Структури на податоци. Сопствени типови

Во сите програми кои ги разгледавме досега, работевме со променливи од само еден податочен тип - нормално, тука ги вклучуваме и низите, кои претставуваа множества на податоци од еден ист тип. Но, често се случува да имаме потреба од работа со групи податоци кои не се од ист тип - на пример, сакаме да чуваме податоци кои ќе опишуваат една, или повеќе, книги. Ваквата структура (книга) треба да содржи информации за името на книгата, авторот, бројот на страници, датумот на издавање, итн.

Во C++, доколку програмата треба да работи со само една книга и сите информации се процесираат во само една функција, податоците може да ги чуваме во неколку различни променливи - секоја со свое име: imeKniga, imeAvtor, brojStranici, datumIzdavanje, итн. Но, доколку овие податоци треба да ги предадеме на друга функција, тогаш треба секоја од овие променливи да ја наведеме како аргумент на функцијата. Слично, доколку сакаме да работиме со множества (низи) од книги - тогаш треба да имаме огромен број на променливи, секоја со сопствено име и вредност.

Структури на податоци претставуваат групи од податоци дефинирани под едно име. Секој податок во структурата се нарекува нејзин член. Во програмскиот јазик C++, структури на податоци се дефинираат на следниот начин:

```
struct ime
{
   tip1 clen1;
   tip2 clen2;
   tip3 clen3;
   ....
   tipN clenN;
};
```

На пример, со следниот код може да креираме структура која содржи 4 члена (imeKniga, imeAvtor, brojStranici, datumIzdavanje) и има име Kniga:

Извадок 13.1

```
struct Kniga
{
string imeKniga;
string imeAvtor;
int brojStranici;
string datumIzdavanje;
};
```

Внимавајте на знакот ';' на крајот од дефиницијата на структурата.

За да не се меша дефинирањето на типовите со процесот на извршување на наредби, дефинициите на структурите треба да ги пишувате надвор од функциите - иако тоа не е задолжително. Дефиницијата на податочната структура Kniga (и на која било структура која ќе ја креирате на начинот прикажан погоре) не зазема меморија и не креира променливи од тип Kniga. Таквата дефиниција само опишува какви податоци ќе содржат променливите од тип Kniga кои подоцна ќе ги декларираме. Всушност, со горната дефиниција креираме податочен тип кој понатаму може да го користиме како и сите останати податочни типови (int, char, long, итн.). Во една програма, променливи од тип Kniga декларираме на истиот начин како што декларираме променливи од кој било друг тип:

Kniga imeNaPromenliva;

Со наредбата дадена погоре сме креирале променлива од тип Kniga, со име imeNaPromenliva. Како и секоја друга променлива, така и овие променливи зафаќаат одреден податочен простор (меморија): по правило, овие променливи зафаќаат онолку простор колку што е потребно за да се чуваат податоците - што пак, претставува збир од меморијата потребна за чување на секој од членовите на структурата. Како и кај сите други променливи, може да го искористиме операторот sizeof() за да одредиме колку точно бајти зафаќаат овие податоци.

Во C++, до податоците (членовите) на една структура пристапуваме со помош на операторот '.'. На пример, со наредбата "imeNaPromenliva.imeKniga" пристапуваме до членот imeKniga на променливата imeNaPromenliva.

Програма 13.1

```
#include <iostream>
#include <string>
using namespace std;
struct Kniga
{
string imeKniga;
string imeAvtor;
int brojStranici;
string datumIzdavanje;
};
int main()
{
Kniga prva;
prva.imeKniga = "C++ Primer Plus (5th Edition)";
prva.imeAvtor = "Stephen Prata";
prva.brojStranici = 1224;
prva.datumIzdavanje = "25.11.2004";
Kniga vtora;
vtora.imeKniga = "C++ Primer Plus (6th Edition)";
vtora.imeAvtor = "Stephen Prata";
vtora.brojStranici = 1200;
```

```
vtora.datumIzdavanje = "28.10.2011";
//C++ Primer Plus (5th Edition) - Stephen Prata
cout << prva.imeKniga << " - " << prva.imeAvtor << endl;
//C++ Primer Plus (6th Edition) - Stephen Prata
cout << vtora.imeKniga << " - " << vtora.imeAvtor << endl;
//chlenovite na strukturata se odnesuvaat kako obichni promenlivi
//i vrz sekoja od niv mozhe da izvrshuvame najrazlichni operacii
cout << prva.imeKniga << " ima " << (prva.brojStranici - vtora.brojStranici)
<< " povekje stranici od " << vtora.imeKniga << endl;
//C++ Primer Plus (5th Edition) ima 24 povekje ...
//stranici od C++ Primer Plus (6th Edition)
return 0;
}
```

Во C++, променливите кои се креираат врз база на дефиниција на одреден податочен тип се нарекуваат објекти од тој тип. На пример, во програмата дадена погоре, prva и vtora се објекти од типот Kniga. Многу е важно да ги разграничиме поимите тип и објект: тип претставува дефиниција на податок (што е тоа Kniga и кои податоци треба да се чуваат за една книга), додека објектите ги содржат вистинските податоци - книга со име "C++ Primer Plus (5th Edition)".

Сигурно се прашува(в)те зошто е потребно на крајот од дефиницијата на одредена структура да го напишеме знакот ';' - кога веќе границите на дефиницијата се строго дефинирани со знаците '{' (за почеток на набројувањето на членови) и '}' (за крај на набројувањето). Одговорот е едноставен - C++ овозможува креирање на објекти (променливи) од структурата веднаш по нејзиното дефинирање. Така, помеѓу знаците '}' и ';' може да ги запишеме имињата на тие променливи. На пример, следниот код дефинира структура Kniga и креира две променливи prva и vtora:

Извадок 13.2

```
struct Kniga
{
string imeKniga;
string imeAvtor;
int brojStranici;
string datumIzdavanje;
} prva, vtora;
```

Всушност, доколку сме сигурни дека prva и vtora ќе бидат единствените променливи од тип Kniga и нема функција која очекува аргумент од тој тип, не е ни задолжително да го наведеме името на структурата. На пример, во кодот даден погоре, зборчето Kniga може слободно да го избришеме.

Во C++, како и во повеќето модерни програмски јазици, дозволено е вгнездување на една структура во друга - на пример дефинирање на структура Avtor (imeAvtor, prezimeAvtor, godinaRaganje, итн) и, потоа, дефинирање на структура Kniga која содржи член од тип Avtor. Ова овозможува креирање на најразлични хиерархии на податоци.

Запамтете дека, како и кај сите останати променливи, по креирањето на објекти од одреден тип, потребно е, на истите, да им се додели почетна вредност. Во програмата со книгите, ова го направивме со неколку наредби - наведени веднаш по креирањето на променливите prva и vtora. Бидејќи, многу често, структурите на податоци содржат голем број на членови и потребни се повеќе наредби за нивно иницијализирање, C++ овозможува доделување на вредности уште при самото креирање на променливите (слично како кај низите од податоци). Доколку има вгнездување на структури, иницијализацијата се прави со вгнездување на самата листа од вредности:

Програма 13.2

```
#include <iostream>
#include <string>
using namespace std;
struct Avtor
{
string ime;
string prezime;
};
struct Kniga
{
string imeKniga;
Avtor avtor;
int brojStranici;
};
int main()
{
//inicijalizacija so lista od vrednosti
Avtor avtor = {"Stephen", "Prata"};
Kniga prva = {"C++ Primer Plus (5th Edition)", avtor, 1224};
//C++ Primer Plus (5th Edition) - Stephen
cout << prva.imeKniga << " - " << prva.avtor.ime << endl;
//inicijalizacija (vgnezduvanje)
Kniga vtora = {"Learn C++", {"John", "Doe"}, 1000};
//Learn C++ - John
cout << vtora.imeKniga << " - " << vtora.avtor.ime << endl;
return 0;
}
```

Од примерот даден погоре може да забележите дека, при иницијализацијата на променливите avtor, prva и vtora, вредностите мора да ги наведеме во истиот редослед како што се наведени членовите во дефиницијата на соодветната структура на податоци (кај avtor прво го наведуваме неговото име, па презиме). Бидејќи, при процесот на програмирање, често сакаме да го промениме бројот на членови кај одредена структура на податоци (со додавање или бришење на податоци), користењето на ваквите листи за иницијализација е опасно - може да доведе до грешки при извршување на програмите кои тешко се откриваат.

Набројувања (енумерации)

Набројување претставува нов податочен тип креиран на наједноставниот можен начин - преку наведување на сите вредности кои променливите од тој податочен тип може да ги добијат. C++ овозможува креирање на набројувања преку наведување на клучниот збор enum. Притоа, важно е да се знае дека, во позадина, набројувањата претставуваат нумерички променливи - секоја вредност од енумерацијата соодветствува на одреден цел број - доколку не се наведе поинаку, за првата вредност тоа е 0, за втората 1, за третата 2, итн. Енумерациите зафаќаат точно онолку меморија колку што зафаќа податочниот тип int.

Следната програма креира набројување Mesec и променлива m од тој тип:

Програма 13.3

```
#include <iostream>
#include <string>
using namespace std;
enum Mesec
{
januari, //0
fevruari, //1
mart, //2
april, //3
maj, //4
juni, //5
juli, //6
avgust, //7
septemvri, //8
oktomvri, //9
noemvri, //10
dekemvri //11
};
int main()
{
Mesec m = fevruari;
if (m == januari)
cout << "Januari" << endl; //ne se pechati
if (m == fevruari)
cout << "Fevruari" << endl; //pechati 'Fevruari'
m = mart;
//pretopuvanje vo int
cout << (int)m << endl; //pechati '2'
return 0;
}
```

Како што може да видите, иако во позадина работиме со цели броеви, доделувањето на вредности и нивната споредба се врши транспарентно - дозволено е споредување на месеци без познавање на нивната нумеричка вредност.

Набројувањата најчесто се користат за дефинирање на состојби во програмата. На пример, во следните неколку линии код, наместо да работиме со цели броеви (што би било крајно нечитливо), тековната состојба на програмата ја претставуваме со набројувања.

Извадок 13.3

```
State state = CONNECTING;
if (connect("www.facebook.com") == false)
{
state = NETWORK_FAILURE;
}
else
{
state = CONNECTED;
}
if (state == CONNECTED)
{
if (fbLogin(username, password) == false)
{
state = LOGIN_FAILED;
} else
{
state = LOGGED_IN;
}
}
```

Како што беше тоа случај и со структурите на податоци, може да креираме променливи уште при самата дефиниција на новиот податочен тип. Тоа се прави со наведување на имињата на променливите по знакот '}'.

Извадок 13.4

```
enum State
{
CONNECTING,
NETWORK_FAILURE,
CONNECTED,
LOGGED_IN,
LOGIN_FAILED
} state;
```

C++ ни дозволува, со употреба на операторот за доделување '=', да ги промениме нумеричките вредности кои соодветствуваат на секој елемент од набројувањето. Оние елементи на кои нема експлицитно да им доделиме нова вредност ќе имаат нумеричка вредност за 1 поголема од нивниот претходник во набројувањето (со исклучок на првиот елемент, кој има вредност 0). Можно е повторување на нумеричките вредности и доделување на вредности кои се негативни цели броеви.

Извадок 13.5

```
enum State
{
A = 100, //A=100
B = -5, //B=-5
C, //C=-4
D = 10, //D=10
```

```
E = 10, //E=10
F = 10, //F=10
G, //G=11
H //H=12
};
```

Дефиниции на податочни типови

C++ дозволува креирање на синоними (други имиња) за постоечки податочни типови. Ова се прави со наведување на клучниот збор typedef, име на постоечки тип и име кое сакаме да го користиме како негов синоним:

Програма 13.4

```
#include <iostream>
using namespace std;
typedef int int32;
typedef long long int64;
typedef double real;
typedef char character;
typedef unsigned int uint32;
typedef int array[100];
int main()
{
int32 n = 3;
n++;
cout << n << endl; //pechati '4'
return 0;
}
```

Програмата дадена погоре креира неколку дефиниции (претставени во табелата подолу), и променлива со име n од тип int (int и int32 се сега синоними и се однесуваат на еден ист тип на податок).

ново име	**име на постоечки тип**
int32	int
int64	long long
real	double
character	char
uint32	unsigned int
array	int[100]

Најважната примена на typedef е сокривањето на разликите кај различните архитектури на процесори. На пример, кај еден систем int може да зафаќа 32 бита, додека кај друг да зафаќа 64 бита (кој, соодветно, може да служи за чување на многу поголеми вредности). Дефинирањето на сопствени типови ни овозможува да ги асоцираме постоечките податочни типови (short, int, long, long long, итн.) со имиња на типови (int32, int64, итн.)

кои ние ќе ги користиме. Доколку сме конзистентни и секаде во кодот ги користиме новите имиња, можно е да компајлираме код за различни платформи преку едноставна промена на неколку typedef дефиниции. На пример, при компајлирање на код за 32 битна платформа (x86), int32 ќе биде синоним за еден постоечки тип, додека при компајлирање на код за 64 битна платформа (x64) за друг. Со соодветна промена на типот чиј синоним е int32, можно е да гарантираме дека сите променливи од тип int32 ќе може да се користат за чување на 32-битни вредности.

Унии. Неименувани унии

На сличен начин како што групиравме повеќе податоци во една структура (struct), можно е да направиме унија од податоци до кои може да се пристапи преку едно заедничко име. Униите од податоци се разликуваат од структурите по тоа што податоците ја делат меморијата - пристапуваат до една иста мемориска локација и заземаат онолку меморија колку што е потребна за чување на најголемиот тип на податок (оној кој зафаќа најмногу меморија). Промената на еден податок – член на унијата, влијае на вредноста на сите останати. Од друга страна, сите елементи (членови) на една структура се чуваат на посебна мемориска локација и промената на вредноста на еден член не влијае на вредностите на останатите.

Униите се дефинираат на ист начин како што се дефинираат и структурите на податоци, со тоа што наместо клучниот збор struct користиме union:

```
union ime
{
   tip1 clen1;
   tip2 clen2;
   tip3 clen3;
   ....
   tipN clenN;
} objekti;
```

На пример, следниот код дефинира унија со име vreme која содржи 3 членови (milisekundi, sekundi i minuti). Промената на вредноста на еден од членовите ќе влијае на вредностите на останатите. Практично, во една унија од податоци може да се чува само еден податок.

Извадок 13.6

```
union vreme
{
int minuti;
int sekundi;
long long milisekundi;
};
```

Како и кај структурите, незадолжително е наведувањето на објекти при самата дефиниција на унијата - може да се креираат променливи од тип ime и на друго место во програмата.

До елементите на унијата пристапуваме како и кај структурите на податоци - со употреба на операторот '.' (objekt.minuti, objekt.sekundi, objekt.milisekundi).

Во ситуации кога податоците до кои треба да пристапуваме зависат едни од други, униите може да се искористат и за заштеда на меморија. На пример, може да креираме унија со два члена (int n и char bytes[4]), и, во текот на програмата, да пристапуваме до секој бајт од n преку низата bytes (големината на char е 1 бајт, додека на int е 4 бајти).

Сепак, однесувањето на униите е непредвидливо - редоследот на членовите и нивното подредување во меморијата е зависно од платформата на која се извршува програмата. Поради тоа, треба да се избегнува користењето на унии колку што е можно повеќе. Денес има доволно меморија за да се чуваат одредени податоци и по повеќе пати. Слично, понекогаш е дозволено и да се пресмета одреден податок повеќе пати - доколку пресметките се едноставни и таквата програма е полесна за пишување и тестирање. Од друга страна, ретко се наоѓаат задачи каде што користењето на унии води до поефикасни програми.

За крај, ќе наведеме дека е дозволено вгнездување на унии во структури, и обратно - вгнездување на структури на податоци во унии. C++ дозволува и креирање на т.н. неименувани унии - унии до чии членови може да пристапиме без наведување на името на унијата:

Извадок 13.7

```
struct vreme
{
string imeNastan;
union
{
int minuti;
int sekundi;
long long milisekundi;
};
} nekojNastan;
```

До членовите minuti, sekundi и milisekundi пристапуваме со изразите:

nekojNastan.minuti
nekojNastan.sekundi
nekojNastan.milisekundi

Доколку, кај дефиницијата на унијата, беше дадено име на објект (на пример, nekojObjekt), тогаш до елементите на унијата ќе пристапувавме на следниот начин: nekojNastan.nekojObjekt.sekundi.