

int stories;

char \*address;

};

main ()

{

static struct house fruzt = {1560.0, 6, 1, "22 Spiffo Road"};

struct house \*sign;

sign = &fruzt;

printf (" %d %d\n" , fruzt.rooms, sign— >stories);

printf (" %s\n", fruzt.address);

printf (" %c %c \n" sign->address [3], fruzt.address [4]);

}

1. Придумайте структурный шаблон, который будет содержать название месяца, трехбуквенную аббревиатуру месяца, количество дней в месяце и номер месяца.
2. Определите массив, состоящий из двенадцати структур того же типа, что и в вопросе 3, и инициализируйте его для невисокосного года.
3. Напишите функцию, которая получает номер месяца, а возвращает общее число дней года вплоть до этого месяца. Считайте, что структурный шаблон и массив из вопросов 3 и 4 описаны как внешние.
4. Взяв за основу нижеследующую функцию typedef, опишите 10-элементный массив указанной структуры. Затем, используя присваивание отдельного элемента, попытайтесь описать третьим элементом массива линзу Ремаркатара с фокусным расстоянием 500 мм и апертурой f/2.0.

typedef struct { /\* описатель линзы \*/

float foclen; /\* фокусное расстояние, мм \*/

float fstop; /\* апертура \*/

char \*brand; /\* фирменная марка \*/

} LENS;

**Ответы.**

1. Должно быть ключевое слово struct, а не structure. Шаблон требует либо имени структуры перед открывающей скобкой или имени переменной после закрывающей скобки. Кроме того, точка с запятой должна стоять после \*togs и в конце шаблона.

6 1

22 Spiffo Road

S p

Элемент fruzt.address является символьной строкой, a fruzt.address [4] является пятым элементом этого массива.

struct month {

char name[10|; /\* или char \*name; \*/

char abbrev[4]; /\* или char\*abbrev; \*/

int days;

int monumb;

};

struct month months[12] = {

{ " Январь" , "Янв", 31, 1},

{ " Февраль", " Фев", 28, 2},

…

и т. д.

…

{ " Декабрь", "Дек", 31, 12}

};

days (month)

int month;

{

int index, total;

if (month < 1 || month > 12)

return (—1); /\* признак ошибки \*/

else

for (index = 0, total = 0; index < month; index++)

total + = months [index]. days; return (total);

}

Заметим, что index содержит номер месяца, уменьшенный на единицу, так как массивы начинаются с индекса 0; следовательно, мы используем выражение index < month вместо index <= month.

1. ЛИНЗА tubby [10];

tubby [2]. foclen = 300.0;

tubby [2]. fstop = 2.0;

tubby [2]. brand = " Ремаркатар" ;

**Упражнения.**

1. Переделайте вопрос 5, используя в качестве аргумента написанное буквами название месяца вместо номера месяца. [Не забывайте о функции strcmp().]
2. Напишите программу, которая запрашивает у пользователя день, месяц и год. Месяц может обозначаться номером, названием месяца или его аббревиатурой. После работы программа выдает общее количество дней в году вплоть до данного дня.
3. Переделайте нашу программу инвентаризации книг таким образом, чтобы она печатала информацию о книгах, упорядоченную в алфавитном порядке по названиям книг, и затем печатала общую стоимость книг.