ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

ИНФОРМАТИКА

част втора

Основи на програмирането на C++

лектор: гл. ас. д-р Стефан М. Панов

Катедра “Информатика”

Лекция 3 (11)

Структури и обединения в C++

Обявяване на структура

Структурата е средство, чрез което логически свързани променливи от различен тип могат да се обединят в едно цяло.

Променливите, принадлежащи на една структура, се наричат членове. По-рядко ги наричат полета или елементи.

Обявяването на структура става по следния начин:

```
struct име_на_структура
{
    тип име_на_променлива1;
    тип име_на_променлива2;
    . . .
    тип име_на_променливаN;
};
```

Обявяването на структура е инструкция, затова завършва с ";"

Заб. В C++ структурите и обединенията имат както обектно-ориентирани, така и не обектно-ориентирани атрибути. В нашия курс разглеждаме само вторите

Членовете на една структура могат да бъдат:

- променливи от основните типове;
- променливи от enum-тип;
- указатели;
- променливи от други структури;
- променливи от union-тип (union-типовете ще се разгледат по-късно);
- битови полета (битовите полета ще се разгледат по-късно);
- масиви;

Т.е. да отбележим изрично: **Членовете на една структура могат да бъдат променливи от друга структура. (т.е. да влагаме структури)**

Важно е да се разбере, че обявяването на структура означава създаване нов, определен от програмиста тип данни. Обявяването на структура не води до заделяне на памет. Това е просто един шаблон, който показва структурата на потребителския тип данни. Вземете за пример тип **int**, който е вграден в езика C++. Компилаторът заделя памет едва когато потребителят дефинира променлива от тоя тип. Аналогично е и със структурите, **компилаторът заделя памет едва когато потребителят дефинира променлива от структурен тип**.

Определянето на структурна-променлива изглежда така:

struct име_на_структура име_на_структурна_на_променлива;

Ключовата дума може **struct** да се пропусне. Т.е. следното

име_на_структура име_на_структурна_на_променлива;

също е правилна дефиниция. Ще се задели памет, достатъчна за съхранение на всички членове на структурата.

Пример. Ще определим структура, която да съдържа информация за стоките на една компания, съхранявани на склад:

```
struct inventar {
        char stoka[34]; // ime na stokata
        double cena; // sebestoinost
        double pazar_cena; // pazarna cena
        int v_nalichnost; // nalichno kolichestvo
        int sledvashta_dostavka; // broi dni do popylvsne na zapasite
};
inventar   st1, st2, st3; //всяка от променливите си има собствени членове
```

Достъпът до членовете на структурата става с точка.

име_на_структурна_променлива.име_на_член

Членовете на структурната-променлива могат да се използват навсякъде където могат да се използват обикновени променливи от същия тип. Пример: cout << st2.pazar_cena << "\n"; gets(st1. stoka);

Масиви от структури - Структурите могат да бъдат елементи на масиви. Това се среща често, пример за 50-елементен масив с нашата структура:

inventar atrikul[50];

За да получим достъп до конкретен елемент (т.е. конкретна структура) на масива, както и преди използваме индекс, а после отново използваме точка за достъп до даден член на структурата. Пример как да изведем на екран наличното количество на шестата стока (шестият елемент):

cout << atrikul[5]. v_nalichnost << "\n"; // защото индексът започва от нула!

Ще разгледаме програма_62 с използването на вече обявената структура, която трябва да осигури изпълнение на следните действия:

- Въвеждане на информация за стоките, съхранявани на склад;
- Извеждане на екрана на всички стоки;
- Модификация на зададена стока;

```
using namespace std; // nashata 62-a programa
#include <iostream> // primer za struktura, izpolzvana za
#include <cstring> // stoki, syhraniavani na sklad
const int SIZE = 50;// razmer na masiva ot strukturi
struct inventar {// ! masivyt go obiaviavame tuka, wednaga sled strukturata
	char stoka[34]; // ime na stokata
	double cena; // sebestoinost
	double pazar_cena; // pazarna (prodajna) cena
	int v_nalichnost; // nalichno kolichestvo
	int sledvashta_dostavka; // broi dni do popylvsne na zapasite
} artikul[SIZE];
int menu(); void vavedi(), init_list(), pokaji(), obnovi(), input(int i); // prototipi
int main(){
	char choice;
	init_list();
```

```
    for (; ;) {
        choice = menu () ;
        switch(choice) {
            case 'v': vavedi(); break;
            case 'p': pokaji(); break;
            case 'o': obnovi (); break;
            case 'i': return 0;
        }
    }
}
void init_list(){// inicializacia na masiva
    for(int t=0; t<SIZE; t++) {artikul[t].stoka[0] ='\0';}
}
int menu(){ // izbor na potrebitelia
    char ch; cout << '\n';
    do {
        cout << "(V)avedi nova stoka\n"; // Vavedi nov element
        cout << "(P)okaji stokite\n"; // Razpechatai masiva
        cout << "(O)bnovi inf za dadena stoka\n";// Promeni element
        cout << "(I)zhod\n\n"; // Krai na programata
```

```
            cout << "Izberete komanda: ";
            cin >> ch;
        } while(!strchr("vpoi", tolower(ch)));
        return tolower(ch);
}
void vavedi(){ int i;
        for(i=0; i<SIZE; i++){// tyrsim pyrvia svoboden element
            if ( !artikul[i] .stoka[0]) break;}
        if (i==SIZE) {cout << "Spisykyt e pylen!\n"; return;}
        input(i);
}
void input(int i){
        cout << "Stoka: "; cin.sync(); gets(artikul[i].stoka);
        cout << "Sebestoinost: "; cin >> artikul[i].cena;
        cout << "Cena na drebno :"; cin >> artikul[i].pazar_cena;
        cout << "V nalichnost: "; cin >> artikul[i].v_nalichnost;
        cout << "Vreme do nova dostavka (v dni): ";
        cin >> artikul[i].sledvashta_dostavka;
}
```

```
void obnovi(){// promiana na inf. za syshtestvuvashta stoka
int i; char name[80];
      cout << "Vavedete imeto na stokata: "; cin.sync(); gets(name);
      for(i=0; i<SIZE; i++) if(!strcmp(name, artikul[i].stoka)) break;
      if(i==SIZE) {cout << "Stokata ne e namerena!\n"; return;}
      cout << "Vavedete nova informacia.\n"; input (i);
}
void pokaji() {// Izvejdane na ekrana na celia masiv
      int t;
      for(t=0; t<SIZE; t++) {
            if (artikul[t].stoka[0]) {// ako elementyt ne e prazen
                  cout << artikul[t].stoka << '\n';// go izvejdame na ekrana
                  cout << "Sebestoinost " << artikul[t].cena << " lv.";
                  cout << "\nCena na drebno: ";
                  cout << artikul[t].pazar_cena << " lv.";
                  cout << "\nV nalichnost: " << artikul[t].v_nalichnost;
                  cout << "\nDo nova dostavka ostavat: ";
                  cout << artikul[t].sledvashta_dostavka << " dni\n\n";
            }
      }}
```

Като относително по-голяма, програма 62 се нуждае от допълнителни пояснения:

Функцията **init_list()** подготвя масива от структури за работа, записвайки в първия байт на полето **stoka** нулев символ за празен низ. Предполага се, че ако това поле е празно, то структурата, в която то се съдържа не се използва.

Във функцията **menu** потребителят избира какво действие да се извърши, като задължително трябва да се избере едно от 4-те предложения чрез избирането на съответната първа буква. Това става благодарение на библиотечната функция strchr (), която има такъв прототип:

```
char *strchr(const char *str, int ch);
```

Функцията проверява дали в низа, сочен от указателя **str**, се среща символът **ch**. Ако символът се открие се връща указател към първото негово срещане. В тоя случай по определение върнатата стойност е ИСТИНА. Но ако

символът не се среща функцията връща нулев указател, който отново по определение има стойност ЛЪЖА. Въпреки, че аргументът **ch** на функцията е от тип **int** той вътрешно се преобразува до **char** преди проверката. Функцията **tolower** прави възможно да се избере както главна, така и малка буква.

Функцията **vavedi** първо намира празна структура и само ако се открие такава се извиква функцията **input**, която попълва от клавиатура елементите на структурата. Ако не се открие свободна структура проверката **(i==SIZE)** ще върне ИСТИНА. За името на стоката в **input** се използва вече разгледаната функция **gets** защото така в името на стоката може да се съдържат и празни символи. Кодът в **input** е обособен като отделна функция, а не е част от **vavedi** защото се използва и във функцията **obnovi**. Функцията **obnovi** е подобна на **vavedi** като основната разлика е, че сега се търси структура, в която името на стоката съвпада с името, въведеното от клавиатура.

Последната функция **pokaji** извежда на екрана елементите на всички структури, които са запълнени, т.е. на които името на стоката не е празен низ.

Главната функция **main** представлява един типичен пример на вечен цикъл **for**, който завършва само когато функцията **menu** върне символа **"i"** за край на програмата.

Самата програма може да се разшири с нови възможности (които биха се реализирали като отделни функции), като например търсене на елементи по други членове на структурата, изтриване на вече ненужна стока и други. Особеност: програмата ще се спре да работи, ако в **input** вместо очакваната числова стойност за членовете след **stoka** се въведе недопустим за числовия тип символ. В показаната програма масивът **artikul** е глобален.

Инициализиране на структурни-променливи

Като всяка променлива, структурните-променливи също могат да се инициализират по време на определянето им.

За целта се осигурява инициализатор за всеки член на структурата. Но ако инициализаторите са по-малко от членовете, членовете, за които не достигат инициализатори се инициализират с 0 или NULL (за указатели). При това можем да пропуснем само последните инициализатори. Инициализаторите трябва да са константи, разделени със запетая един от друг и затворени в къдрави скоби.

```
struct име_на_структура     име_на_структурна_променлива =
{   инициализатор-член1,
     инициализатор-член2,
     …
     инициализатор-членN
};
```

За вече дефинираната структура една възможна инициализация е следната:

inventar leka_kola = {"Honda Acord", 27000, 42056.34, 5, 200};

Присвояване на структурни-променливи

Структурните-променливи могат да се присвояват една на друга. Това е възможно само ако структурните-променливи са от един и същ тип. Пример:

inventar leka_kola = {"Honda Acord", 27000, 42056.34, 5, 200};

inventar leka_kola2; leka_kola2 = leka_kola; // inventar leka_kola2=leka_kola;

След изпълнение на присвояването двете променливи ще имат еднакви стойности за всичките си членове.

Структури и оператора sizeof

Различните компютри (архитектури) имат различни изисквания как да се разполагат многобайтовите променливи в паметта. Едни компютри изискват многобайтовите променливи да се разполагат на определен адрес, най-често кратен на **машинната дума**. **Такива променливи се наричат подравнени в паметта.** За такива компютри достъпът до неподравнени променливи би довел до специалната ситуация **генериране на изключение**. Други компютри (по-рядко се среща) нямат изискване за подравняване на многобайтовите променливи, но при тях достъпът до неподравнени променливи изисква повече машинни инструкции, отколкото ако са подравнени.

Поради гореописаните изисквания, при разполагане на членовете на структурна-променлива в паметта, могат да се появят „дупки“ (уплътняващи байтове) между два съседни члена и след последния член. Това означава, че

действителният размер на структурната-променлива може да е по-голям от сумарния размер на всички членове.

Размерът на една структура (броят на байтовете, заемани от нейните членове в паметта) трябва да се определя с операцията sizeof. Пример:

```
cout << "Razmer na struct inventar = "<<sizeof(inventar)<<" baita\n\n";
```

извежда 64 докато сумата от дължините на елементите е 58 байта.

Накратко: **Машинна дума** — машинно-зависима и платформо-зависима величина, измервана в битове или байтове, равна на размера на регистрите на процесора и/или на шината за данни.

Структура като параметър на функция

При предаване на структура в качеството си на аргумент към функция се използва механизмът за предаване на параметри по стойност. Вече знаем какво означава това: каквито и да е промени, направени в структурата в тялото на функцията, не влияят на структурата, използвана като аргумент. Все пак трябва да сме наясно, че предаването на големи структури води до значителни разходи на системни ресурси. На практика „структура като параметър на функция“ се среща рядко, за разлика от указателите към структури, които ще разгледаме в следващия раздел.

Указатели към структури

Членовете на една структура могат да се достигат и чрез указател. Обобщената форма на дефиниране на указател към структура изглежда така: **тип_на_структура *име_на_указател;**

Достъпът до членовете на структура чрез указател става със стрелка -> така

име_на_указател -> име_на_член

Указател към структура може да се използва като параметър на функция. Следва вариант на предишната програма, където масивът от структури е локален и достъпът до него се осъществява с указател. И за трите основни функции при тяхното викане аргументът е името на масива, при което, както знаем, се предава адресът на първия елемент на масива. Самият формален параметър може да се укаже по два начина: с име на масив както е в **init_list()** или като обикновен указател, както е при другите функции.

```
using namespace std; // nashata 62b programa
#include <iostream> // primer za struktura, izpolzvana za
#include <cstring> // stoki, syhraniavani na sklad
const int SIZE = 50;// razmer na masiva, koito se sega e lokalen
struct inventar {// imame funkcii, v koito ukazatel kym struktura e parametyr
	char stoka[34]; // ime na stokata
	double cena; // sebestoinost
	double pazar_cena; // pazarna (prodajna) cena
	int v_nalichnost; // nalichno kolichestvo
	int sledvashta_dostavka; // broi dni do popylvsne na zapasite
}; // t.e. strukturata e syshtata kato v prog_62
int menu(); void vavedi(inventar *p), init_list(inventar *p),
  pokaji(const inventar *p), obnovi(inventar *p), input(inventar *p);
int main(){
	char choice;
	inventar artikul[SIZE];// lokalen masiv
	init_list(artikul);
	inventar kola = {"Honda Acord", 27000, 42056.34, 5, 200};
	inventar kola2 = kola;
	cout << "Razmer na struct inventar = "<<sizeof(inventar)<<" baita\n\n";
```

```
	cout << kola2.stoka << '\n'; // Da proverim dali inicializaciata e uspeshna
	cout << "Sebestoinost " <<kola2.cena << " lv.";
	cout << "\nCena na drebno: "<< kola2.pazar_cena << " lv.";
	cout << "\nV nalichnost: " << kola2.v_nalichnost;
	cout << "\nDo nova dostavka ostavat: ";
	cout << kola2.sledvashta_dostavka << " dni\n\n";
	for (; ;) {
		choice = menu () ;
		switch(choice) {
			case 'v': vavedi(artikul); break;
			case 'p': pokaji(artikul); break;
			case 'o': obnovi(artikul); break;
			case 'i': return 0;
		}
	}
	return 0;
}
void init_list(inventar art[]){// inicializacia na masiva
	for(int t=0; t<SIZE; t++) {art[t].stoka[0] ='\0';}}
int menu(){ // izbor na potrebitelia
```

```
    char ch; cout << '\n';
    do {
        cout << "(V)avedi nova stoka\n"; // Vavedi nov element
        cout << "(P)okaji stokite\n"; // Razpechatai masiva
        cout << "(O)bnovi inf za dadena stoka\n";// Promeni element
        cout << "(I)zhod\n\n"; // Krai na programata
        cout << "Izberete komanda: ";
        cin >> ch;
    } while(!strchr("vpoi", tolower(ch)));
    return tolower(ch);
}
void vavedi(inventar *p){
    int i;
    for(i=0; i<SIZE; i++){// tyrsim pyrvia svoboden element
        if ( !(p+i)->stoka[0]) break;}
    if(i==SIZE) {cout << "Spisykyt e pylen!\n"; return;}
    input(p+i);
}
void input(inventar *p){
    cout << "Stoka: "; cin.sync(); gets(p->stoka);
```

```
        cout << "Sebestoinost: "; cin >> p->cena;
        cout << "Cena na drebno :"; cin >> p->pazar_cena;
        cout << "V nalichnost: "; cin >> p->v_nalichnost;
        cout << "Vreme do nova dostavka (v dni): ";
        cin >> p->sledvashta_dostavka;
}
void obnovi(inventar *p){ // promiana na syshtestvuvashta stoka
        int i;
        char name[80];
        cout << "Vavedete imeto na stokata: "; cin.sync(); gets(name);
        for(i=0; i<SIZE; i++) if(!strcmp(name, (p+i)->stoka)) break;
        if(i==SIZE) {cout << "Stokata ne e namerena!\n"; return;}
        cout << "Vavedete nova informacia.\n"; input (p+i);
}
void pokaji(const inventar *p) { // Izvejdane na ekrana na celia masiv
        for(int t=0; t<SIZE; t++) {
                if((p+t)->stoka[0]) {
                        cout << (p+t)->stoka << '\n';
                        cout << "Sebestoinost " << (p+t)->cena << " lv.";
                        cout << "\nCena na drebno: ";
```

```
            cout << (p+t)->pazar_cena << " lv.";
            cout << "\nV nalichnost: " << (p+t)->v_nalichnost;
            cout << "\nDo nova dostavka ostavat: ";
            cout << (p+t)->sledvashta_dostavka << " dni\n\n";
        }
    }
}
```

Тъй като от функцията **pokaji** не се очаква да променя масива, а само да го изведе на екрана, е използван спецификаторът **const** към нейния указател. Навсякъде, където в първия вариант на задачата имаме **artikul[i].**, т.е. достъп до **i**-тата структура във втория вариант се използва **(p+i)->** Защото **p** сочи към началото на масива, оттам **(p+i)** сочи също към **i**-тата структура (съдържа началния адрес на **i**-я елемент), а стрелката осигурява достъп до съответния член, каквото прави и операторът **.** (точка) когато достъпът не е с указател.

За достъп до структура може да се използва и псевдоним, но само за единична структура, а не за масив от структури както в разгледаната задача. За получаване на достъп до членовете на структурата се употребява операторът “точка”. Операторът “стрелка” се използва само с указател.

Битови полета

Битовото поле е целочислена променлива, която се състои от определен брой битове. Битовото поле може да бъде само член на структура или обединение. Определянето на битово поле изглежда така:

тип име_на_битово_поле: широчина;

където **тип** може да бъде само разновидност на **int.** Трябва да се отбележи, че **signed** или **unsigned** както и **short** или **long** също могат да се използват. Ако компилаторът поддържа типа **long long** той също може да се употреби.

Т.е. каквото е позволено при обявяване на целочислена променлива може и тука.

широчина определя броя на битовете в битовото поле и е константен неотрицателен целочислен израз, чиято стойност не трябва да превишава броя на битовете на специфицирания тип (но зависи и от компилатора).

Пример:

```
struct day {
        unsigned  long sec:6; // ! izprobvaite sys sec:7 i signed min:6;
        unsigned  long min:6;
        unsigned  long hour:5;
        unsigned long  days:15;// do (2 na stepen 15) - 1 dni raboti verno
    } d; // структурната променлива с битови полета d от типа day
```

Понеже секундите са число от 0 до 59 шест бита са достатъчни за да се помести стойността, а знаков бит не е необходим. Аналогично пет бита са достатъчни за часовете от 0 до 23. Тъй като **long int** има размер (за нашия

компилатор) 32 бита, то ако заделим 15 бита за **days** общия брой битове на всички членове на структурата също е 32 и размерът на тип **day** ще е 4 байта.

Все пак истинският размер на структурата е добре да се узнава чрез оператора **sizeof**. При 15 бита за **days** максималната стойност, която може да се запише в нея е $2^{15}-1$. Но ако очакваме по-големи стойности се налага да увеличим броя на битовете на **days**, примерно на 7. Но в такъв случай общия брой битове става 33, а това веднага ще увеличи размера на структурата (до 8 байта за повечето компилатори).

Тънкостите при определяне на размера на структурите с битови полета са онагледени в програма_63, която ще бъде разгледана на упражнения, но не е включена тук. Битовите полета могат да се използват навсякъде, където е позволено да се използват целочислени променливи. За достъп до битовите полета се използват същите конструкции както и за достъп до обичайните членове на структура или обединение, а именно точка и стрелка надясно.

Една структура може да съдържа едновременно битови полета и обичайни членове.

В такъв случай е препоръчително битовите полета да се декларират след обичайните членове. Това води до минимизиране на броя байтове заемани от структурната-променлива. Пример, структура подобна на вече показаната с допълнително небитово поле time.

```
struct day2 {
        unsigned  long time; // обикновено поле
        unsigned  long sec:6;
        unsigned  long min:6;
        unsigned  :2;// поле без име, може даже unsigned  :0;
        unsigned  long hour:5;
        unsigned long  days:15;
    } d2; // структурна променлива d2 от типа day2
```

Битовите полета могат да се инициализират по същия начин както и обичайните полета на структурата. Ако структурата съдържа неименувани

битови полета (такова е полето след полето min), инициализатори за тях не се изискват. Ако се опитате да добавите инициализатор за неименувано поле, компилаторът ще генерира грешка.

Ограничения на битовите полета

- Битово поле не може да съществува самостоятелно извън тялото на структура или обединение. Не може примерно в main() да има **int bp:4;**
- Не може да се извлича адреса на битово поле. Не може **int *p=&d.min;**
- Не може да се дефинира указател към битово поле. Не може **int:5 *p;**
- Не може да се дефинира масив от битови полета. Не може **int:5 m[12];**

Отделно от изброените ограничения трябва да се знае, че битовите полета имат много ниска преносимост към други компютърни архитектури, когато е важно конкретното разположение между полетата. Не можем да знаем

последователността на битовите полета: отдясно наляво или отляво надясно. Т.е. на различни компютри програмите могат да работят различно.

Все пак, може да използваме битови полета за да дефинираме целочислени променливи с определена дължина, ако не се интересуваме от конкретното разположение на битовете в паметта. Например битовите полета често се използват за дефиниране на флагове. Флагът е променлива, подобна на булевата, имаща само две стойности и служи за указване дали дадено събитие е възникнало (1) , или не (0).

Битовите полета често се използват за анализ на входните данни, получени от специализирани външни устройства, влизащи в състава на оборудването на системата. Например, портът за състоянията на последователния адаптер за връзка може да връща байт, организиран като битови полета-флагове.

Обединения (unions)

Обединението, подобно на структурата, е потребителски тип данни, в който обаче **всички членове споделят една и съща памет**. Разбира се, в даден момент може да се използва само един от тези членове.

Членовете на едно обединение могат да бъдат:

- променливи от основни типове;
- масиви;
- указатели;
- структурни-променливи;
- променливи от друго обединение;
- битови полета;
- променливи от тип enum;

Т.е. членовете могат да бъдат различни видове променливи. Тогава обявяването на обединение можем да представим по следния начин:

```
union име_на_обединение
{
    тип име-променлива1;
    тип име-променлива2;
    . . .
    тип име-променливаN;
};
```

Пример:

```
union utype {
    short int i;
    char ch;
};
```

Можем да обявим променлива от новия тип по следния начин: **utype u;**

Необходимо е още веднъж да изясним: невъзможно е да направим така, че това обединение да съхранява и целочислена стойност, и символ едновременно, доколкото променливите **i** и **ch** заемат една и съща памет.

Но програмата може във всеки един момент да обработва информацията, съдържаща се в това обединение, като целочислена стойност или като символ. Следователно, обединението осигурява два (или повече) начина на представяне на една и съща порция данни.

Действителният размер на union-променлива трябва да се определя с операцията sizeof. Причината е, че компилаторът може да вмъкне уплътняващи байтове.

При обявяване на обединение компилаторът автоматично заделя област от паметта, достатъчна за съхранение на променливата от най-големия по обем тип. За да получим достъп към даден елемент от обединението се използва същият синтаксис, акто и при структури: операторите "точка" и "стрелка". Примери: **u.ch = 'R'; utype *pu = &u; pu->i =12345;**

Следващата програма_64 използва обединение за да размести двата байта, които изграждат стойността на една променлива от типа **short int**.

```
using namespace std; // nashata 64 programa
#include <iostream> // primer za obedinenie (union)
void disp_binary(unsigned short u);
union swap_bytes {
	short int num; char ch[2]; };
int main(){ // razmiana na baitovete na edno short int chislo
	swap_bytes sb; char temp;
	sb.num = 31; // dvoichen kod: 0000 0000 0001 1111
	cout << "Nachalni baitove: "; disp_binary(sb.ch[1]); cout << " ";
	disp_binary(sb.ch[0]); cout << "\n\n"; // razmeniame baitovete
	temp = sb.ch[0]; sb.ch[0] = sb.ch[1]; sb.ch[1] = temp;
	cout << " Sled razmianata: "; disp_binary(sb.ch[1]); cout << " ";
	disp_binary(sb.ch[0]); cout << "\n";
	return 0;
}
void disp_binary(unsigned short u){ // Izvejdane na bitovete ot baita
	for(short int t=128; t>0; t=t/2)// pobitovo &
		if (u & t) cout << "1 "; else cout << "0 ";
}
```

Подобно на останалите променливи **union**-променливите също могат да се инициализират по време на дефинирането си. Особеното е, че само първият член на обединението може да се инициализира. Ето защо типът на инициализатора трябва да бъде съвместим с типа на първия член и може да бъде само константа.

В последната програма_65 имаме използването на обединение, в което структура с битови полета е негов член. Типът на структурата **day** е обявен вътре в дефиницията на обединението, но той може да бъде дефиниран и преди това. Тогава обединението би имало вида:

```
union u { day d; signed int mnogo_sek;} g1, g2;
```

Забележете, че union-променливите g1 и g2 са обявени по време на обявяването на самото обединение. Самата програма е решавана по-рано, но с употребата на обикновени променливи. В новия вариант са използвани битови полета като техният размер беше вече разискван в предишния

раздел. Особеното е, че битово поле не може да бъде използвано заедно с инструкция **cin**, (защото **cin** е изградена само за основните типове данни) затова е въведена допълнителната променлива buffer. При въвеждането на всяка от входните стойности се прави проверка за нейната валидност с помощта на цикъл do-while. Логиката за изчисляване на времето на завършване на процеса има малки разлики в сравнение с първият вариант на задачата (програма_65b, непоказана в лекцията, но ще бъде достъпна на упражнения). Причина за това е, че само полето mnogo_sek има достатъчно голям размер да съдържа непреобразуваните секунди, минути и часове. Последната тънкост е как се извежда на екрана новото време. Инструкцията **if** сработва само когато процесът не завършва същия ден, като се прави разлика чрез „условната операция“ разгледана в лекция 4 дали да се изведе „дни“ или „ден“.

```
using namespace std; // nashata 65-ta programa
#include <iostream> // primer za obedinenie i bitovi poleta
union u { // Da se vavede vreme na nachalo na process (chas-min-sec)
	struct day { // i negovata prodyljitelnost v secundi.
		unsigned  long sec:6; // Da se nameri sled kolko dni i v kolko
		unsigned  long min:6; // chasa zavyrshva procesa.
		unsigned  long hour:5;
		unsigned long  days:15;// do (2 na stepen 15) - 1 dni raboti verno
	} d;
	signed int mnogo_sek;
} g1, g2;
int main(){
	unsigned int buffer;
	do{
		cout <<"Prodyljitelnost na procesa v secundi = ";
		cin>>g2.mnogo_sek;
	} while (g2.mnogo_sek <=0);
	do{
		cout <<"hour=";
		cin >> buffer;// ! ne moje cin >> g1.d.hour; Zashtoto
```

```
    } while (buffer > 23); // cin e izgraden samo za osnovnite tipove danni
    g1.d.hour = buffer;
    do{
        cout <<"min=";   cin>>buffer;
    } while (buffer > 59);
    g1.d.min = buffer;
    do{
        cout <<"sec=";   cin>>buffer;
    } while (buffer > 59);
    g1.d.sec = buffer;  g2.mnogo_sek += g1.d.sec; // obsht broi sekundi
    g1.d.sec = g2.mnogo_sek % 60; //
    g2.mnogo_sek = g1.d.min +g2.mnogo_sek/60; // obsht broi minuti
    g1.d.min = g2.mnogo_sek % 60;
    g2.mnogo_sek = g1.d.hour +g2.mnogo_sek/60; // obsht broi chasove
    g1.d.hour = g2.mnogo_sek % 24;
    g1.d.days = g2.mnogo_sek/24; //  dni
    cout <<"\nProcesyt zavyrshva ";
if (g1.d.days > 0) cout<< "sled "<<g1.d.days<<((g1.d.days==1)?" den ": " dni ");
    cout << "v "<<g1.d.hour<<":"<<g1.d.min<<":"<<g1.d.sec<<"\n";
    return 0;   }
```