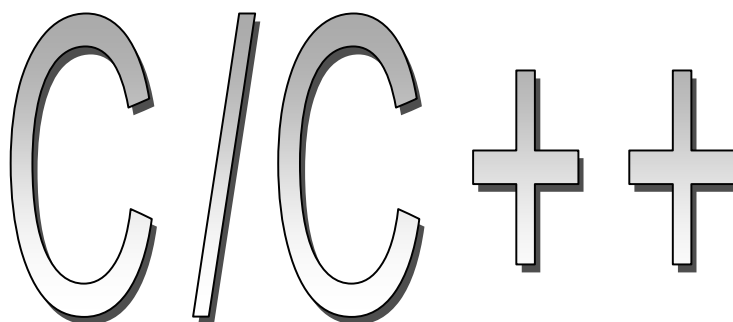




Programmer's Reference

Herbert Schildt

***Osborne / McGraw – Hill
1997 , Berkeley CA
ISBN 0-07-882367-6***

*Превод и обработка – MesserSchmidt
gripen@abv.bg*

Съдържание:

Увод	12
ГЛАВА 1 - Типове данни, променливи и константи.....	12
Основни типове.....	12
Деклариране на променливи.....	14
Инициализиране на променливи.....	14
Идентификатори.....	14
Класове.....	15
Наследяване.....	17
Структури.....	18
Обединения.....	18
Изброявания.....	20
С тагове.....	21
Спецификатори на класове за съхранение.....	22
extern.....	22
auto.....	22
register.....	22
static.....	22
mutable.....	23
Типови определители.....	23
const.....	23
volatile.....	24
Масиви.....	24
Дефиниране на нови типове имена използвайки typedef.....	25
Константи.....	25
Шестнайсетични и осмични константи.....	26
Низови константи.....	26
Булеви константи.....	26
Константи след обратно наклонена черта.....	26
ГЛАВА 2 - Функции , интервали , именовани пространства и хедъри.....	27
Функции.....	27
Рекурсия.....	28
Предефиниране на функции.....	29
Подразбиращи се аргументи.....	29
Прототипи.....	30
Разбиране на интервалите и продължителността на съществуване на променлива.....	31
Именовани пространства.....	32

Функцията <code>main()</code>	32
Аргументи на функция.....	33
Подаване на указатели.....	34
Параметри псевдоними.....	35
Конструктори и деструктори.....	35
Спецификатори на функция.....	36
Свързваща спецификация.....	36
Стандартните библиотеки на C и C++.....	37

ГЛАВА 3 - Оператори 39

Аритметични оператори.....	39
Релационни и логически оператори.....	39
Побитови оператори.....	40
Оператори <code>&</code> , <code> </code> и <code>^</code>	41
Оператор за единствено допълнение <code>~</code>	41
Преместващите оператори <code>>></code> и <code><<</code>	41
Указателни оператори.....	42
Указателен оператор <code>&</code>	42
Указателен оператор <code>*</code>	43
Оператори за присвояване.....	43
Оператора <code>?</code>	43
Оператори за членове.....	44
Оператора запетайка.....	44
<code>sizeof</code>	45
Преобразуване.....	45
Преобразувания в C++.....	45
Оператори за I/O.....	46
<code>.</code> , <code>*</code> и <code>->*</code> указатели към член функции.....	46
Оператор за определяне област на видимост <code>::</code>	47
<code>new</code> и <code>delete</code>	47
<code>typeid</code>	47
Предефиниране на оператори.....	48
Таблица на приоритетите на операторите.....	48

ГЛАВА 4- Предпроцесор и коментари..... 49

<code>#define</code>	49
<code>#error</code>	50
<code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> , <code>#else</code> , <code>#elif</code> , и <code>#endif</code>	50
<code>#include</code>	51
<code>#line</code>	52
<code>#pragma</code>	52
<code>#undef</code>	52
Предпроцесорните оператори <code>#</code> и <code>##</code>	53
Предефинирани имена на макроси.....	53

Коментари.....	54
----------------	----

ГЛАВА 5 – Справочник на ключовите думи.....	54
--	-----------

asm.....	55
auto.....	55
bool.....	55
break.....	55
case.....	55
catch.....	56
char.....	56
class.....	56
const.....	56
const_cast.....	56
continue.....	57
default.....	57
delete.....	57
do.....	57
double.....	57
dynamic_cast.....	58
else.....	58
enum.....	58
explicit.....	58
extern	59
false.....	59
float.....	59
for.....	59
friend.....	60
goto.....	60
if.....	61
inline.....	62
int.....	62
long.....	62
mutable.....	62
namespace.....	62
new.....	63
operator.....	64
private.....	65
protected.....	65
public.....	66
register.....	66
reinterpret_cast.....	67
return.....	67
short.....	67
signed.....	67
sizeof.....	67

static.....	68
static_cast.....	68
struct.....	68
switch.....	69
template.....	70
this.....	73
throw.....	73
true.....	74
try.....	75
typedef.....	75
typeid.....	75
typename.....	75
union.....	75
unsigned.....	76
using.....	76
virtual.....	76
void.....	76
volatile.....	77
wchar_t.....	77
while.....	77

ГЛАВА 6 – Стандартни C I/O функции..... 77

clearerr.....	78
fclose.....	78
feof	79
ferror.....	79
fflush.....	79
fgetc.....	79
fgetpos.....	80
fgets.....	80
fopen.....	80
fprintf.....	82
fputc.....	82
fputs.....	82
fread.....	83
freopen.....	83
fscanf.....	83
fseek.....	83
fsetpos.....	84
ftell.....	84
fwrite.....	84
getc.....	85
getchar.....	85
gets.....	85
perror.....	86

printf.....	86
putc.....	88
putchar.....	89
puts.....	89
remove.....	89
rename.....	89
rewind.....	89
scanf.....	90
setbuf.....	92
setvbuf.....	92
sprintf.....	92
sscanf.....	93
tmpfile.....	93
tmpnam.....	93
ungetc.....	93
vprintf () , vfprintf () и vsprintf ()	94

ГЛАВА 7 – С - низови и символни функции..... 94

isalnum.....	94
isalpha.....	94
iscentrl.....	95
isdigit.....	95
isgraph.....	95
islower.....	95
isprint.....	95
ispunct.....	95
isspace.....	96
isupper.....	96
isxdigit.....	96
memchr.....	96
memcmp.....	96
memcpy.....	96
memmove.....	97
memset.....	97
strcat.....	97
strchr.....	97
strcmp.....	97
strcoll.....	98
strcpy.....	98
strcspn.....	98
strerror.....	98
strlen.....	98
strncat.....	99
strncmp.....	99
strncpy.....	99

strpbrk.....	99
strrchr.....	100
strspn.....	100
strstr.....	100
strtok.....	100
strxfrm.....	101
tolower.....	101
toupper.....	101

ГЛАВА 8 – Математични функции в C..... 101

acos.....	102
asin.....	102
atan.....	102
atan2.....	102
ceil.....	102
cos.....	103
cosh.....	103
exp.....	103
fabs.....	103
floor.....	103
fmod.....	103
frexp.....	103
ldexp.....	104
log.....	104
log10.....	104
modf.....	104
pow.....	104
sin.....	104
sinh.....	104
sqrt.....	105
tan.....	105
tanh.....	105

ГЛАВА 9 – Функции в C за време , дата и локализационни функции..... 105

asctime.....	106
clock.....	106
ctime.....	106
difftime.....	106
gmtime.....	107
localeconv.....	107
localtime.....	108
mktime.....	108
setlocale.....	109

srftime.....	110
time.....	111

ГЛАВА 10 – Функции в С за динамично заделяне

calloc.....	112
free.....	112
malloc.....	112
realloc.....	113

ГЛАВА 11 – Разнообразни функции на С..... 113

abort.....	114
abs.....	114
assert.....	114
atexit.....	114
atof.....	115
atoi.....	115
atol.....	115
bsearch.....	115
div.....	116
exit.....	116
getenv.....	116
labs.....	116
ldiv.....	117
longjmp.....	117
qsort.....	117
raise.....	118
rand.....	118
setjmp.....	118
signal.....	119
srand.....	119
strtod.....	119
strtol.....	120
strtoul.....	120
system.....	120
va_arg () , va_start () и va_end ()	121

ГЛАВА 12 – Системата на С++ за I/O в стар стил..... 122

Основни класове за потоци.....	123
Предефинирани потоци в С++.....	123
Форматни флагове.....	123
I/O Манипулатори.....	124
Iostream функции в стар стил.....	126
bad.....	126

clear.....	126
eatwhite.....	127
eof.....	127
fail.....	127
fill.....	127
flags.....	127
flush.....	127
fstream () , ifstream () и ofstream ()	128
gcount.....	129
get.....	129
getline.....	129
good.....	130
ignore.....	130
open.....	130
peek.....	132
precision.....	132
put.....	132
putback.....	132
rdstate.....	133
read.....	133
seekg () и seekp ()	133
setf.....	134
setmode.....	134
str.....	135
stringstream () , istream () и ostream ()	135
sync_with_stdio.....	136
tellg () и tellp ()	136
unsetf.....	136
width.....	136
write.....	137

ГЛАВА 13 – Стандартната система на C++ за I/O..... 137

Използване на новата iostream библиотека	137
Базови класове за I/O.....	137
Предефинирани потоци в C++.....	138
Типа streamsize.....	138
Типа iostate.....	139
Форматни флагове и типа fmtflags.....	139
I/O манипулатори.....	140
Стандартни iostream функции.....	142
bad.....	142
clear.....	142
eof.....	142
fail.....	142
fill.....	142

flags.....	143
flush.....	143
fstream () , ifstream () и ofstream ()	143
gcount.....	144
get.....	144
getline.....	145
good.....	145
ignore.....	145
open.....	145
peek.....	146
precision.....	146
put.....	147
putback.....	147
rdstate.....	147
read.....	147
seekg () и seekp ().....	148
setf.....	148
sync_with_stdio.....	149
tellg () и tellp ()	149
unsetf.....	149
width.....	149
write.....	150

ГЛАВА 14 – Стандартната библиотека с шаблони (STL)

на C++	150
Преглед на контейнери , алгоритми и итератори	150
Контейнер.....	150
Алгоритми.....	151
Итератори.....	151
Алокатори.....	152
Предикати и сравнителни функции.....	152
Инструменти и функционални хедъри.....	152
Контейнерни класове	153
bitset.....	155
deque.....	156
list.....	158
map.....	160
multimap.....	162
multiset.....	163
queue.....	165
priority_queue.....	165
set.....	166
stack.....	167
vector.....	168
Алгоритми.....	170

adjacent_find.....	170
binary_search.....	171
copy.....	171
copy_backward.....	171
count.....	171
count_if.....	171
equal.....	171
equal_range.....	171
fill и fill_n.....	172
find.....	172
find_end.....	172
find_first_of.....	172
find_if.....	172
for_each.....	172
generate и generate_n.....	173
includes.....	173
inplace_merge.....	173
iter_swap.....	173
lexicographical_compare.....	173
lower_bound.....	174
make_heap.....	174
max.....	174
max_element.....	174
merge.....	174
min.....	174
min_element.....	175
mismatch.....	175
next_permutation.....	175
nth_element.....	175
partial_sort.....	176
partial_sort_copy.....	176
partition.....	176
pop_heap.....	176
prev_permutation.....	176
push_heap.....	177
random_shuffle.....	177
remove, remove_if, remove_copy и remove_copy_if.....	177
replace, replace_copy, replace_if и replace_copy_if.....	178
reverse и reverse_copy.....	178
rotate и rotate_copy.....	178
search.....	179
search_n.....	179
set_difference.....	179
set_intersection.....	179
set_symetric_difference.....	180
set_union.....	180

sort.....	180
sort_heap.....	180
stable_partition.....	181
stable_sort.....	181
swap.....	181
swap_ranges.....	181
transform.....	181
unique и unique_copy.....	182
upper_bound.....	183

ГЛАВА 15 – C++ Низове и изключения.....	183
Низове.....	183
Изключения.....	192
< exception >.....	192
< stdexcept >.....	193

УВОД

C и C++ са най - важните световни програмни езици . Всъщност, за да бъдете професионален програмист днес означава опитност в тези два езика.Те са основата, върху която е изградено модерното програмиране. C беше изобретен от Денис Ричи през 1970г. C е език от средно ниво.Той комбинира контролните структури на езиците от високо ниво със способността да манипулира битове, байтове и указатели (адреси) . Така C дава почти цялостен контрол на програмиста върху машината . C беше стандартизиран късно през 1989г.когато стандарта на Американския национален институт за стандарти /ANSI/ беше одобрен.Този стандарт беше одобрен и от ISO /Международната организация за стандарти/ през 1990г.Стандартът на C беше коригиран леко през 1996г.

C++ беше създаден от **Bjarne Stroustrup** през 1980г.Той започва като множество от обектно-ориентирани разширения на C, но скоро се разшири да съществува като отделен програмен език.Докато C++ е все още изграден върху основата на C, то неговите нови свойства увеличиха почти двойно размера на езика.Излишно е да казвам, че C++ е един от най-мощните компютърни езици създадени досега.Стандарта ANSI/ISO за C++ беше завършен и сега C++ е стандартизиран език.Материалът в тази книга описва ANSI стандарт C и ANSI/ISO стандарт C++.

Както вие несъмнено усещате C и C++ са обширни теми.Тук, разбира се не е възможно да покривем всички аспекти на тези важни езици.Вместо това този бърз справочник извлича техните най-забележителни качества в удобна и лесна за употреба форма.

ГЛАВА 1 - Типове данни,променливи и константи

C и C++ предлагат на програмиста богат асортимент от вградени типове данни . Програмистки-дефинираните типове данни могат да бъдат създадени да стават навсякъде където трябва . Също възможно е да определите константи от вградените в C/C++ типове . В тази глава са описани разнообразни черти отнасящи са до типовете данни , променливите и константите.

Основни типове

C дефинира 5 основни вградени типове данни:

Тип	ключова дума
Символ	<i>char</i>
Цяло число	<i>int</i>
Число със плаваща точка	<i>float</i>
Двойно число със плаваща точка	<i>double</i>
Без стойност	<i>void</i>

C++ добавя два:

Тип	ключова дума
Булев(<i>true/false</i>)	<i>bool</i>
Широко-символен	<i>wchar_t</i>

Няколко от тези типове могат да бъдат модифицирани чрез използване на един или повече от тези модификатори за тип:

signed unsigned short long

Типовите модификатори предшестват името на типа което изменят. Всички вградени типове данни включително модификатори позволени в C и C++ са показани в следващата таблица заедно със техните минимални гарантирани интервали. Повечето компилатори ще превишават минимума за един или повече типове. Също ако вашия компютър използва два комплекта аритметика (както повечето правят), тогава най-малката отрицателна стойност която може да бъде съхранена чрез цяло число със знак ще бъде със 1 повече от показания минимум. Например интервала на *int* за повечето компютри -32 768 до 32767. Дали типа *char* е със знак или е без знак е в зависимост от изпълнението.

Тип	минимален интервал
<i>bool</i>	<i>true/false</i>
<i>char</i>	-127 до 127 или 0 до 255
<i>unsigned char</i>	0 до 255
<i>signed char</i>	-127 до 127
<i>int</i>	-32 767 до 32 767
<i>unsigned int</i>	0 до 65 535
<i>signed int</i>	също като <i>int</i>
<i>short int</i>	също като <i>int</i>
<i>unsigned short int</i>	0 до 65 535
<i>signed short int</i>	също като <i>short int</i>
<i>long int</i>	-2 147 483 647 до -2 147 483 647
<i>signed long int</i>	също като <i>long int</i>
<i>unsigned long int</i>	0 до 4 294 967 295
<i>float</i>	6 цифрова точност
<i>double</i>	10 цифрова точност
<i>long double</i>	10 цифрова точност
<i>wchar_t</i>	също като <i>unsigned int</i> .

Когато модификатора за тип е използван сам се предполага *int*. Например вие може да определите *unsigned int* като просто използвате ключовата дума *unsigned*. Така тези декларации са еквивалентни :

```
unsigned int i;  
unsigned i ; // тук int е по подразбиране
```

Деклариране на променливи

Всички променливи трябва да бъдат декларирани преди употреба. Общата форма за деклариране е:

```
тип име_на_променлива;
```

Например за да декларирате *x* да бъде *float*, *y* да бъде *int* и *ch* да бъде символ вие трябва да напишете :

```
float x ;  
int y ;  
char ch ;
```

Вие може да декларирате повече от една променлива от тип чрез използване на разделен със запетайки списък. Например следващия израз декларира три цели числа :

```
int a , b , c ;
```

Инициализиране на променливи

Променлива може да бъде инициализирана чрез следващ след нейното име знак за равенство и начална стойност. Например тази декларация присвоява на *count* начална стойност 100:

```
int count = 100;
```

Инициализатор може да бъде всеки валиден израз когато е декларирана променлива . Това включва други променливи и функции. Обаче във C глобалните променливи и *static* локални променливи трябва да бъдат инициализирани чрез изрази само от константи .

Идентификатори

Променливи , функции и потребителски дефинирани имена на типове са примери за идентификатори . В C/C++ идентификаторите са поредица от букви , цифри , и подчертавания със един или няколко символа дължина (идентификатора не може да започва обаче с цифра). Идентификаторите могат да имат всякаква дължина , но в C само първите 31 символа са гарантирано от значение . Подчертаването е често използвано за яснота , като *first_time* или за начало на име като *_count*. Главните и малките букви се приемат за различни. Например *test* и *TEST* са две различни променливи. C++ резервира всички идентификатори които започват със две подчертавания или подчертаване следвано от главна буква .

Класове

Класът е основната част в капсулирането на C++. Класът е дефиниран чрез използването на ключовата дума *class*. Класовете не са част от езика C. Класа е основата си сбирка от променливи и функции които манипулират тези променливи. Променливите и функциите които оформят класа са наречени членове. Общата форма на *class* е показана тук:

```
class име_на_клас : списък-с-наследявания {
    // членове private по подразбиране
protected:
    // членове private които могат да се наследяват
public:
    // членове public
} списък от обекти;
```

Тук *име_на_клас* е името на класовия тип. Веднъж когато декларацията на класа бъде съставена, *име_на_клас* става нов тип име за данни което може да се използва за деклариране на обекти от класа. *Списъка с обекти* е разделен със запетайки списък от обекти от тип *име_на_клас*. Този списък е по избор. Обектите класове могат да бъдат декларирани по-късно във вашата програма просто чрез използване на името на класа. *Списъка с наследявания* е също по избор. Когато е представен, той определя базовия клас или класовете които наследява новия клас. (Виж предстоящата секция озаглавена "Наследяване") Класа може да включва функция конструктор и функция деструктор (и двете са по избор). Конструктора се извиква когато се създава първи обект от класа. Деструктора се извиква когато се разрушава обект. Конструктора има същото име като класа. Деструктора има същото име като класа но предшествано със ~ (тилда). Нито конструктора нито деструктора имат връщани типове. В класовата йерархия, конструкторите се изпълняват по ред на наследяването, а деструкторите се изпълняват в обратен ред. По подразбиране всички елементи на класа са *private* за този клас и могат да бъдат достъпни само за другите членове на този клас. За да позволите на елемент от класа да бъде достъпен за функции които не са членове на класа, вие трябва да го декларирайте след ключовата дума *public*. Например

```
class myclass {

    int a, b;

    public:

    // членове на клас достъпни за нечленове

    void setab (int I, int j) {a=I; b=j;}

    void showab() {cout<<a<<' '<<b<<endl;}
```

```
};

myclass ob1, ob2;
```

Тази декларация създава тип клас наречен *myclass* които съдържа две *private* променливи *a* и *b*. Той също съдържа две *public* функции наречени *setab()* и *showab()*. Фрагмента също декларира два обекта от тип *myclass* наречени *ob1* и *ob2*. За да позволите на елемент от класа да бъде наследен, но иначе да бъде *private*, определете го чрез *protected*. *Protected* члена се съдържа във производния клас но е *private* в неговата класова йерархия. Когато оперирате със обект от клас използвайте оператора точка(.) за обръщане към индивидуалните членове. Оператора стрелка(->) се използва когато достъпвате обект чрез указател. Например следващия код достъпва функцията *putinfo()* от *ob* използвайки оператор точка и функцията *show()* използвайки оператор стрелка:

```
struct cl_type{

    int x;

    float f;

    public:

    void putinfo ( int a, float t) {x= a; f= t;}

    void show () {cout<< a <<' '<<f<<endl;}

};

cl_type ob, *p;

//...

ob.putinfo(10, 0.23);

p = & ob; // слага адреса на ob в p

p->show(); // показва данните във ob
```

Възможно е да създадете шаблонни класове използвайки ключовата дума *template* (Виж “*template*” във “Справочника на ключовите думи”)

Наследяване

В C++ , един клас може да наследи характеристиките на друг .Наследения клас е обикновено наречен базов клас .Наследяващия клас е представен като производен клас. Когато един клас наследява друг се формира класова йерархия .Общата форма за наследяване на клас е:

```
class име_наклас : достъп име_на_базов_клас {
    // ....
};
```

Тук „достъп“ определя как се наследява базовия клас и той трябва да бъде или *private*, *public* или *protected* .(Той също може да бъде пропуснат и в този случай се приема че е *public* ако базовия клас е *struct* , или *private* ако базовия клас е *class*. За да наследите повече от един клас , използвайте разделен със запетайки списък. Ако достъп е *public* , всички *public* и *private* членове от базовия клас стават *public* и *protected* членове на производния клас .Ако достъп е *private* , всички *public* и *protected* членове на базовия клас стават *protected* членове на производния клас. В следващата класова йерархия *derived* наследява *base* като *private* . Това значи че *i* става *private* член на *derived*:

```
class base{
    public:
    int i ;
};

class derived : private base{
    int j;
    public:
    derived (int a) {j=i=a;}
    int getj () { return j ; }
    int geti () { return i ; } // OK derived има достъп до i
};

derived ob(9); //създава обект derived
```

```
cout << ob.geti () << ob.getj () ;
// ob.i = 10 ; // ГРЕШКА i е private за derived
```

Структури

Структура се създава чрез използване на ключовата дума **struct**. В C++ структура също дефинира клас. Единствената разлика между **class** и **struct** е че по подразбиране всички членове на структурата са **public**. За да направите член **private**, вие трябва да използвате ключовата дума **private**. Общата форма за деклариране на структура е тази :

```
struct име_на_структура : списък с наследяване {
    // public членове по подразбиране
    protected:
    // private членове които се наследяват
    private:
    // private членове
} списък- със- обекти;
```

В C, няколко ограничения са приложени за структурите. Първо те могат съдържат само членове-данни ; член-функции не са разрешени. C-структурите не поддържат наследяване. Също всички членове са **public** и ключовите думи **private** и **protected** не се използват.

Обединения

Обединение е тип клас в които всички членове- данни споделят едно и също място в паметта. В C++ , обединението може да включва и член-функции и данни. Всички членове на обединението са **public** по подразбиране . За да създадете **private** елементи, вие трябва да използвате ключовата дума **private**. Общата форма за деклариране на обединение е:

```
union име_на_клас{
    // public членове по подразбиране
    private:
```

```
// private членове
```

```
} списък с обекти;
```

В C, обединенията могат да съдържат само членове-данни и ключовата дума *private* не се поддържа. Елементите на обединението се припокриват. Например:

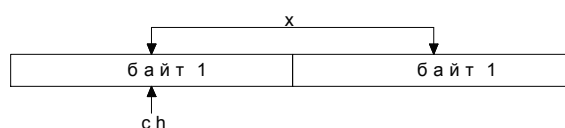
```
union tom{
```

```
char ch;
```

```
int x;
```

```
} t;
```

декларира обединение *tom*, което изглежда така в паметта (приемаме 2-байтово цяло число)



Като класа отделните променливи които съставят обединението се достъпват чрез използване на оператора точка. Оператора стрелка се използва с указател към обединение. Има няколко ограничения които се прилагат за обединенията. Първо обединение не може да наследи никакъв друг клас от никакъв тип. Обединение не може да бъде базов клас. Обединение не може да има виртуални член-функции. Никакви членове не могат да бъдат декларирани като *static*. Обединение не може да има като член обект който предефинира оператора *=*. Накрая никакъв обект не може да бъде член на обединение ако класа на обекта точно определя конструктор или деструктор функции (обектите които имат само подразбиращи се конструктори и деструктори са разрешени). Има специален тип обединение в C++ наречено анонимно обединение. Декларацията на анонимното обединение не съдържа име на клас и никакви обекти от това обединение не са декларирани. Вместо това анонимното обединение просто казва на компилатора че неговите член-променливи споделят едно и също място в паметта. Обаче обръщането към променливите става директно, без използване на нормалния синтаксис със операторите точка и стрелка. Променливите които правят анонимното обединение са на същото ниво на видимост като всяка друга променлива декларирана в същия блок. Това предполага че имената на променливите в обединението не трябва да

влизат в конфликт със никакви други имена които са валидни в техния обseg. Тук е пример за анонимно обединение:

```
union { // анонимно обединение

int a;

float f;

};

a = 10;

cout << f;
```

Тук *a* и *f* споделят едно и също място в паметта. Както можете да видите, имената на променливите в обединението се пишат директно без използване на операторите точка или стрелка. Всички ограничения които се отнасят за обединенията се напълно отнасят и за анонимните обединения. В добавка, анонимните обединения трябва да съдържат само данни- никакви член-функции не са позволени. Анонимните обединения не могат да съдържат ключовите думи **private** или **protected**. Накрая анонимно обединение в обхвата на именовано пространство трябва да бъде декларирано като **static**.

Програмен съвет

В C++ е общо приета практика да се използва **struct** когато създаваме структури в C-стил които включват само членове-данни. **Class** обикновено е предназначено за създаване на класове които съдържат член-функции. Понякога акронима **POD** се използва да означава C-стилова структура. **POD** означава Обикновенни Стари Данни.

Изброявания

Друг тип променлива която може да се създаде е наречена изброяване. Изброяването е списък от именовани целочислени константи. Така типа изброяване е просто спецификация на списък от имена които принадлежат на изброяването. Създаването на изброяване изисква използване на ключовата дума **enum**. Общата форма на тип изброяване е:

```
enum име_на_изброяването { списък с имена}
списък с променливи;
```

Име_на_изброяването е името на типа изброяване. Списъка от имена е разделен със запетайки. Например следващия фрагмент дефинира изброяване от

градове наречено *cities* и променлива *c* от тип *cities*. Накрая на нея се присвоява стойността "*Houston*":

```
enum cities {Houston , Austin , Amarillo} c ;
```

```
c = Houston ;
```

В изброяването стойността на първото (най-отляво) име е по подразбиране 0, второто име има стойност 1, третото има стойност 2 и т. н. Въобще, всяко име има стойност със 1 по-голяма от предшестващото го. Вие можете да дадете на име определена стойност чрез добавяне на инициализация. Например в следващото изброяване *Austin* ще има стойност 10:

```
enum cities {Houston, Austin =10 ,Amarillo} ;
```

В този пример *Amarillo* ще има стойност 11 понеже всяко име ще бъде със 1 по-голямо от това пред него. В C++ обект изброяване може да присвоява само онези стойности които са дефинирани в неговия тип изброяване. В C на обект изброяване може да бъде присвоена всякаква целочислена стойност.

C тагове

В C имената на структурите, обединенията и изброяванията не дефинират пълно име на тип. В C++ това е така. Например следващия фрагмент е валиден за C++ , но не и за C :

```
struct s_type {
```

```
int i;
```

```
double d;
```

```
};
```

```
// ...
```

```
s_type x ; // ОК за C++ , но не и за C
```

В C++, *s_type* дефинира пълно име на тип и може да бъде използвано само за деклариране на обекти. В C ,*s_type* само дефинира таг. В C вие трябва да поставите пред тага или *struct*, *union* или *enum* когато декларирате обекти. Например:

```
struct s_type x; // сега е ОК за C
```

Този синтаксис е също разрешен в C++ но е рядко използван.

Спецификатори на класове за съхранение

Спецификаторите за съхраняване *extern*, *auto*, *register*, *static* и *mutable* се използват за променяне на начина по който C/C++ съхранява променливите. Тези спецификатори предхождат името на типа който модифицират.

extern

Ако спецификатора *extern* е поставен преди име на променливата , компилатора ще знае че променливата има вътрешно свързване. Външното свързване значи че обекта е видим извън неговия файл. Всъщност *extern* казва на компилатора типа на променливата без в действителност да заделя съхранител за нея. Модификатора *extern* е най-често използван когато имаме два или повече файла споделящи еднакви глобални променливи.

auto

Спецификатора *auto* казва на компилатора че локалната променлива която следва е създадена при влизането в блок и се разрушава при излизането от блок. Откакто всички променливи дефинирани вътре във функциите са *auto* по подразбиране , ключовата дума *auto* е рядко (ако дори) използвана.

register

Когато беше създаден C *register* можеше само да бъде използван върху локални целочислени или символни променливи понеже той беше причина компилатора да направи опит да пази променливата в регистрите на CPU вместо да ги поставя в паметта . Това правеше всички извиквания на променливата извънредно бързи. Дефиницията на *register* оттогава беше разширена .Сега всяка променлива може да бъде определена като *register* и е работа на компилатора да оптимизира достъпа до нея. За символите и целочислените , това още означава използване на кеш паметта ,например. Запомнете че *register* е само заявка. Компилатора е свободен да я отхвърли. Причината за това е че ограничен брой променливи могат да бъдат оптимизирани за скорост. Когато този лимит бъде надвишен, компилатора просто игнорира по нататъшните *register* заявки.

static

Спецификатора *static* инструктира компилатора да пази променлива в съществуване през цялото действие на програмата вместо да я създава и разрушава всеки път когато влиза и излиза от обсега. По този начин правейки локални променливи *static* позволяваме да поддържат техните стойности между извикванията на функциите .Модификатора *static* може да бъде приложен към глобални променливи .Когато това е направено , то е причина за ограничаване обхвата на променлива във файла в който е декларирана .Това значи че тя ще има вътрешно свързване . Вътрешно

свързване значи че идентификатора е известен само във неговия собствен файл. В C++ ,когато **static** е използвана върху членовете-даннина клас причинява само едно копие от този член да бъде разделен между всички обекти от този клас.

mutable

Спецификатора **mutable** е приложен само в C++. Той позволява на член на обект да предефинира константността. Така **mutable** член на **const** обект не е **const** и може да бъде изменян.

Типови определители

Типовите определители **const** и **volatile** доставят допълнителна информация за променливите които предхождат.

const

Обектите от тип **const** не могат да бъдат променени от вашата програма през изпълнението ѝ . Също обект указван чрез **const** указател не може да бъде изменян . Компилятора е свободен да постави променливите от този тип в ROM паметта. **const** променлива ще получи стойност или от изрична инициализация или чрез някакъв хардуерно- зависим начин. Например :

```
const int a = 10 ;
```

Ще създаде цяло число **a** със стойност 10 която не може да бъде изменена от вашата програма . Тя може обаче да бъде използвана обаче в други типове изрази.

Програмен съвет

Ако класова член функция е модифицирана със **const** тя не може да промени обекта който извиква функцията . За деклариране на **const** член функция сложете **const** след нейния списък с параметри. Например:

```
class MyClass{  
  
int i;  
  
public:  
  
// функция const  
  
void wrong (int a) const {  
  
i = a ; // Грешка! Не можете да изменяте извикващия обект
```

```

}

void right (int a) {

i = a; //OK, не е const функция

}

};

```

както подсказват коментарите, *wrong()* е *const* функция и тя не може да измени обекта който я извиква.

volatile

Модификатора *volatile* казва на компилатора че стойностите на променлива могат да бъдат променени по начини не точно определени от програмата. Например адреса на глобална променлива може да бъде подаден по часовников начин от операционната система и да се изменя със всеки цикъл на часовника. В тази ситуация съдържанието на променливата се променя без никакъв ясен присвояващ израз в програмата. Това е важно понеже компилаторите понякога автоматично оптимизират определени изрази допускайки че съдържанията на променливи са непроменени вътре в израза. Това е направено за постигане на по-висока производителност. Модификатора *volatile* ще предотврати тази оптимизация в онези редки ситуации в които това предположение е вярно.

Масиви

Вие може да декларирате масиви от всякакъв тип данни. Общата форма на едномерен масив е:

тип име_на_променлива [размер] ;

Където **тип** определя типа данни на всеки елемент в масива и **размер** определя броя на елементите в масива. Например за да декларирате целочислен масив *x* от 100 елемента вие ще трябва да напишете:

int x[100];

Това ще създаде масив който е 100 елемента дълъг със първи елемент 0 и последен 99. Например следващия цикъл ще зареди числата от 0 до 99 в масива *x*:

```

for ( t = 0; t < 100 ; t++)
    x [t] = t;

```

Вие може да декларирате масиви от всеки валиден тип данни, включително класовете които създавате. Многомерни масиви се декларират чрез поставяне на допълнителни размери вътре в допълнителни скоби. Например за деклариране на 10x20 масив вие трябва да напишете :

int x[10][20];

Масивите могат да бъдат инициализирани чрез използване на заграден в скоби списък от инициализатори. Например :

int count [5] = {1,2,3,4,5 };

Дефиниране на нови типове имена използвайки typedef

Вие може да създадете ново име за съществуващ тип използвайки ***typedef***. Неговата обща форма е :

typedef тип ново_име ;

Например следния фрагмент от код казва на компилатора че ***feet*** е друго име за ***int***:

typedef int feet;

Сега следващата декларация е напълно позволена и създава целочислена променлива наречена ***distance***:

feet distance ;

Константи

Константите също наречени литерали се отнасят за фиксирани стойности които не могат да бъдат променени от програмата. Константи могат да бъдат всякакви основни типове данни. Начина по който се представя всяка константа зависи от нейния тип. Символните константи са затворени между единични кавички. Например '*a*' и '*t*' са две символни константи. Целочислените константи се определят като числа без дробни части. Например 10 и -100 са целочислени константи. Константите със плаваща точка изискват употребата на десетична точка следвана от дробната част на числото. Например 11.123 е константа със плаваща точка. Вие също можете да използвате научно означение на числата със плаваща точка. Има два типа със плаваща точка : ***float*** и ***double***. Също има няколко вида от основните типове които се получават чрез използване на модификаторите за тип. По подразбиране компилатора поставя числовите константи във най- малкия съвместим тип данни който ще ги съдържа. Единственото изключение от правилото за най- малкия тип са константите със плаваща точка , които се приемат да са от тип ***double***. За повечето програми подразбирането на компилатора е напълно достатъчно . Обаче е възможно да определите точно типа на константата която искате. За определяне точния тип на числова константа използвайте суфикс. За типовете със плаваща точка ако сложите след числото ***F*** то ще се третира като ***float***. Ако го последвате със ***L*** числото става ***long double***. За целочислените типове суфикса ***U*** се разбира за ***unsigned*** и ***L*** за ***long***. Няколко примера са показани тук:

тип данни	Примери за константи
<i>int</i>	1 123 21000 - 234
<i>long int</i>	35000 <i>L</i> -34 <i>L</i>
<i>unsigned int</i>	100000 <i>U</i> 987 <i>U</i>
<i>float</i>	123.23 <i>F</i> 4.34e-3 <i>F</i>

<i>double</i>	123.23 -0.987324
<i>long double</i>	1001.2L

Шестнайсетични и осмични константи

Понякога е по-лесно да използвате числова система базирана на 8 или 16 вместо 10. Числовата система базирана на 8 е наречена осмична и използва цифрите от 0 до 7. В осмичната числото 10 е същото като 8 в десетичната. Базираната на 16 числова система е наречена шестнайсетична и използва цифрите от 0 до 9 плюс буквите от **A** до **F** които означават 10, 11, 12, 13, 14 и 15. Например шестнайсетичното число 10 е 16 в десетичната. Понеже често тези две числови системи се използват C/C++ ви позволява да определите целочислени константи в шестнайсетична или осмична вместо в десетична ако предпочитате. Шестнайсетичните константи трябва да започват със **0** (нула следвана от **x**) или **0X** следвана от константата във шестнайсетична форма. Осмична константа започва с **0**. Тук са два примера:

```
int hex = 0x80; //128 в десетична
int oct = 012; //10 в десетична
```

Низови константи

C/C++ поддържа един друг тип константа в добавка към тези предефинирани типове данни: низ. Низа е множество от символи затворено в двойни кавички(" "). Например *"this is a test"* е низ. Вие не трябва да бъркате низовете със символите. Единична символна константа е затворена между единични кавички такава като *'a'*. Обаче *"a"* е низ съдържащ само една буква. Низовите константи се автоматично терминират със **0** от компилатора. C++ също поддържа и клас *string*, който е описан по-късно в тази книга.

Булеви константи

C++ определя две булеви константи: *true* и *false*.

Константи след обратно наклонена черта

Заграждането на символни константи в единични кавички работи за повечето печатни символи, но някой такива като връщане на каретката е невъзможно да бъдат въведени във вашия програмен сорс код от клавиатурата. По тази причина C/C++ разпознава няколко константи след обратно наклонена черта също наречени **escape** последователности. Тези константи са изброени тук:

Код	Значение
\b	Backspace
\f	Връщане в началото на реда (formfeed)
\n	Нов ред
\r	Връщане на каретката
\t	Хоризонтална табулация
\"	Двойни кавички
\'	Единични кавички
\\	Обратно наклонена черта
\v	Вертикална табулация
\a	Сигнал от високоговорителя

<code>\N</code>	Осмична константа(където N е осмичната константа)
<code>\xN</code>	Шестнайсетична константа (където N е шестнайсетичната константа)
<code>\?</code>	Знак ?

Константите след обратно наклонената черта могат да бъдат използвани навсякъде където могат и символите . Например следващия израз извежда нов ред и табулация и тогава отпечата низа “*This is a test*”:

```
cout << “\n \t This is a test”;
```

ГЛАВА 2 - Функции , интервали , именования пространства и хедъри

Функциите са изграждащите блокове на C/C++ програмите. Елементите от програма включително функциите съществуват в един или повече интервала. В C++ има специален интервал наречен именовано пространство . Прототипите за всички стандартни функции са декларираны във различни хедъри. Тези теми са разгледани тук.

Функции

Същината на C/C++ програмата е функцията. Тя е мястото в което се извършва цялата дейност на програмата. Общата форма на функция е :

```
връщан-тип име_на_функция(списък с параметри)
{
тяло на функцията
}
```

Типа на върнатите данни е определен от **връщан -тип**. **Списък с параметри** е разделен със запетайки списък от променливи които ще получават някакви аргументи подадени на функцията. Например следващата функция има два целочислени параметъра *i* и *j* и *double* параметър наречен *count* :

```
void f(int i , int j , double count)
{...
```

Забележете че трябва да декларираме всеки параметър отделно. В C++ ако функцията няма параметри тогава нейния списък с параметри е празен. В C за определяне на празен списък с параметри използвайте думата *void*. Тук е пример :

```
int f( void )
{...
```

Тази употреба на *void* в C++ е разрешена но е излишна. В C ако връщания тип не е точно определен той по подразбиране е *int*. Стандартния C++ не поддържа подразбиране за *int* но повечето C++ компилатори го разрешават. Функцията прекъсва и се връща автоматично на извикващата процедура когато се срещне последната фигурна скоба. Вие можете да направите по –

ранно връщане със използване на израза **return**. Всички функции с изключение на онези декларирани като **void**, връщат стойност. Типа на връщаната стойност трябва да отговаря на типа деклариран от функцията. Ако израза **return** е част от не-**void** функция, връщаната стойност от функцията е стойността в израза **return**. В C++ е възможно създаването на общи функции използвайки ключовата дума **template** (Виж "**template**" в "Справочника на ключовите думи")

Рекурсия

В C/C++ функциите могат да извикват себе си. Това е наречено рекурсия и функцията която извиква себе си се нарича рекурсивна. Прост пример функцията **factr()** показана тук която изчислява факториела на цяло число. Факториела на числото **N** е произведението от всички числа от 1 до **N**. Например 3 факториел е $1 \times 2 \times 3$ или 6.

```
// Изчислява факториела на число използвайки рекурсия
int factr (int n)
{
    int answer;
    if (n==1) return 1;
    answer = factr (n - 1) *1;
    return answer ;
}
```

Когато **factr ()** е извикана с аргумент 1 функцията връща 1, иначе тя връща произведението **factr (n - 1) * n**. За изчисляването на този израз **factr ()** е извикана със **n - 1**. Този процес продължава докато **n** стане равна на 1 и извикванията към функцията започнат връщане. Когато **factr ()** накрая върне първоначалното извикване, крайната върната стойност ще бъде факториела на първоначалния аргумент. Когато функция извиква себе си, нови локални променливи и параметри заделят място в стека и кода на функцията се изпълнява със тези нови променливи от неговото начало. Рекурсивното извикване не прави ново копие на функцията. Само аргументите са нови. Когато всяка рекурсия се върне старите локални променливи и параметри се премахват от стека и изпълнението се подновява от точката на рекурсивното извикване вътре във функцията. Рекурсивните функции може да се каже че излизат навън и се връщат обратно.

Програмен съвет

Рекурсивните функции трябва да бъдат използвани внимателно. Рекурсивните версии на много методи се изпълняват малко по бавно от техните итеративни еквиваленти поради прибавянето отгоре на повтарящи се извиквания на функции. Много рекурсивни извиквания към функции могат да причинят препълване на стека. Понеже съхранението на параметрите на функцията и локалните променливи е в стека и всяко ново извикване създава копие на тези променливи във него и мястото се изчерпва и ако това стане се

получава препълване на стека . Ако това се случи в нормално използване на дебъгната рекурсивна функция опитайте да увеличите заделянето на място в стека за вашата програма . Когато пишете рекурсивни функции вие трябва да включите условен израз някъде който предизвиква връщане от функцията без изпълнение на рекурсивното извикване . Ако не го направите веднъж щом извикате функцията тя ще извика себе си докато не се изчерпи стека . Това е много честа грешка когато разработвате рекурсивни функции . Използвайте щедро изходни изрази при разработката и така вие може да видите какво е започнало и да прекъснете изпълнението ако видите че сте направили грешка.

Предефиниране на функции

В C++ функциите могат да бъдат предефинирани . Когато е предефинирана функция , две или повече функции споделят еднакво име . Обаче всяка версия на предефинираната функция трябва да има различен брой или тип параметри (връщаните типове на функцията могат също да се различават но това не е необходимо) . Когато се извика предефинирана функция , компилатора решава коя версия да използва базирайки се на типа и броя на аргументите извиквайки функцията имаща най – близко съвпадение . Например имайки тези три предефинирани функции:

```
void myfunc ( int a ) {
cout<< "a is " << a << endl;
}
// предефинирана myfunc
void myfunc ( int a, int b ) {
cout<< "a is " << a << endl;
cout<< "b is " << b << endl;
}
// предефинирана отново myfunc
void myfunc ( int a, double b ) {
cout <<"a is " << a <<endl ;
cout <<"b is " << b << endl ;
}
```

следните извиквания са разрешени :

```
myfunc (10 ) ; // извиква myfunc ( int )
myfunc ( 12 , 24 ) ; // извиква myfunc ( int , int )
myfunc ( 99 , 123.23 ) ; // извиква myfunc ( int , double )
```

Във всички случай типа и броя на аргументите определя коя версия на **myfunc ()** е в действителност изпълнена . Предефинирането на функции не се поддържа в C .

Подразбиращи се аргументи

В C++ вие можете да присвоите на параметър на функция подразбираща се стойност , която ще бъде използвана автоматично когато няма съответен аргумент определен при извикването на функцията . Подразбиращата се

стойност е определена по начин синтактично подобен на инициализацията на променлива. Например тази функция присвоява на нейните два параметъра подразбиращи се стойности:

```
void myfunc ( int a = 0 , int b = 10 )  
{ //...
```

Давайки подразбиращи се аргументи , **myfunc ()** може да бъде легално извикана по тези три начина :

```
myfunc ( ~ ); // a по подразбиране 0;  
                // b по подразбиране 10  
myfunc (-1 ); // на a се подава -1; b по подразбиране 10  
myfunc ( -1 , 99 ); //на a се подава -1; b е 99
```

Когато създавате функции които имат подразбиращи се аргументи , вие може да определите подразбирещите се стойности само веднъж , или във прототипа на функцията или във нейната дефиниция (вие не можете да ги определите на всяко място , дори ако използвате еднакви стойности). Обикновено подразбиращите се стойности се определят във прототипа. Когато давате на функцията подразбиращи се аргументи запомнете че трябва да определите всички подразбиращи се аргумент първи. Веднъж щом започнете да определяте подразбиращи се аргументи , не можете да ги смесвате със неподразбиращи се. Подразбиращите се аргументи не се поддържат от C .

Прототипи

В C++ всички функции трябва да имат прототип. В C прототипите са технически незадължителни , но са силно препоръчвани. Общата форма на прототип е показана тук :

връщан- тип име (списък с параметри) ;

Всъщност прототипа е просто връщания тип , името и списъка с параметри на дефиницията на функцията следван от точка и запетайка. Следващия пример показва прототипа на функцията **fn ()** :

```
float fn ( float x ) ; // прототип  
.  
.  
.  
// дефиниция на функцията  
float fn ( float x ) ;  
{  
//...  
}
```

За определяне прототипа на функция приемаща променлив брой аргументи , използвайте три последователни точки където започва променливия брой параметри. Например функцията **printf ()** може да има следния прототип:

int printf (const char *format , ...) ;

Когато определяте прототип за предефинирана функция всякаверсия на тази функция трябва да има свой прототип. Когато член функция е декларирана в своя клас , това представлява прототипа за функцията. В C за да определите

прототип на функция която няма параметри използвайте **void** в нейния списък с параметри .

Програмен съвет

Два термина са обикновено обърквани в програмирането на C/C++ : декларация и дефиниция . Тук е обяснено какво означават . Декларацията определя името и типа на обект . Дефиницията заделя място за съхраняването му . Това се отнася и за функциите . Декларирането на функция (прототип) определя връщания тип , името и параметрите на функцията . Самата функция (тоест функцията със нейното тяло) е нейната дефиниция . В много случай декларацията е също и дефиниция . Например когато не – **extern** променлива е декларирана тя е също и дефинирана . Или когато е дефинирана по – рано функция за употреба , нейната дефиниция също служи като декларация .

Разбиране на интервалите и продължителността на съществуване на променлива

C и C++ определят правила на интервала които управляват видимостта и продължителността на живот на обектите . Макар че има няколко тънкости в повечето случай има два интервала : глобален и локален . Глобалния интервал съществува извън всички други интервали . Име декларирано в глобалния интервал е известно навсякъде във програмата . Например глобална променлива е на разположение за употреба от всички функции в програмата . Глобалните променливи съществуват през цялото времетраене на програмата . Локалния интервал е дефиниран чрез блок . Така локалния интервал е започнал при отварящата фигурна скоба и завършва при затварящата скоба . Име декларирано във локален интервал е познато само в този интервал . Понеже блоковете могат да бъдат вложени , локалните интервали също могат да бъдат вложени . Разбира се най- общия локален интервал е този дефиниран чрез функция . Локалните променливи се създават когато техния блок е започнал и се разрушават когато техния блок е напуснат . Това значи че локалните променливи не държат техните стойности между извикванията на функцията . Вие обаче може да използвате модификатора **static** да запазите стойностите между извикванията . В C++ локалните променливи могат да бъдат декларирани почти навсякъде във блока . В C те трябва да бъдат декларирани в началото на блока , преди всеки действащ израз да се изпълни . Например следния код е валиден за C++ , но не и за C :

```
void f(int a)
{
    int a ;
    a = 10 ;
    int b ; // ОК за C++, но не и за C
```

Глобална променлива трябва да бъде декларирана извън всички функции включително и функцията **main ()**. Глобалните променливи се обикновено поставят на върха на файла преди **main ()**, за улесняване на четенето и понеже променлива трябва да бъде декларирана преди да се използва. Формалните параметри на функция са също локални променливи и като изключим тяхната работа при получаването на стойността на от извикващите аргументи, се държат и могат да бъдат използвани както всяка друга локална променлива.

Именовани пространства

В C++ е възможно създаването на локален интервал използвайки ключовата дума **namespace**. Именованото пространство определя декларативен регион. Неговото предназначение е да локализира имената. Общата форма на **namespace** е показана тук:

```
namespace име {
    //...
}
```

Тук име е името на именованото пространство. Това е пример:

```
namespace MyNameSpace {
    int count;
}
```

Това създава именовано пространство наречено **MyNameSpace** и променлива **count** декларирана във него. Имена декларирани във именовано пространство могат да бъдат насочени точно към други изрази в същото именовано пространство. Извън тяхното именовано пространство, имената могат да бъдат достигнати по два начина. Първо вие може да използвате оператора за включване на интервал. Например приемайки **MyNameSpace** като горния, следващия израз е валиден:

```
MyNameSpace :: count = 10;
```

Вие можете също да определите израз **using**, който внася определено име или именовано пространство в текущия интервал. Това е пример:

```
using namespace MyNameSpace;
count = 100;
```

В този случай **count** може да бъде посочено точно понеже е било внесено във текущия интервал. Традиционно заглавията декларирани в C++ библиотеката бяха във глобалното (тоест неименовано) пространство. Обаче сегашната спецификация за C++ слага всичките тези заглавия във именованото пространство **std**.

Функцията main ()

В C/C++ програма изпълнението започва от **main ()** (Windows програмите извикват **WinMain ()**, но това е специален случай). Вие не трябва да имате повече от една функция наречена **main ()**. Когато **main ()** се прекъсне, програмата приключва и контрола се подава обратно на операционната

система . Функцията **main ()** няма прототип. Така различни форми на **main ()** могат да бъдат използвани . И за C и за C++ следващите версии на **main ()** са валидни (други форми са също позволени) :

```
int main ( )
int main (int argc , char *argv[ ] )
```

Както показва втората форма , най – малко два параметъра се поддържат от **main ()** . Това са **argc** и **argv** (някой компилатори ще разрешат и допълнителни параметри) . Тези две променливи ще съдържат броя на аргументите от командния ред и указател към тях съответно . Параметъра **argc** е цяло число и неговата стойност ще бъде най - малко 1 понеже името на програмата е първи аргумент що се касае за C/C++. Параметъра **argv** трябва да бъде деклариран като масив от символни указатели . Всеки указател сочи към аргумент от командния ред . Тяхната употреба е показана по – надолу в къса програма която ще отпечати вашето име на екрана :

```
#include <iostream>
using namespace std ;
int main ( int argc, char *argv[ ] )
{
if (argc < 2 )
cout << "Enter your name . \n " ;
else
cout << "hello " << argv[ ] ;
return 0;
}
```

Аргументи на функция

Ако функция използва аргументи тя трябва да декларира променливи които приемат стойностите на аргументите . Тези променливи са наречени формални параметри на функцията . Те се държат като всяка друга локална променлива вътре във функцията и се създават при влизане във функцията и се разрушават при изход от нея . Както и с локалните променливи , вие можете да правите присвоявания за формалните параметри на функцията или да ги използвате във всеки разрешен C/C++ израз . Дори макар тези променливи да изпълняват специална задача да получават стойност от аргументите подадени на функцията , те могат да бъдат използвани като всяка друга локална променлива . Най – общо предаването на аргументи е по два начина . Първия е наречен извикване по стойност . Този метод копира стойността на аргумент във формалния параметър на функцията . Промените направени върху параметрите на функцията нямат ефект на променливите използвани за извикването и . Втория начин за подаване на параметри на функция е означен като извикване по адрес . В този метод , адреса на аргумента се копира в параметъра . Вътре във функцията се използва за достъп до действителния аргумент използван в извикването . Това значи че промените направени върху параметъра се отразяват върху променливата използвана за извикването на функцията . По подразбиране C и C++ използват извикване по стойност за

подаване на аргументи . Това значи че във функцията вие обикновено не можете да промените променливите използвани за извикване на функцията . Разгледайте следната функция :

```
int  sqr ( int x )
{
  x = x * x ;
  return x ;
}
```

В този пример , когато се извърши присвояването $x = x * x$, само се променя локалната променлива x . Аргумента използван да извика **sqr ()** още има неговата първоначална стойност . Запомнете че само копие на стойността на аргумента се подава на функцията . Какво се случва вътре във функцията няма ефект върху променливата използвана при извикването .

Подаване на указатели

Дори макар че C/C++ използват извикване по стойност за подаването на параметри по подразбиране , възможно е ръчно да конструирате извикване по адрес чрез подаване на указател към аргумента . Понеже това подава адреса на аргумента на функцията , тогава е възможно да промените стойността на аргумента извън функцията . Указатели се подават на функциите като всяка друга стойност . Разбира се необходимо е да декларирате параметрите като типове указатели . Например функцията **swap ()** , която разменя стойностите на нейните два целочислени аргумента следва :

```
//използва параметри указатели
void swap ( int * x , int * y )
{
  int temp;
  temp = * x ; // запазва стойността на адреса x
  * x = * y ; // слага y във x
  * y = temp ; // слага x във y
}
```

Важно е да запомните че **swap ()** (или всяка друга функция която използва параметри указатели) трябва да бъде извикана със адреса на аргументите . Следващия фрагмент показва правилния начин за извикване на **swap ()** :

```
int  a , b ;
A = 10 ;
B = 20 ;
swap ( & a , & b );
```

В този пример , **swap ()** е извикана със адреса на **a** и **b** . Унарния оператор **&** се използва да получи адреса на променливите . По този начин адресите на **a** и **b** , а не техните стойности се подават на функцията **swap ()** . След извикването , **a** ще има стойност 20 а **b** ще има стойност 10 .

Указател към функция.

Пример.

```
void Funct(CString InputValue);  
void (*Funct1)(CString);
```



```
void main()  
{  
    Funct1 = Funct;           //Сега Funct1 сочи Funct  
    Funct1("Test");          //Извиква Funct  
    или  
    (*Funct1)("test");        // Също извиква Funct1  
}
```

Указателя към функция може да се инициализира и при декларирането му.

```
void (*Funct1)(CString)= &Funct;  Кое е аналогично на void  
(*Funct1)(CString)= Funct;
```

върщан тип:

```
int (*test)(int input); //Указател към функция тип int  
CString Funct(int input);
```

```
test = Funct; // Това е Грешка, върнатия тип не отговаря.
```

Правилата за предаване на аргументи са същите като при директно извикване на функция.

Пример за дефиниране на тип

```
typedef void (*SYG_TYPE)(int);  
typedef void (*SYG_ARG_TYP)(int);  
SYG_TYPE signal(int, SYG_ARG_TYP);
```

Параметри псевдоними

В C++ е възможно автоматично да подадете адрес на променлива към функция. Това се осъществява чрез използването на параметър псевдоним. Когато използвате параметър псевдоним, адреса на аргумента се подава на функцията и тя оперира със аргумента а не със копие. За създаване на параметър псевдоним, сложете пред неговото име **&** (амперсанд). Вътре във функцията можете да използвате параметъра нормално без никаква нужда от употреба на оператора ***** (астерикс). Компилятора автоматично ще прехвърли адреса за вас. Например следващия код създава версия на **swap()** която използва два параметъра псевдоними за разменяне стойностите на нейните два аргумента :

```
//използва параметри псевдоними
void swap (int &x, int &y)
{
    int temp;
    temp = x ; //запазва стойността на адреса x
    x = y ; //слага y във x
    y = temp ; //слага x във y
}
```

Когато извиквате **swap()**, вие просто използвайте нормалния синтаксис на извикване. Това е пример :

```
int a, b;
A = 10;
B = 20;
swap (a, b);
```

Понеже сега **x** и **y** са параметри псевдоними, адресите на **a** и **b** се генерират автоматично и подават на функцията. Вътре във функцията имената на параметрите се използват без никаква нужда от оператора ***** понеже компилатора автоматично се отнася към извикващите аргументи всеки път когато се използват **x** и **y**. Параметрите псевдоними се използват само във C++.

Конструктори и деструктори

В C++, клас може да съдържа функция конструктор, функция деструктор или и двете. Конструктора се извиква когато се създава обект от класа, а деструктора се извиква когато се разрушава обект от класа. Конструктора има еднакво име като класа на който е член а името на деструктора е същото като името на неговия клас с изключение на това че пред него има **~** (тилда). Нито конструктора, нито деструктора имат връщана стойност. Функциите конструктори могат да имат параметри. Вие може да използвате тези параметри да подадете стойност към конструктора, която може да бъде използвана за инициализацията на обект. Аргументите които са подадени към параметрите се определят когато се създава всеки обект. Например този фрагмент илюстрира как да подадете аргумент на конструктора :

```

class myclass {
    int a;
public:
    myclass ( int i ) { a = i ; } //конструктор
    ~myclass ( ) { cout << "Destructing ... " ; }
};
//...
myclass ob ( 3 ) ; //подава 3 на i

```

Когато **ob** е деклариран стойността 3 се подава към параметъра на конструктора **i** който я присвоява на **a**.

Спецификатори на функцията

C++ дефинира три спецификатора за функция: **inline**, **virtual** и **explicit**. Спецификатора **inline** е заявка към компилатора да разшири кода на функцията в реда вместо да я извиква. Ако компилатора не може да постави функцията на реда, той е свободен да отхвърли заявката. И член и не-член функции могат да бъдат определени като **inline**. **Virtual** функция се дефинира в базовия клас и се предефинира от производния клас. Виртуалните функции са начина по който C++ поддържа полиморфизъм. Спецификатора **explicit** се прилага само към конструкторите. Конструктор определен като **explicit** ще бъде само използван когато инициализацията точно се прилага както определя конструктора. Никакво автоматично преобразуване не става (тя създава "непреобразуващ конструктор").

Свързваща спецификация

Понеже често свързвате C++ функция със функции генерирани от друг език (такъв като C), C++ позволява да определите свързваща спецификация която казва на компилатора как да свърже функцията. Тя има тази обща форма:

extern "език" прототип на функцията

Както можете да видите, свързващата спецификация е разширение на ключовата дума **extern**. Тук "език" означава езика за който искате да се свърже функцията. C и C++ свързване са гарантирани да се поддържат. Вашият компилатор може също да поддържа и други свързвания. За деклариране на няколко функции използващи една спецификация, вие може да използвате тази обща форма:

```

extern "език" {
    прототипи на функциите
}

```

Свързващата спецификация се поддържа само във C++ и я няма във C.

Стандартните библиотеки на C и C++

Нито C нито C++ имат ключови думи изпълняващи I/O , манипулиращи низове , изпълняват различни математически изчисления или изпълняват някакви други полезни процедури . Начина по който тези неща са изпълнени е чрез използване на множество от предефинирани библиотечни функции които са доставени със компилатора . Има два основни стила библиотеки :функционалната библиотека на C и C++ класовата библиотека , която е приложена само за C++ . И двете библиотеки са описани по – късно във този справочник . Преди вашата програма да може да използва библиотечна функция , тя трябва да включи подходящ хедър . За C програми , хедърите се определят използвайки техните файлови имена , които завършват на **.H** . Хедърните файлове са дефинирани от **ANSI C** стандарта както е показано тук:

C хедър файл

assert . h

ctype . h

errno . h

float . h

iso646 . h

limits . h

local . h

math . h

setjmp . h

signal . h

stdarg . h

stddef . h

stdio . h

stdlib . h

string . h

time . h

wctype . h

wchar . h

Поддържа

Макроса *assert ()*

Манипулиране на символи

Сведения за грешки

Зависещи от изпълнението числа със плаваща точка

Няколко макроса които могат да бъдат използвани на мястото на различни оператори , такива като **not** за ! или **xor** за ^

Различни зависещи от изпълнението лимити

Функцията *setlocale ()*

Различни дефиниции използвани за математическата библиотека

Нелокален скок

Сигнални стойности

Аргументен списък със променлива дължина

Общо употребявани константи

Файлов I/O

Разнообразни декларации

Низови функции

Системно време и функции за дата

Манипулиране със широки символи

Функции за широки символи

В модерната спецификация за C++ , хедърите се определят чрез използване на стандартни хедърни имена , които не завършват със **.h** . Тези хедъри в нов стил не определят имена на файлове . Вместо това те са просто стандартни идентификатори които компилатора може да манипулира както му е удобно . Това значи че хедъра може да бъде съпоставен на файлово име , но не е задължително . Новите C++ хедъри са показани тук . Тези отнасящи се до стандартната шаблонна библиотека (**STL**) са означени :

C++ хедър

<algorithm>
<bitset>
<complex>
<deque>
<exception>
<fstream>
<functional>
<iomanip>
<ios>
<iosfwd>
<iostream>
<istream>
<iterator>
<limits>
<list>
<locale>
<map>
<memory>
<new>
<numeric>
<ostream>
<queue>
<set>
<sstream>
<stack>
<stdexcept>
<streambuf>
<string>
<typeinfo>
<utility>
<valarray>
<vector>

Поддържа

Различни операции в контейнери (**STL**)
 Битсети (**STL**)
 Комплексни числа
 Опашки с два края (**STL**)
 Манипулиране на изключения
 Базиран на потоци файлове I/O
 Различни функции (**STL**)
 I/O манипулатори
 I/O класове от ниско ниво
 Съвременни декларации за I/O системата
 Стандартни I/O класове
 Входни потоци
 Достъп до съдържанието на контейнери (**STL**)
 Различни изпълнителни лимити
 Линейни списъци (**STL**)
 Локализационна информация
 Карти (ключове със стойности) (**STL**)
 Заделяне на памет (**STL**)
 Заделяне на памет чрез използване на **new**
 Числови операции с общо предназначение
 Изходни потоци
 Опашки (**STL**)
 Множества (**STL**)
 Низови потоци
 Стекове (**STL**)
 Стандартни изключения
 Буферирани потоци
 Стандартния клас **string** (**STL**)
 Информация за типа по време на изпълнение
 Шаблони с общо предназначение (**STL**)
 Операции върху масиви съдържащи стойности
 Вектори (динамични масиви) (**STL**)

C++ също дефинира следните хедъри в нов стил които съответстват на C хедърите :

<cassert>	<cctype>	<cerrno>
<cfloat>	<ciso646>	<climits>
<ctype>	<cmath>	<csetjmp>
<csignal>	<cstdlib>	<csignal>
<stdio>	<stdlib>	<string>
<time>	<wchar>	<wctype>

В стандартния C++ , цялата информация отнасяща се до стандартната библиотека е дефинирана в именованото пространство **std** . Така , за получаване на директен достъп до тези имена , вие ще трябва да включите следващия **using** израз след включването на необходимите хедъри .

using namespace std ;

Програмен съвет

Ако използвате стар C++ компилатор , тогава той може да не поддържа новите C++ хедъри или командата `namespace` . Ако случая е такъв , тогава вие трябва да използвате старите в традиционен стил хедъри . Те използват еднакви имена като новите хедъри но включват *.h* (така те приличат на C хедъри) . Например следващото включва *<iostream>* използвайки традиционно обръщане :

#include <iostream.h>

Когато използвате хедъри в традиционен стил , всички имена дефинирани от хедъра се поставят в глобалното именовано пространство , а не във това определено от *std* . Така не се изисква израз *using* .

ГЛАВА 3 - Оператори

C/C++ имат богато множество от оператори които могат да бъдат разделени в следните класове : аритметични , релационни и логически , побитови , указатели , присвояване , I/O и различни .

Аритметични оператори

C/C++ има следните седем аритметични оператора :

Оператор	Действие
-	Изваждане , унарен минус
+	Събиране
*	Умножение
/	Деление
%	Целочислено деление
--	Намаляване
++	Увеличаване

Операторите + , - , * и / работят по очаквания начин . Оператора % връща остатъка от целочислено деление . Увеличаващия и намаляващия оператори увеличават или намаляват операнда с единица . Тези оператори имат следния ред на приоритет :

Приоритет	Оператори
Най – висок	++ -- - (унарен минус) * / %
Най – нисък	+ -

Операторите от еднакво ниво на приоритет се изчисляват отляво надясно .

Релационни и логически оператори

Релационните и логическите оператори се използват за получаване на *true* / *false* резултати и са често използвани заедно . В C/C++ всяко ненулево число се изчислява като *true* .0 е *false* . В C++ резултата от релационните и

логическите оператори е от тип *bool*. В C, резултата е 0 или ненулево цяло число. Релационните оператори са изброени тук:

Оператор	Значение
>	По – голямо
>=	По – голямо или равно
<	По – малко
<=	По – малко или равно
==	Равно
!=	Неравно

Логическите оператори са показани тук:

Оператор	Значение
&&	AND
	OR
!	NOT

Релационните оператори се използват за сравнение на две стойности.

Логическите оператори се използват да свържат две стойности или в случая на ! да обърнат стойността. Приоритета на тези оператори е показан във следващата таблица:

Приоритет	Оператор
Най – висок	! > >= < <= == != &&
Най – нисък	

Като пример следващия израз *if* изчислява за *true* и отпечатва реда *x is less than 10*:

```
X = 9;
if (x < 10) cout << "x is less than 10";
```

Обаче в следващия пример, никакво съобщение не се показва понеже двете операнди свързани със && трябва да са *true* за да е резултата *true*:

```
X = 9;
Y = 9;
if (x < 10 && y > 10)
    cout << "this will not print. ";
```

Побитови оператори

C и C++ доставят оператори които действат върху действителните битове които съдържа стойността. Побитовите оператори могат да бъдат използвани само върху целочислени типове. Побитовите оператори са:

Оператор	Значение
&	AND
	OR
^	XOR
~	Единствено допълнение
>>	Преместване надясно
<<	Преместване наляво

Оператори &, | и ^

Таблицата за истинност на &, | и ^ е:

p	q	p&q	p q	p^q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Тези правила се прилагат към всеки бит във всеки операнд когато побитовите **AND**, **OR** или **XOR** се изпълнят. Например пример за побитова **AND** операция е показана тук:

$$\begin{array}{r} 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \\ \& \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \\ \hline 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \end{array}$$

Побитова **OR** операция изглежда така:

$$\begin{array}{r} 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \\ | \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \\ \hline 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \end{array}$$

Побитова **XOR** операция е показана тук:

$$\begin{array}{r} 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \\ ^ \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \\ \hline 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \end{array}$$

Оператор за единствено допълнение ~

Оператора ~ ще инвертира всички битове в неговия операнд. Например ако символната променлива **ch** има следната схема:

$$0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1$$

тогава

$$ch = \sim ch;$$

поставя битовия шаблон

$$1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0$$

в **ch**.

Преместващите оператори >> и <<

Десния (>>) и левия (<<) преместващи оператори преместват всички битове във целочислена стойност със зададени позиции. Когато битовете се преместват с 1 назад, се прибавя 0 в другия край (ако стойността която се премества е отрицателна и се изпълни дясно преместване, тогава се поставят 1 отпред за да се запази знзка). Числото от дясната страна на преместващия оператор определя броя на преместващите позиции. Общата форма на всеки преместващ оператор е:

стойност >> число

стойност << число

Тук **число** определя броя на позициите на които се премества стойността. Подавайки битовия шаблон (и приемайки неозначена стойност):

0 0 1 1 1 1 0 1
при преместване надясно се получава :

0 0 0 1 1 1 1 0

а при преместване наляво :

0 1 1 1 1 0 1 0

Програмен съвет

Преместването надясно е в действителност деление на 2, а наляво е умножение по 2. За много компютри, преместването е по – бързо от умножението или делението. Ако ви трябва бърз начин да умножите или разделите на 2, разгледайте употребата на преместващите оператори. Например следващия фрагмент от код първо ще умножи и след това ще раздели стойността в *x* на 2 :

```
int x;
x = x << 1;
x = x >> 1;
```

Разбира се, когато използвате преместващите оператори за да изпълните умножение, трябва да внимавате да не преместите битовете извън края. Приоритета на побитовите оператори е:

Приоритет	Оператор
Най-висок	~
	>> <<
	&
	^
Най-нисък	

Указателни оператори

Двата указателни оператора са * и &. Указател е променлива която съдържа адреса на друг обект. Или представено различно, променлива която съдържа адреса на друг обект се казва че сочи към този обект.

Указателен оператор &

Оператора & връща адреса на обекта който предхожда. Например, ако цялото число *x* е разположено в паметта на адрес 1000 тогава

```
p = &x;
```

Поставя стойността 1000 в *p*. & може да бъде означен като “адреса на “
Например предишния израз може да бъде прочетен като “постави адреса на *x* във *p* “

Указателен оператор *

& е индиректен оператор. Той използва текущата стойност на променливата която предхожда като адрес на който данните ще бъдат съхранени или получени. Например фрагмента:

$p = \&x;$ /* слага адреса на x във p */

$*p = 100;$ /* използва адреса съдържан от $*p$ */

поставя стойността 100 във x . * може да бъде запомнен като “на адрес”. В този пример втория ред може да бъде прочетен като “постави стойност 100 на адрес p ”. Понеже p съдържа адреса на x , стойността 100 е в действителност съхранена в x . Казано в думи p се казва че сочи към x . Оператора * може също да бъде използван в дясната страна на присвояване. Например:

$p = \&x;$

$*p = 100;$

$z = *p/100;$

поставя стойността 10 във z .

Оператори за присвояване

В C/C++ присвояващия оператор е единичен знак за равенство. Когато присвоявате обща стойност на няколко променливи, вие можете да ги “вържете заедно”. Например:

$a = b = c = 10;$

Присвоява на a , b и c стойността 10. C/C++ позволява удобна форма на “късо” присвояване в тази обща форма:

променлива = променлива операция израз;

Присвоявания от този тип могат да бъдат скъсени към:

променлива операция = израз;

Например двете присвоявания:

$x = x + 10;$

$y = y / z;$

Могат да бъдат пренаписани така:

$x += 10;$

$y /= z;$

Оператора ?

Оператора **?** е тернарен оператор (работи със три операнда). Има следната обща форма:

израз1 ? израз2 : израз3;

Ако **израз1** е **true**, тогава резултата от операцията е **израз2**, в противен случай е стойността на **израз3**.

Програмен съвет

? се използва често да замества **if-else** условието от този общ вид:

```
if ( израз1 ) променлива = израз2 ;  
else променлива = израз2 ;
```

Например поредицата

```
if (  $y < 10$  )  $x = 20$  ;  
else  $x = 40$  ;
```

Може да бъде написана така :

```
 $x = (y < 10) ? 20 : 40$  ;
```

Тук x присвоява стойност 20 ако y е по-малко от 10 и 40 ако това не е така. Една причина че оператора $?$ съществува освен спестяването на писане е че компилатора може да произведе много бърз код за това условие - много по-бързо отколкото за подобни **if-else** изрази.

Оператори за членове

Оператора $.$ и оператора $\rightarrow$ (стрелка) се използват за обръщане към отделните членове на класове, структури и обединения. Оператора точка е приложен към действителния обект. Оператора стрелка се използва със указател към обект. Например подавайки следната структура :

```
struct date_time {  
    char date[16] ;  
    int time ;  
} tm ;
```

за присвояване на стойността "3/12/99" на **date** член от обект **tm** вие трябва да напишете :

```
strcpy ( tm.date , "3/12/99" ) ;
```

Обаче ако **p_tm** е указател към обект от тип **date_time** се използва следния израз :

```
strcpy ( p_tm  $\rightarrow$  date , "3/12/99" ) ;
```

Оператора запетайка

Оператора запетайка предизвиква да бъде изпълнена поредица от операции. Стойността на целия израз със запетайки е стойността на последния израз от списъка. Например след изпълнението на следния фрагмент :

```
 $y = 15$  ;  
 $x = (y = y - 5, 50 / y)$  ;
```

x ще има стойност 5 понеже първоначалната стойност на y от 15 се намалява със 5 и тогава се разделя на 50 получавайки като резултат 5.

Можете да смятате оператора запетайка като "направи това и това". Оператора се използва често във изразите **for**. Например :

```
for (  $z = 10, b = 20 ; z < b ; z++ , b--$  ) { // ...
```

Тук z и b се инициализират и изменят чрез използване на разделени със запетайки изрази.

sizeof

Въпреки че *sizeof* е също ключова дума, той е оператор по време на компилация, използван за определяне размера в байтове на променлива или типове данни включително класове, структури и обединения. Ако се използва със тип, името на типа трябва да бъде затворено със кръгли скоби. За повечето 32 – битови компилатори следващия пример отпечатва 4:

```
int x;
cout << sizeof x;
```

Преобразуване

Преобразуване е специален оператор който конвертира един тип данни във друг. И C и C++ поддържат тази форма на преобразуване:

(тип) израз;

където **тип** искания тип данни. Например следващото преобразуване предизвиква резултата от целочисленото делене да бъде от тип *double*:

```
double d;
d = (double) 10 / 3;
```

Преобразувания в C++

C++ поддържа допълнителни оператори. Това са *const_cast*, *dynamic_cast*, *reinterpret_cast* и *static_cast*. Техните общи форми са:

```
const_cast <тип> (обект)
dynamic_cast <тип> (обект)
reinterpret_cast <тип> (обект)
static_cast <тип> (обект)
```

Тук **тип** определя целевия тип на преобразуването и **обект** е обекта който ще бъде преобразуван в новия тип. *const_cast* оператора се използва изрично да предефинира *const* и/или *volatile* за преобразуване. Целевия тип трябва да бъде еднакъв като изходния тип с изключение на техните *const* и *volatile* атрибути. Най – честата употреба на *const_cast* е да премахва константността. Оператора *dynamic_cast* изпълнява преобразуване по време на изпълнение (**runtime**) който проверява валидността на преобразуването. Ако преобразуването не може да стане, то пропада и израза се изчислява на 0. Например имайки два полиморфни класа **B** и **D** и **D** е произведен на **B**, *dynamic_cast* може винаги да преобразува **D*** указател във **B*** указател. *dynamic_cast* може да преобразува **B*** указател във **D*** указател само ако обекта който указва е в действителност **D***. Въобще *dynamic_cast* ще е успешна ако опитаното полиморфно преобразуване е позволено (тоест ако целевия тип може легално да се добави към типа от обекта който ще се преобразува) Ако преобразуването не може да стане тогава *dynamic_cast* се изчислява на 0. Оператора *static_cast* изпълнява непалиморфно преобразуване. Например той да преобразува указател към базов клас във указател към произведен клас. Той може също да бъде използван за всякакво стандартно превръщане. Никакви **runtime** проверки не се изпълняват. Оператора

reinterpret_cast променя един тип във фундаментално различен тип . Например той може да бъде използван да промени указател във цяло число .

reinterpret_cast трябва да бъде за преобразуване на изцяло несъвместими типове указатели . Само ***const_cast*** може да преобразува константността . Нито ***dynamic_cast*** , ***static_cast*** нито ***reinterpret_cast*** могат да променят константността на обект .

Оператори за I / O

В C++ , <<i>i</i> >> са предефинирани да изпълняват I/O операции . Когато се използват в израз в който левия оператор е поток , >> е входен оператор , а << е изходен оператор . В езика C++ >> е наречен екстрактор понеже извлича данни от входния поток . << е наречен вмъквач понеже той вмъква данни във изходния поток . Общата форма на тези оператори е :

входен поток >> променлива

изходен поток << израз

Например , следващия фрагмент въвежда две целочислени променливи:

```
int i, j;
cin >> i >> j;
```

Следващия израз показва “This is a test 10 20 “

```
cout << “This is a test “ << 10 << ‘ ‘ << 4 * 5;
```

I / O операторите не се поддържат от C .

. * и ->* указатели към член функции

C++ позволява да направите специален тип указател който сочи всъщност към член на клас , а не към член на обект . Този вид указател е наречен указател към член . Указателя към член не е еднакъв като нормалния C++ указател . Вместо това указателя към член доставя само отместването в обекта на члена на класа на което този член може да бъде намерен . Понеже член указателите не са истински указатели . и -> оператори не могат да бъдат приложени към тях . За достъп на член от клас със указател към него трябва да използвате специалните указатели към тях . * и ->* . Когато достигате член от обект подаващ обект или псевдоним към обект използвайте .* оператор . Когато достъпвате член подаващ указател към обект използвайте ->* оператор . Указател към член се декларира по тази обща форма :

тип име-на-клас :: * указател ;

Тук **тип** е базовия тип на члена , **име-на-клас** е името на класа , и **указател** е името на указателя към член променливата която се създава . Веднъж създаден , **указател** може да сочи към всеки член на неговия клас който е от тип **тип** . Това е къс пример който демонстрира .* оператор . Обърнете специално внимание на начина по който са декларирани членовете указатели :

```
#include <iostream>
using namespace std;
class cl {
public;
    cl (int i) { val = i; }
```

```

    int val;
    int double_val () { return val + val ; }
};
int main ()
{
    int cl :: *data ; // указател към int член променливи
    int (cl :: *func) () ; // указател към член функции
    cl ob1 (1) , ob (2) ; // създава обекти
    data = &cl :: val ; // получава отместването
    func = &cl :: double_val ; // получава отместването
    cout << "Here are values : " ;
    cout << ob1. *data << " " << ob2. *data << "\n" ;
    cout << "Here they are doubled : " ;
    cout << (ob1. *func) () << " " ;
    cout << (ob2. *func) () << "\n" ;
    return 0 ;
}

```

Указателите към член не се поддържат от C .

Оператор за определяне област на видимост ::

Оператора за разрешаване на интервал :: определя интервала към който принадлежат членовете . Той има следната обща форма :

име :: име_на_член

Тук **име** е името на класа или именованото пространство което съдържа члена определен от **име_на_член** . Казано различно , **име** определя интервала в който може да бъде открит идентификатора определен от **име_на_член** . За обръщане към глобален интервал не трябва да определяте име на интервал . Например за обръщане към глобална променлива наречена **count** която ще бъде скрита от локална променлива count трябва да използвате този израз :

:: count

Оператора за разрешаване на интервал не се поддържа от C .

new и delete

Операторите **new** и **delete** са оператори за динамично заделяне в C++ . Те са също ключови думи . Виж глава 5 за детайли . C не поддържа операторите **new** и **delete** .

typeid

В C++ оператора **typeid** връща псевдоним към **type_info** обект който описва типа на обекта към който е бил приложен **typeid** . Общата форма на **typeid** е :

typeid (обект)

Оператора **typeid** поддържа идентификация по време на изпълнение (RTTI) в C++ . Класа **type_info** дефинира следните **public** членове :

```

bool operator == (const type_info &ob) const;
bool operator != (const type_info &ob) const;
bool before (const type_info &ob) const;
const char * name () const;

```

Предефинираните `==` и `!=` са добавени за сравнение на типове. Функцията `before()` връща `true` ако извиквания обект е преди обекта използван като параметър в сравнителен ред. (Тази функция е главно за вътрешна употреба. Нейната връщана стойност не прави нищо със наследяването или класовата йерархия). Функцията `name()` връща указател към името на типа. Когато `typeid` е приложен към указател към базов клас от полиморфичен клас, той автоматично ще върне типа на обекта който указва (полиморфичен клас е такъв който съдържа най-малко една виртуална функция) Така `typeid` може да бъде използван за определяне типа на обекта указван от указателя към базов клас. С не поддържа `typeid`.

Предефиниране на оператори

В C++ операторите могат да бъдат предефинирани чрез използване на ключовата дума `operator` (виж глава 5) Предефинирането на оператори не се поддържа от С.

Таблица на приоритетите на операторите

Следващата таблица изброява приоритетите на всички С и C++ оператори. Отбележете че всички оператори с изключение на унарните, присвояващите и `?`, се свързват отляво надясно:

Приоритет
Най – висок

оператори

`() [] -> :: .`

`! ~ ++ -- * & sizeof new delete typeid`
преобразуване на тип

`.* ->*`

`* / %`

`+ -`

`<< >>`

`< <= > >=`

`== !=`

`&`

`^`

`|`

`&&`

`||`

`?:`

`= += -= *= /= %= >>= <<= &= ^= |=`

,

Най – нисък

ГЛАВА 4- Предпроцесор и коментари

C и C++ включват няколко предпроцесорни директиви които се използват за задаване на инструкции към компилатора . Предпроцесорните директиви са изброени тук :

<i>#define</i>	<i>#error</i>	<i>#include</i>
<i>#elif</i>	<i>#if</i>	<i>#line</i>
<i>#else</i>	<i>#ifdef</i>	<i>#pragma</i>
<i>#endif</i>	<i>#ifndef</i>	<i>#undef</i>

Всяка е разгледана накратко в тази секция

#define

Директивата *#define* се използва да изпълни заместване със макрос на една част от текста със друга навсякъде във файла в който е използвана . Общата форма на директивата е :

#define име_на_макрос символна- последователност

Тук , всеки път когато **име_на_макрос** се срещне се замества със определената символна последователност . Забележете че няма точка и запетайка в този израз . Освен това когато започне символната поредица тя се прекъсва само от края на реда . Например ако искате да използвате стойност 1 за думата **“TRUE ”** и стойност 0 за **“FALSE ”** , трябва да декларирате тези два изрази *#define* :

```
#define TRUE 1
#define FALSE 0
```

това ще предизвика компилатора да замени със 1 или 0 всеки път когато думите **TRUE** или **FALSE** се срещнат . Директивата *#define* има и друго свойство : макроса може да има аргументи . Макрос който приема аргументи действа почти като функция . Всъщност , този тип макрос е често определян като макрос подобен на функция . Всеки път когато макроса се срещне , аргументите свързани със него се заместват със действителните аргументи във програмата . Това е пример :

```
#include <iostream>
using namespace std;
#define ABS(a) ((a)<0 ? -(a) : (a))
int main()
{
    cout << “abs of -1 and 1 : “ << ABS(-1) << ‘ ‘
    << ABS(1);
    return 0;
}
```

Когато тази програма се компилира а във макросната дефиниция ще се замени със стойностите -1 и 1 .

Програмен съвет

Трябва да поставите във скоби макросите подобни на функция когато ги създавате . Ако не го направите , те може да не работят във всички ситуации . Например ограждането на *a* във предишния пример е необходимо за осигуряване на правилното заместване във всички случай . Ако ограждането бъде махнато изрази :

ABS (10 – 20)

ще бъде преобразуван във :

10 – 20 ? – 10- 20 : 10 – 20

и се получава грешен резултат . Ако имате макрос подобен на функция който не се държи правилно проверете огражданията.

#error

Директивата **#error** предизвиква компилатора да спре компилацията когато я срещне . Тя се използва главно при дебъгване . Общата ѝ форма е :

#error съобщение

Когато се срещне **#error** се показват съобщение и номера на реда .

#if , #ifdef , #ifndef , #else , #elif , и #endif

Предпроцесорните директиви **#if** , **#ifdef** , **#ifndef** , **#else** , **#elif** и **#endif** се използват за селективна компилация на различни части от програмата . Общата идея е че ако изрази след **#if** , **#ifdef** , или **#ifndef** е true кода който е между един от предходните и **#endif** ще се компилира , иначе ще бъде прескочен . **#endif** се използва да маркира края на блока **#if** . **#else** може да се използва със всяко от тях за да добави алтернатива . Общата форма на **#if** е :

#if израз- от –константи

Ако **израз- от – константи** е *true* кода който следва непосредствено ще бъде компилиран . Общата форма на **#ifdef** е:

#ifdef име_на_макрос

Ако **име_на_макрос** е било дефинирано в израз **#define** кода след изрази ще се компилира . Общата форма на **#ifndef** е :

#ifndef име_на_макрос

Ако **име_на_макрос** е недефинирано в израз **#define** кода след изрази ще се компилира . Например , тук е показано как тези предпроцесорни директиви работят заедно . Следващия код ще отпечати “Hi Ted “ и “Hi Jon “ но не и ”Hi George “ :

```
#define ted 10
//...
#ifdef ted
    cout << “Hi Ted \n ”;
#endif
    cout << “Hi Jon “ \n “;
    #if 10 < 9
        cout << “ Hi George \n “;
    #endif
```

Директивата **#elif** се използва да създаде израз **if – else – if**. Нейната обща форма е :

#elif израз – от – константи

Можете да вържете заедно серия от **#elif** за манипулиране на различни алтернативи. Можете също да използвате **#if** или **#elif** за определяне дали е било дефинирано име на макрос чрез използване на предпроцесорния оператор **defined**.

Той има тази обща форма :

```
#if defined име_на_макрос
поредица от изрази
#endif
```

Ако **име_на_макрос** е било дефинирано, поредицата от изрази ще се компилира. В противен случай ще бъде прескочена. Например следващия фрагмент компилира условния код понеже **DEBUG** е дефиниран :

```
#define DEBUG
// ...
int i = 100 ;
//...
#if defined DEBUG
    cout << "value of I is : " << i << endl ;
#endif
```

Можете също да поставите пред **defined** оператора ! за да предизвика условна компилация когато макроса не е дефиниран.

#include

Директивата **#include** инструктира компилатора да прочете и компилира друг сорс файл. Тя има тези общи форми :

```
#include "име_на_файл "
#include <име_на_файл >
```

Сорс файла който ще бъде четен трябва да бъде затворен между двойни кавички или ъглови скоби. Например :

```
#include <stdio.h >
```

ще инструктира компилатора да прочете и компилира хедъра за функциите на C за I/O. Ако **име_на_файл** е затворено в ъглови скоби, файла се търси по начин определен от създателя на компилатора. Често това означава претърсване на някакво специално множество директории за хедърни файлове. Ако файловото име е заградено във кавички файла се търси по друг определен от изпълнението начин. За повечето случаи това означава претърсване на текущата работна директория. Ако файла не е открит търсенето се повтаря както ако името на файла е поставено във ъглови скоби. Вие трябва да проверите ръководството на компилатора за повече подробности за разликите между тях. **#include** изразите могат да бъдат вложени във други включвани файлове. Докато всички версии на C++ приемат тези версии на **#include**, стандартния C++ дефинира хедъри в нов стил които се използват да включат информация за стандартната C++ библиотека. Хедърите в нов стил не са имена на файлове (въпреки че могат

да се съпоставят на имена на файлове) За включване на хедър в нов стил използвайте тази обща форма :

```
#include <име_на_хедър>
```

Тук **име_на_хедър** ще бъде един от хедърите в нов стил описани във Глава 2 . Например за включване на информация за хедър за I/O системата използвайте :

```
#include <iostream>
```

Понеже хедърите в нов стил не са имена на файлове , те не трябва да имат разширение **.h** .Вижте вашата документация на компилатора за повече информация за включването на стандартните C++ хедъри във вашите C++ програми .

#line

Директивата **#line** се използва да промени съдържанието на **__LINE__** и **__FILE__** ,които са предефинирани идентификатори .Общата форма на командата е :

```
#line номер "име_на_файл"
```

където **номер** е някакво положително цяло число и **име_на_файл** е някакъв валиден файлов идентификатор . Стойността на номер става номера на текущия ред от кода и **име_на_файл** става името на сорс файла . Името на файла не е задължително . **#line** е обикновено използвана за дебъгване и специални приложения . Идентификатора **__LINE__** е цяло число и **__FILE__** е нулево – термилиран низ . Например следващия код установява текущия брояч на редове на 10 и файловото име на "test" :

```
#line 10 "test"
```

#pragma

Директивата **#pragma** е зависеща от изпълнението директива която позволява различни инструкции да бъдат подадени на компилатора . Например компилатора може да има опция да поддържа трасиране на изпълнението на програмата . Трасиращата опция тогава ще трябва да бъде определена чрез израз **#pragma** . Проверете вашата документация на компилатора за детайли и опции .

#undef

Директивата **#undef** премахва дефинирано преди това име на макрос . Общата форма е :

```
#undef име_на_макрос
```

Например във следващия код :

```
#define LEN 100
#define WIDTH 100
char array [LEN][WIDTH];
#undef LEN
#undef WIDTH
```

/* от тази точка и двете **LEN** и **WIDTH** са недефинирани

*/

и двете **LEN** и **WIDTH** са дефинирани докато не се срещнат изразите **#undef**

Предпроцесорните оператори # и

C/C++ доставя два предпроцесорни оператора: # и ##. Тези оператори се използват за употреба във **#define** макросите. Оператора # предизвиква аргумента който предхожда да бъде преобразуван във низ със кавички. Например разгледайте следната програма:

```
#include <iostream>
using namespace std;
#define mkstr(s)  #s
int main()
{
    cout << mkstr(I like C++);
    return 0;
}
```

Предпроцесора преобразува реда

```
cout << mkstr(I like C++);
```

във

```
cout << "I like C++";
```

Оператора ## се използва да свърже два символа. Например във следващата програма:

```
#include <iostream>
using namespace std;
#define concat(a, b)  a ## b
int main()
{
    int xy = 10;
    cout << concat(x, y);
    return 0;
}
```

Предпроцесора трансформира

```
cout << concat(x, y);
```

във

```
cout << xy;
```

Ако тези оператори изглеждат странни за вас, запомнете че те не са нужни или използвани във повечето програми. Те съществуват главно да позволят някой специални случай да бъдат манипулирани от предпроцесора.

Предефинирани имена на макроси

C/C++ определя пет вградени предефинирани имена на макроси. Те са:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
```

Макросите `__FILE__` и `__LINE__` са описани в разглеждането на *#line* по-рано във тази глава. Другите са представени тук. Макроса `__DATE__` е низ във формата **месец / ден / година**, който е датата на преобразуването на сорс файла във обектен код. Часът на превеждането на сорс файла във обектен код се съдържа като низ във `__TIME__`. Формата на низа е **час : минута : секунда**. Макроса `__cplusplus` е дефиниран когато компилирате C++ програма. Този макрос няма да бъде дефиниран от C компилатор. Макроса `__STDC__` е дефиниран когато компилирате C програма и може да бъде дефиниран от C++ компилатор. И в двата случая проверете документацията на компилатора за подробности. Повечето C/C++ компилатори дефинират различни други вградени макроси които се отнасят за специфичната среда и изпълнение.

Коментари

C++ дефинира два стила за коментари. Първия е многоредов коментар. Той започва със `/*` и завършва със `*/`. Всичко между коментарните символи се отхвърля от компилатора. Многоредовия коментар може да бъде разширен на няколко реда. Втория тип коментар е едноредовия коментар. Той започва със `//` и завършва в края на реда. Многоредовия коментар е единствения тип коментар поддържан от C. Обаче повечето C компилатори приемат едноредовите коментари макар и че не са стандартни.

ГЛАВА 5 – Справочник на ключовите думи

Езика C дефинира следните 32 ключови думи:

<i>auto</i>	<i>double</i>	<i>int</i>	<i>struct</i>
<i>break</i>	<i>else</i>	<i>long</i>	<i>switch</i>
<i>case</i>	<i>enum</i>	<i>register</i>	<i>typedef</i>
<i>char</i>	<i>extern</i>	<i>return</i>	<i>union</i>
<i>const</i>	<i>float</i>	<i>short</i>	<i>unsigned</i>
<i>continue</i>	<i>for</i>	<i>signed</i>	<i>void</i>
<i>default</i>	<i>goto</i>	<i>sizeof</i>	<i>volatile</i>
<i>do</i>	<i>if</i>	<i>static</i>	<i>while</i>

C++ включва всички ключови думи дефинирани от C и добавя следните:

<i>asm</i>	<i>inline</i>	<i>template</i>
<i>bool</i>	<i>mutable</i>	<i>this</i>
<i>catch</i>	<i>namespace</i>	<i>throw</i>
<i>class</i>	<i>new</i>	<i>true</i>
<i>const_cast</i>	<i>operator</i>	<i>try</i>
<i>delete</i>	<i>private</i>	<i>typeid</i>
<i>dynamic_cast</i>	<i>protected</i>	<i>typename</i>
<i>explicit</i>	<i>public</i>	<i>using</i>
<i>false</i>	<i>reinterpret_cast</i>	<i>virtual</i>
<i>friend</i>	<i>static_cast</i>	<i>wchar_t</i>

По – старите версии на C++ също дефинираха ключовата дума ***overload*** но тя е излязла от употреба . Всички ключови думи са със малки букви . Следва кратко описание на всяка от ключовите думи :

asm

Asm се използва да вгради асемблерен език директно във вашата C++ програма . Общата форма на asm израза е :

asm (“ инструкция ”) ;

Тук **инструкция** е инструкция на асемблерен език , която се подава директно на компилатора за вграждане във програмата . Много C++ компилатори позволяват допълнителни форми на израза ***asm*** . Например **Borland C++** позволява следните ***asm*** изрази :

```
asm инструкция ;
asm {
    последователност от инструкции
}
```

Тук **последователност от инструкции** е списък със асемблерни инструкции .

Забележка : **Microsoft Visual C++** използва ***__asm*** за вграждане на асемблерен код . Той иначе е подобен на ***asm*** .

auto

auto декларира локална променлива . Тя е напълно незадължителна и рядко използвана .

bool

Спецификатора за тип ***bool*** се използва за деклариране на булеви (тоест ***true*** (***false***)) стойности .

break

break се използва за изход от ***do*** , ***for*** или ***while*** цикъл прескачайки нормалното условие на цикъла . Тя също се използва за изход от израза ***switch*** . Пример за ***break*** във цикъл е показан тук :

```
do {
    x = getx ( ) ;
    if ( x < 0 ) break ; //прекъсва if ако е отрицателно
    process (x) ;
} while (!done) ;
```

Тук ако ***x*** е отрицателно , цикъла се прекъсва . Във израз ***switch*** , ***break*** предпазва програмата от влизане във следващия ***case*** (Вижте “***switch*** “ за подробности) . Break прекъсва само ***for*** , ***do*** , ***while*** или ***switch*** които го съдържат . Той няма да прекъсне никой вложен цикъл или ***switch*** израз .

case

Израза ***case*** се използва със израза ***switch*** . Виж “ ***switch*** “ .

catch

Изрази *catch* манипулира изключение генерирано от *throw*. Виж “*throw*”.

char

char е тип данни използван за деклариране на символни променливи.

class

class се използва за деклариране на класове – основната част в C++ за капсулиране. Неговата обща форма е:

```
class име_на_клас : списък с наследявания {
    // private членове по подразбиране
protected :
    // private членове които могат да се наследяват
public :
    // public членове
} списък със обекти ;
```

Тук *име_на_клас* е името на новия тип данни който се генерира от декларацията *class*. **Списъка с наследявания** който е незадължителен, определя всички базови класове наследени от новия клас. По подразбиране, членовете на *class* са *private*. Те могат да станат *protected* или *public* чрез използване на ключовите думи *protected* и *public*, съответно. **Списък с обекти** е опционен. Ако не е представен, декларацията просто определя форма на клас. Тя не създава никакви обекти от класа.

Забележка : За допълнителна информация вижте Глава 1

const

Модификатора *const* казва на компилатора че променливата не може да бъде променяна от програмата. *const* променлива може обаче да взема инициализираща стойност когато се декларира.

const_cast

Оператора *const_cast* се използва да предефинира явно *const* и/или *volatile* при преобразуване. Той има тази форма:

```
const_cast <тип> (обект)
```

Целевия тип трябва да бъде еднакъв като изходния тип с изключение на променянето на неговите *const* или *volatile* атрибути. Най – честата употреба на *const_cast* е да премахва константността (*const*).

continue

continue се използва да прескочи часта от кода във цикъл и да предизвика условия израз да бъде изчислен. Например следващия цикъл **while** просто ще чете символи от клавиатурата докато не се въведе **s** :

```
while ( ch = getchar ( ) ) {  
    if ( ch != 's' ) continue ; // чете друг знак  
    process ( ch ) ;  
}
```

Извикването на **process** () няма да стане докато **ch** не съдържа символа **s** .

default

default се използва във израза **switch** да означава подразбиращ се блок от код който да бъде изпълнен ако няма съвпадения открити във **switch** (Виж “ **switch** ”)

delete

Оператора **delete** освобождава паметта указвана от неговия аргумент . Тази памет трябва да бъде заделена преди това от **new** . Формата на **delete** е :

```
delete p_var ;
```

където **p_var** е указател към преди това заделена памет . За освобождаване на масив който е бил заделен чрез **new** използвайте тази форма на **delete** :

```
delete [] p_var ;
```

do

Цикъла **do** е един от трите конструкции за цикъл налични във C++ . Общата форма на цикъла е :

```
do {  
    блок от изрази  
} while ( условие ) ;
```

Ако само един израз се повтаря скобите не са необходими , но те добавят яснота на израза . Цикъла **do** е единствения цикъл в C++ който винаги ще има поне една итерация понеже условието се проверява на края на цикъла .

double

double е спецификатор за тип данни използван за деклариране на променлива със плаваща точка със двойна точност.

dynamic_cast

dynamic_cast изпълнява преобразуване по време на изпълнение (**runtime**) което проверява валидността на преобразуването. Има тази обща форма:

***dynamic_cast* <тип> (обект)**

Главната употреба на *dynamic_cast* е да изпълни преобразуване на полиморфни типове. Например, имайки два полиморфни класа ***B*** и ***D***, и ***D*** е произведен от ***B***, *dynamic_cast* може винаги да преобразува ***D**** указател във ***B**** указател. *dynamic_cast* може да преобразува ***B**** указател във ***D**** указател само ако обекта който ще се указва е в действителност ***D****. Въобще *dynamic_cast* ще успее ако опитаното полиморфно преобразуване е позволено (тоест ако целевия тип може легално да бъде прибавен към типа от обекта който се преобразува). Ако преобразуването се осъществи тогава *dynamic_cast* се изчислява на 0.

else

Виж “*if*”.

enum

Спецификатора за тип ***enum*** се използва да създаде изброителни типове. Изброяването е просто списък от именовани целочислени константи. Общата форма на изброяване е:

***enum* име { списък-с – имена } списък – с – променливи ;**

Име е името на типа изброяване. Списък – с – променливи е опционен и изброителните променливи могат да бъдат деклариран отделно от дефиницията на типа, както показва следващия пример. Този код декларира изброяване наречено ***color*** и променлива от този тип наречена ***c*** и тя изпълнява присвояване и условна проверка:

enum color { red , green , yellow } c ;

c = red ;

if (c == red) cout << “ is red \n “ ;

Забележка : За повече подробности за изброяванията се обърнете към Глава 1.

explicit

Спецификатора ***explicit*** се прилага само за конструкторите. Конструктор определен като ***explicit*** ще може да бъде използван само когато инициализацията точно съответства на определената чрез конструктора. Никакво автоматично преобразуване не се извършва (създава се “ непреобразуващ конструктор “)

extern

extern е модификатор за типове данни който казва на компилатора за променлива която е дефинирана някъде на друго място в програмата . Това е често използвано при свързването със отделно компилирани файлове които поделят еднакви глобални данни и се свързват заедно . Всъщност тя информира компилатора за типа на променливата без да я предефинира . Например ако **first** беше дефинирана във друг файл като цяло число , тази декларация ще бъде използвана във други файлове :

```
extern int first ;
```

Тази декларация определя типа на **first** , но не заделя място за съхранението ѝ . В C++ **extern** се използва за създаване на свързваща спецификация . Тя има тази форма :

```
extern “ език “ прототип-на-функция
```

Тук **език** определя езика в който искате да се свърже функцията . С и C++ свързването са гарантирано поддържани . Вашия компилатор може също да поддържа и други свързвания . За да декларирате няколко функции използвайки еднаква свързваща спецификация използвайте тази обща форма :

```
extern “ език “ {  
    прототипи- на – функции  
}
```

false

false е булева константа за **false** .

float

float е спецификатор за тип данни използван за даклариране на променливи със плаваща точка .

for

Цикъла **for** автоматично инициализира и инкрементира променлива- брояч . Формата му е :

```
for ( инициализация ; условие ; инкрементиране ) {  
    блок със – изрази  
}
```

Ако **блок със – изрази** е само един израз скобите не са необходими . Макар че **for** позволява няколко варианта , най- общо инициализация установява променлива за контрол на цикъла в начална стойност . **Условие** е обикновено релационен израз който проверява контролната променлива със прекъсваща стойност , и инкрементиране я увеличава (или намалява) . Ако **условие** е **false**

във началото , тялото на цикъла **for** няма да се изпълни нито веднъж . Следващия израз ще отпечати съобщението “**hello**” 10 пъти :

```
for (t = 0 ; t < 10 ; t++) cout << “hello \n “ ;
friend
```

Ключовата дума **friend** разрешава не – член функция да има достъп до **private** членовете на клас . За да направите **friend** функция , включете нейния прототип във **public** секцията на декларацията на класа и сложете пред нейния прототип думата **friend** . Например в следващия клас **myfunc ()** е **friend** и не е член на **myclass** :

```
class myclass {
//...
public :
    friend void myfunc ( int a , float b ) ;
//...
};
```

Запомнете че **friend** функцията няма **this** указател понеже не е член на класа .

goto

Ключовата дума **goto** предизвиква изпълнението на програмата да прескочи до **етикет** определен във израза **goto** . Формата на **goto** е :

```
goto етикет ;
.
.
.
етикет :
```

Всички етикети трябва да завършват със двоеточие и не трябва да влизат във конфликт със ключови думи или имена на функции . Освен това **goto** може само да прескача във текущата функция , а не от една функция към друга.

Програмен съвет

Макар че **goto** беше изхвърлен от употреба преди десетилетия като метод за програмен контрол , понякога има употреба . Едно от приложенията му е като начин за изход от дълбоко вложени цикли . Например разгледайте този фрагмент :

```
int i , j , k ;
int stop = 0 ;
for ( i = 0 ; i < 100 && !stop ; i++ ) {
    for ( j = 0 ; j < 10 && !stop ; j++ ) {
        for ( k = 0 ; k < 20 ; k++ ) {
            // . . .
            if ( something ( ) ) {
                stop = 1 ;
```

```

        break ;
    }
}

```

Както можете да видите променливата **stop** се използва за отказ от два външни цикъла ако се случи някакво програмно събитие . Обаче по – добър начин да изпълните това е показан тук чрез използване на **goto** :

```

int i, j, k ;
for ( I = 0 ; I < 100 ; I++ ) {
    for ( j = 0 ; j < 10 ; j++ ) {
        for ( k = 0 ; k < 20 ; k++ ) {
            // . . .
            if ( something ( ) ) {
                goto done ;
            }
        }
    }
}
done : // ...

```

Както можете да видите , употребата на **goto** елиминира свръхпретрупването което се получава чрез добавянето на повтарящи се проверки на **stop** във предишната версия . Докато употребата на **goto** като форма със общо предназначение за контрол на цикъл трябва да бъде избягвано , понякога може да бъде ангажирана със голям успех .

if

Ключовата дума **if** позволява посоката на действието да бъде определена от резултата от условие . Формата на израза **if** е :

```

if ( условие ) {
    блок – с – изрази 1
}
else {
    блок – с – изрази 2
}

```

Ако се използват единични изрази , скобите не са необходими . **else** е незадължително . Условието може да бъде всякакъв израз . Ако се изчислява на **true** (всяка стойност различна от 0) тогава **блок – с – изрази 1** ще се изпълни , в противен случай ще се изпълни **блок – с – изрази 2** . Следващия фрагмент проверява дали **x** е по – голямо от 10 :

```

if ( x > 10 )
    cout << "x is greater than 10 . " ;
else
    cout << "x is less than or equal to 10 . " ;

```

inline

Спецификатора **inline** казва на компилатора да разшири кода на функцията отколкото да я извиква. Спецификатора **inline** е заявка а не команда, понеже различни фактори могат да попречат кода да бъде разширен. Някой общи ограничения като например рекурсивни функции, функции съдържащи цикъл или **switch** изрази и функции съдържащи **static** данни. Спецификатора **inline** предхожда останалата част от декларацията на функцията. Следващия фрагмент казва на компилатора да генерира **inline** код за **myfunc()**:

```
inline void myfunc ( int I )
{
    //...
}
```

Когато дефиницията е включена във декларацията на клас, тогава кода автоматично се разглежда като **inline** ако е възможно.

int

int е спецификатор за тип използван за деклариране на целочислени променливи.

long

long е модификатор на типове данни използван за деклариране на дълги целочислени променливи.

mutable

Спецификатора **mutable** позволява на член на обект да предефинира константността (**const**). Така **mutable** член на **const** обект не е **const** и може да бъде изменен. Главната употреба на **mutable** е за позволяване на **const** член – функция да изменя избрани членове данни.

namespace

Ключовата дума **namespace** ви позволява да разделите глобалното именовано пространство чрез създаване на декларативен регион. Всъщност именованото пространство определя интервал. Общата форма на **namespace** е:

```
namespace име {
    // декларации
}
```

В добавка може да имате безименно именовано пространство както е показано тук:

```
namespace {
    // декларации
}
```

Безименното именовано пространство ви позволява да създадете уникални идентификатори които са познати само в обхвата на единствен файл. Тук е пример за *namespace* :

```
namespace MyNameSpace {
    int i, k;
    void myfunc (int j) { cout << j; }
}
```

Тук *i*, *k* и *myfunc()* са част от интервал определен от именованото пространство *MyNameSpace*. Понеже *namespace* дефинира интервал, трябва да използвате оператора за разрешаване на интервал за достъп до обекти дефинирани в него. Например за присвояване на стойност 10 на *i*, трябва да използвате този израз :

```
MyNameSpace :: i = 10 ;
```

Ако членовете на именовано пространство ще бъдат често използвани, може да използвате директива *using* за опростяване на достъпа до тях. Израза *using* има тези две форми :

```
using namespace име ;
using име :: член ;
```

В първата форма, *име* определя името на именованото пространство до което искате достъп. Всички членове във определеното именовано пространство могат да бъдат достъпвани директно. Във втората форма само определен член на именованото пространство се прави видим. Например приемайки *MyNameSpace* както е показано по-горе следващите *using* изрази и присвоявания са валидни :

```
using MyNameSpace :: k ; // само k се прави видим
k = 10 ; // ОК понеже k е видим
using namespace MyNameSpace ;
// всички членове на MyNameSpace са видими
i = 10 ;
// ОК понеже сега са видими всички членове на MyNameSpace
```

new

Оператора *new* заделя динамично памет и връща указател от съответния тип към нея. Общата му форма е :

```
p_var = new тип ;
```

Тук *p_var* е указателна променлива която получава адреса на заделената памет, а *тип* е типа данни които ще се съхранява паметта. Оператора *new* автоматично заделя достатъчно памет за съхранението на данни от определения тип. Например следващия фрагмент заделя достатъчно памет за съхранение на *double* :

```
double *p_var ;
p = new double ;
```

Ако заявката за заделяне пропадне, ще се случи едно от тези две събития : или ще се върне нулев указател или ще се предизвика изключение *bad_alloc* (понастоящем някои компилатори извеждат изключение *xalloc*).

Забележка : По време на написването на това точното държане на *new* при грешка е вече определено . Различните компилатори изпълняват различни действия при грешка в заделянето . Проверете документацията на компилатора за информация за сегашната ви работна среда . Можете да инициализирате заделената памет чрез определянето на инициализатор използвайки тази форма :

***p_var* = new тип (инициализатор) ;**

Тук **инициализатор** е стойността която ще бъде присвоена на заделената памет . За заделяне на едномеран масив използвайте тази форма :

***p_var* = new тип [размер] ;**

Тук **размер** определя дължината на масива . *new* автоматично ще задели достатъчно място за съхранение на масива от определения тип и големина . Когато заделите динамично масиви не можете да ги инициализирате .

operator

Ключовата дума **operator** се използва за създаване на предефинирана функция оператор . Операторните функции имат две разновидности : член и нечлен . Общата форма на операторна член функция е :

**връщан-тип име_на_клас :: operator# (списък-с –
параметри) {**
 //...
}

Тук **връщан – тип** е връщания от функцията тип , **име_на_клас** е името на класа за който се предефинира оператора и **#** е оператора който се предефинира . Когато предефинирате унарен оператор , **списък – с – параметри** е празен (оператора е подаден косвено в *this*) Когато предефинирате бинарен оператор **списък – с – параметри** определя операнда от дясната страна на оператора (оператора от лявата страна е подаден косвено в *this*) За нечлен функции , операторната функция има тази обща форма :

връщан – тип operator# (списък – с – параметри) {
 //...
}

Тук **списък-с – параметри** съдържа един параметър когато предефинираме унарен оператор и два параметъра когато предефинираме бинарен оператор . Когато предефинираме бинарен оператор операнда отляво е подаден в първия параметър , а операнда отдясно е подаден в десния параметър . Има няколко ограничения към предефинирането на оператори . Първо не можете да промените приоритета на оператора , не може да промените броя на операндите на оператора , не може да промените значението на оператор отнасящ се до вградените в C++ типове данни , не може да създадете нов оператор . Предпроцесорните оператори **#** и **##** не могат да се предефинират . Не могат да се предефинират и следните оператори : **. : .* ?**

private

Спецификатора за достъп *private* декларира *private* членове на клас. Той също се използва за частично наследяване на базов клас. Когато се използва за деклариране на *private* членове има тази форма:

```
class име_на_клас {
    // ...
    private :
        // private членове
};
```

Членовете на клас са *private* по подразбиране. Така спецификатора само ще се използва във декларацията на клас за започване на друг блок от *private* декларации. Например това е валидна декларация на клас:

```
class myclass {
    int a, b; // private по подразбиране
    public :
        int x, y; // тези са public
    private :
        int c, d; // тези са private
};
```

Когато се използва като спецификатор при наследяване *private* има тази форма:

```
class :: име_на_клас : private базов – клас { // ...
```

Чрез определяне на *базов – клас* като *private* всички *public* и *protected* членове на базовия клас стават *private* членове на производния клас. Всички *private* членове на базовия клас остават *private* за него.

protected

Спецификатора за достъп *protected* декларира членове на клас които са *private* за класа но могат да се наследяват от всеки производен клас. Има тази форма:

```
class име_на_клас {
    // ...
    protected : // прави ги protected
        // protected членове
};
```

Например :

```
class base {
    // ...
    protected :
        int a;
        // ...
};
// сега наследява base в derived клас
class derived : public base {
```

```
// ...
public ;
// ...
// derived има достъп до a
void f() { cout << a ; }
};
```

Тук *a* е *private* за *base* и не е достъпна никоя нечлен функция. Обаче, *derived* наследява достъп към *a*. Ако *a* беше просто дефинирана като *private*, *derived* нямаше да има достъп до нея. Когато се използва като спецификатор при наследяване *protected* има тази форма:

```
class :: име_на_клас : protected базов – клас { // ...
```

Чрез определяне на базовия клас като *protected*, всички *public* и *protected* членове на базовия клас стават *protected* членове на производния клас, а във всички случаи *private* членовете на базовия клас остават *private* за базовия.

public

Спецификатора за достъп *public* декларира *public* членове на клас. Той също се използва за публично наследяване на базов клас. Когато се използва за деклариране на *public* членове има тази форма:

```
class име_на_клас {
    // private членове по подразбиране
    public : // прави ги public
    // public членове
};
```

Членовете на клас са *private* по подразбиране. За да декларирате *public* членове трябва да ги определите като *public*. Когато се използва като спецификатор при наследяване, *public* има тази форма:

```
class :: име_на_клас : public базов – клас { // ...
```

Чрез определянето на базовия клас като *public*, всички *public* членове на базовия клас стават *public* членове на производния клас и всички *protected* членове на базовия клас стават *protected* членове на производния клас. Във всички случаи, *private* членовете на базовия клас остават *private* за базовия.

register

Модификатора за съхраняване на клас *register* заявява че достъпа към променлива ще бъде оптимизиран по скорост. Традиционно *register* се прилага само към *integer* и символни променливи, предизвиквайки да бъдат съхранявани във регистрите на CPU вместо да се поставят в паметта. Значението на *register* беше разширено да включва всички типове данни. Обаче други данни освен *integer* и символи не могат обикновено да се съхраняват в регистрите на CPU. За другите типове данни се използва или кеш паметта (или някакъв друг тип оптимизационна схема), или *register* заявката се отхвърля. *register* може да бъде използван само за локални променливи. В C не можете да вземете адреса на *register* променлива. Обаче

в C++ можете (макар че правейки това можете да попречите променливата да бъде оптимизирана).

reinterpret_cast

Оператора **reinterpret_cast** променя един тип във фундаментално различен тип. Например той може да се използва да промени указател във **integer**. Има тази обща форма:

reinterpret_cast <тип> (обект)

reinterpret_cast може да бъде използван за преобразуване на напълно несъвместими указателни типове.

return

Израза **return** предизвиква връщане от функция и може да бъде използван за прехвърляне на стойност към извикващия процес. Има тези две форми:

return ;
return стойност ;

В C++, формата която не определя стойност трябва да бъде използвана само във **void** функции. Следващата функция връща резултата от нейните два целочислени аргумента:

```
int mul (int a , int b)
{
    return a*b ;
}
```

Запомнете че когато се срещне **return**, функцията се връща прескачайки всеки друг код който може да има във нея. Също функция може да има повече от един **return** израз.

short

short е модификатор за тип данни използван за деклариране на къси цели числа.

signed

Главната употреба на модификатора за тип **signed** е за определяне на тип данни **signed char**. Неговата употреба във други типове е излишна понеже целите числа са със знак по подразбиране.

sizeof

Оператора по време на компилация **sizeof** връща дължината (в байтове) на променливата или типа който предхожда. Ако е поставен пред тип тогава типа трябва да бъде заграден в скоби. Ако е пред променлива ограждането е незадължително. Например имайки:

```
int i ;
cout << sizeof ( int ) ;
```

```
cout << sizeof i ;
```

и двата израза ще изведат 4 за повечето 32– битови C++ компилатори .

static

static е модификатор за тип данни който създава постоянно съхраняване за локална променлива която предхожда . Това позволява определената променлива да поддържа стойността си между извикванията на функция . **static** може също да бъде използван за деклариране на глобални променливи . В този случай тя ограничава интервала на променливата която модифицира само до файла в който тя е декларирана . В C++ когато **static** се използва върху член променливи на клас , тя предизвиква само едно копие от този член да бъде поделено от всички обекти от този клас .

static_cast

Оператора **static_cast** изпълнява полиморфно преобразуване . Например може да бъде използван за преобразуване на указател към базов клас във указател към производен клас . Той също може да бъде използван за всякакво стандартно превръщане . Никакви **runtime** проверки не се извършват . Има тази обща форма :

```
static_cast < тип > ( обект )
```

struct

Ключовата дума **struct** се използва за създаване на агрегатен тип данни наречен структура . В C++ структурата може да съдържа и функции и данни . В C++ структурата има еднакви свойства като **class** с изключение на това че по подразбиране нейните членове са **public** вместо **private** . Общата форма на C++ структура е:

```
struct име_на_клас : списък – с – наследявания {
    // public членове по подразбиране
protected :
    // private членове които могат да се наследяват
private :
    // private членове
} списък от обекти ;
```

Име_на_клас е името на типа структура която е тип клас . Отделните членове се достигат чрез използване на точка когато оперираме със структура или чрез използване на оператор стрелка когато оперирате чрез указател към структура . **Списък-с-наследявания** и **списък от обекти** са незадължителни . В C++ структурите могат да съдържат само член променливи , **private** и **protected** спецификаторите не са разрешени и няма списък с наследявания . Следващата структура в C-стил съдържа низ наречен **name** и две **integer** наречени **high** и **low** . Тя също декларира една променлива наречена **my_var** :

```
struct my_struct {
    char name [80] ;
    int high ;
```

```
int low ;
} my_var ;
```

Забележка : Виж Глава 1 за повече детайли за структурите .

switch

Израза **switch** е многократно разклоняващ израз в C/C++ . Той се използва да насочи изпълнението в един от няколко различни пътя . Общата форма на израза е :

```
switch ( израз ) {
    case константа 1 : последователност – от – изрази 1 ;
        break ;
    case константа 2 : последователност – от – изрази 2 ;
        break ;
    .
    .
    .
    case константа N : последователност – от – изрази N ;
        break ;
    default : изрази- по – подразбиране ;
}
```

Всяка **последователност от изрази** може да бъде от един или няколко израза . Частта **default** е опционна . И двете **израз** и **case** константи трябва да са целочислени типове . **switch** работи чрез проверяване на израз със константи . Ако е открито съвпадение се изпълнява тази последователност . Ако последователността от изрази свързани със съвпадащото **case** не съдържа **break** , изпълнението ще продължи във следващия **case** . Казано различно , от точката на съвпадение , изпълнението ще продължи докато или не се срещне израз **break** или **switch** не завърши . Ако няма съвпадение и случая **default** съществува , се изпълнява неговата последователност от изрази . В противен случай не се изпълнява нищо . Следващия пример обработва избор от меню :

```
switch ( ch ) {
    case 'e' : enter ( ) ;
        break ;
    case 'l' : list ( ) ;
        break ;
    case 's' : sort ( ) ;
        break ;
    case 'e' : enter ( ) ;
        break ;
    case 'q' : exit ( 0 ) ;
        break ;
    default :
        cout << "Unknown command! \n " ;
        cout << "Try again. \n " ;
}
```

template

Ключовата дума **template** се използва за създаване на шаблонни функции и класове. Типа данни опериращи със шаблонната функция или клас е определен като параметър. Така една функция или дефиниция на клас може да бъде използвана със няколко различни типове данни. Следват детайлите отнасящи се до шаблонните функции и класове. Шаблонната функция дефинира общо множество от операции което може да бъде приложено към различни типове данни. Шаблонната функция има тип данни с които ще оперира подаден като параметър. Използвайки този механизъм еднаква обща процедура може да бъде приложена към широк кръг данни. Както знаете много алгоритми са логически еднакви независимо от типа данни които обработват. Например алгоритъма **Quicksort** е еднакъв независимо дали е приложен към масив от **integer** или масив от числа със плаваща точка. Само типа данни които ще се сортират е различен. Чрез създаване на шаблонна функция може да дефинирате независимо от никакви данни естеството на алгоритъма. След като го направите, компилатора автоматично генерира правилния код за реално използваните данни когато изпълнявате функцията. Всъщност когато създавате шаблонна функция вие създавате функция която може автоматично да се предефинира. Общата форма на дефиниция на шаблонна функция е:

```
template < class тип- данни > връщан – тип
име_на_функция (списък с параметри )
{
    // тяло на функцията
}
```

Тук **тип – данни** е мястото за съхраняване на типа данни върху които в действителност ще оперира функцията. Следващата програма създава шаблонна функция която разменя стойностите на две променливи. Понеже процеса по размяната на две стойности не зависи от типа на променливите, тя е добър кандидат да бъде направена шаблонна функция:

```
// пример за шаблонна функция
#include <iostream>
using namespace std;
template < class X> void swapvals ( X &a , X &b )
{
    X temp ;
    temp = a ;
    a = b ;
    b = temp ;
}
int main ( )
{
    int i = 10 , j = 20 ;
    float x = 10.1 , y = 23.3 ;
    cout << "Original i , j : " << i << " " << j << endl ;
```

```

cout << "Original x,y: " << x << " " << y << endl;
swapvals (i,j); // разменя integer
swapvals (x,y); // разменя float
cout << "Swapped i,j: " << i << " " << j << endl;
cout << "Swapped x,y: " << x << " " << y << endl;
return 0;
}

```

В тази програма реда :

```

template <class X> void swapvals (X &a, X &b)

```

казва на компилатора две неща: първо че е създадена шаблонна функция и второ че *X* е шаблонен тип който се използва като място за съхранение. Тялото на *swapvals()* е дефинирано чрез използване на *X* като тип данни за стойностите които ще се разменят. В *main()* функцията *swapvals()* се извиква използвайки два различни типа данни: **integer** и числа с плаваща точка. Понеже *swapvals()* е шаблонна функция компилатора автоматично създава две версии на *swapvals()* – една която разменя **integer** и една която разменя **float**. Можете да дефинирате повече от един шаблонен тип използвайки в израза **template** списък разделен със запетайки. Шаблонните функции са подобни на предефинираните функции с изключение на това че са по ограничаващи. Когато се предефинира функция, можете да имате различни действия изпълнявани от тялото на всяка функция. Шаблонната функция трябва да изпълнява едно и също действие във всички версии. В добавка към шаблонните функции можете да дефинирате шаблонен клас. Когато правите това вие създавате клас който дефинира всички алгоритми използвани от този клас, но действителния тип манипулационни данни ще бъде определен като параметър когато се създават обекти от този клас. Шаблонните функции са полезни когато класа съдържа обобщаваща логика. Например един алгоритъм който поддържа опашка от **integer** ще работи също със опашка от символи. Също механизма който управлява свързан списък от пощенски адреси може да управлява списък от авточасти. Чрез използване на шаблонен клас можете да създадете клас който ще поддържа опашка, свързан списък и множество други типове данни. Компилатора ще генерира автоматично правилния тип обект базирайки се на типа който сте определили когато се създава обекта. Общата форма на декларация на общ клас е:

```

template <class тип-данни> class име_на_клас {
// ...
};

```

В този случай, **тип данни** е мястото за съхранение на типа данни със които ще оперира класа. Когато декларира обект от шаблонен клас вие определяте типа данни между ъглови скоби използвайки тази обща форма:

```

име_на_клас <тип> обект;

```

Следващия пример е шаблонен клас. Тази програма създава много прост клас за едностранно свързан списък. След това демонстрира класа чрез създаване на свързан списък който съхранява символи:

```

// прост шаблонен свързан списък
#include <iostream>

```

```

using namespace std ;
template < class data_t > class list {
    data_t data ;
    list *next ;
public :
    list (data_t d) ;
    void add ( list *node ) { node-> next = this ; next = 0 ; }
    list *getnext () { return next ; }
    data_t getdata () { return data ; }
};
template <class data_t >
list < data_t > :: list ( data_t d )
{
    data = d ;
    next = 0 ;
}
int main ()
{
    list< char > start ( 'a' ) ;
    list<char > *p , *last ;
    int i ;
    // изгражда списък
    last = &start ;
    for (i = 0 ; i < 26 ; i++) {
        p = new list<char > ('a' + i) ;
        p-> add ( last ) ;
        last = p ;
    }
    // следва списъка
    p = &start ;
    while (p) {
        cout << p->getdata () ;
        p = p->getnext () ;
    }
    return 0 ;
}

```

Както можете да видите декларацията на шаблонния клас е подобна на декларацията на шаблонна функция. Типа данни съхранявани от списъка е направен общ в декларацията на класа. В **main()** обектите и указателите са създадени така че типа данни във списъка да бъде **char**. Списъци от други типове също може да бъде създаден. Обърнете специално внимание на тази декларация :

```
list< char > start ('a') ;
```

Забележете как желания тип данни е подаден вътре в ъглови скоби .

this

this е указател към обекта който генерира извикването към член функция . На всички член функции автоматично се подава *this* указател .

throw

throw е част от подсистемата на C++ за обработване на изключения . Следва описание . Обработката на изключения е изградена от три ключови думи : *try* , *catch* и *throw* . В повечето случай , програмните изрази които искате да наблюдавате за изключения се съдържат в блок *try* . Ако изключение (грешка) стане във *try* блок , то се прехвърля (чрез *throw*) . Изключението се прихваща чрез *catch* и се обработва . Следното описание доказва това . Както беше казано всеки израз който предизвиква изключение трябва да бъде изпълнен вътре във *try* блок (функциите извикани от *try* блок могат също да прехвърлят изключение) Всяко изключение трябва да бъде прихванато чрез *catch* израз който следва непосредствено изрази *try* който прехвърля изключението . Общата форма на *try* и *catch* е :

```
try {
    // блок try
}
catch ( тип1 аргумент ) {
    // блок catch
}
catch ( тип2 аргумент ) {
    // блок catch
}
catch ( тип3 аргумент ) {
    // блок catch
}
//...
catch ( типN аргумент ) {
    // блок catch
}
```

Блока *try* трябва да съдържа частта от вашата програма която искате да наблюдавате за грешки . Това може да са няколко изрази в една функция или пълно заграждане на функцията *main ()* във *try* блок (което означава да бъде наблюдавана цялата програма) Когато се предизвика изключение , то се прихваща чрез съответния му израз *catch* който го обработва . Може да има повече от един *catch* израз свързан със *try* блока . Кой *catch* израз ще се използва се определя от типа на изключението . Така ако типа данни определен от *catch* съответства на изключението , тогава този *catch* израз се изпълнява (всички други се пропускат) Когато се прихване изключение аргумент ще получи неговата стойност . Всеки тип данни може да бъде прихванат включително и класовете които създавате . Ако не се предизвика

изключение (тоест не са се случили грешки във **try** блока), тогава никой **catch** израз не се изпълнява. Формата на израза **throw** е:

throw изключение ;

throw трябва да бъде изпълнен или във самия блок **try** или от извиквана отвътре на блока **try** функция (директно или индиректно) **Изключение** е прехвърляната стойност. Ако предизвика изключение за което няма приложен **catch** израз може да стане ненормално приключване на програмата. Предизвикването на необработвани изключения причинява извикването на функцията **terminate()**. По подразбиране **terminate()** извиква **abort()** за спиране на програмата. Обаче можете да определите собствени манипулатори ако искате, използвайки **set_terminate()**. Тук е показан прост пример за начина на обработка на изключения в C++:

```
// пример за просто манипулиране на изключения
#include <iostream.h>
using namespace std;
int main ()
{
    cout << "Start\n ";
    try { // стартира try блок
        cout << "Inside try block\n ";
        throw 100; // прехвърля грешка
        cout << "This will not execute ";
    }
    catch (int i) { // прехваща грешка
        cout << "Caught an exception – value is : ";
        cout << i << "\n ";
    }
    cout << "End ";
    return 0;
}
```

Тази програма извежда следния резултат:

```
Start
Inside try block
Caught an exception – value is : 100
End
```

Както можете да видите, във **try** блока има три израза и **catch (int i)** обработва **integer** изключение. Във **try** блока, само два от трите израза ще се изпълнят: първия **cout** израз и **throw**. Когато се предизвика изключение контрола преминава към **catch** израза и **try** блока се прекъсва. Тоест **catch** не се извиква. По скоро програмата се прехвърля към него (програмния стек автоматично установява кога трябва да се изпълни това) Така **cout** израза след **throw** никога няма да се изпълни.

true

true е булева константа за **true**.

try

try е част от механизма на C++ за обработка на изключения (виж *throw*)

typedef

Ключовата дума *typedef* ви позволява да създавате ново име за съществуващ тип данни . Типа данни може да бъде или един от вградените типове или клас , структура , обединение или изброяване . Формата на *typedef* е :

***typedef* спецификатор- на – тип ново – име ;**

Например , за да използвате думата *balance* на мястото на *float* трябва да напишете :

typedef float balance ;

typeid

В C++ оператора *typeid* връща референция към *type_info* обект който описва типа на обекта на който е приложен *typeid typeid* има тази обща форма :

***typeid* (обект)**

typeid поддържа идентификация по време на изпълнение (**RTTI**) в C++ .

Забележка : Виж “ *typeid* “ в Глава 3 .

typename

C++ поддържа ключовата дума *typename* .Тя може да бъде използвана на мястото на ключовата дума *class* във *template* декларация или за означаване на неопределен тип .

union

union е специален тип клас който слага две или повече променливи на едно и също място в паметта . Формата за дефиниране и действието на операторите точка (.) и стрелка (->) към член е еднакво като за *class* .По подразбиране членовете са *public* .Общата форма е :

***union* име_на_клас {**
// *public* членове по подразбиране
***private* :**
// *private* членове
} списък – с –обекти ;

Име_на_клас е името на типа обединение .

Забележка : В C , обединенията могат да съдържат само член променливи и спецификатора *private* не е позволен . Така C- обединенията могат да съдържат само данни . Например следващия пример създава обединение между *double* и символен низ и създава променлива наречена *my_var* :

union my_union {
char time [30] ;
double offset ;
} my_var ;

Обединенията са разгледани по детайлно в Глава 1 .

unsigned

unsigned е модификатор за тип данни който декларира **integer** без знак .
unsigned integer могат да съдържат само положителни стойности .

using

Виж *namespace* .

virtual

Спецификатора на функция **virtual** създава виртуална функция . Виртуалната функция е член на базов клас който може да бъде предефиниран от производния клас . Ако функцията не е предефинирана от производния клас се използва дефиницията във базовия клас . Чисто виртуална функция е член функция която няма дефиниция . Това означава че чисто виртуалната функция трябва да бъде предефинирана във производния клас . Чисто виртуалната функция има такъв прототип :

virtual връщан-тип име_на_функция (списък с – параметри) = 0 ;

Тук **връщан тип** е връщания тип на функцията , **име_на_функция** е името и **списък – с – параметри** определя някакви параметри . Важна характеристика е **= 0** . Това казва на компилатора че виртуалната функция няма дефиниция що се отнася до виртуалния клас . **Runtime** полиморфизма се постига когато виртуалните функции се достигат чрез указател към базов клас . Когато това е направено , типа на обекта определя коя версия на виртуалната функция ще се извика .

Програмен съвет

Клас който съдържа най – малко една чисто виртуална функция се нарича абстрактен . Абстрактния клас не може да бъде използван за създаване на обекти . Той също не може да бъде използван като тип параметър на функция или като връщан тип . Обаче може да създадете указател към абстрактен клас . Клас който наследява абстрактен клас и не предефинира всички чисто виртуални функции сам по себе си е също абстрактен . Производния клас трябва да предефинира всички чисто виртуални функции от всички свой базови класове преди да стане определен и да могат да се създават обекти от този клас .

void

Спецификатора за тип **void** е главно използван да декларира изрично че функциите нямат връщана стойност . Той също се използва за създаване на **void** указатели (указатели към **void**) които са способни да сочат към всеки

тип обекти. В C *void* се използва за деклариране на празен списък с параметри във декларация на функция.

volatile

Модификатора *volatile* казва на компилатора че променлива може да промени съдържанието си по начини не изрично определени от програмата. Например променливи които могат да бъдат променени чрез хардуера като часовника, прекъсвания или други начини, трябва да бъдат деклариращи като *volatile*.

wchar_t

wchar_t определя широко-символен тип. Широките символи са дълги 16 бита.

while

Цикъла *while* има тази обща форма:

```
while (условие) {  
    блок-сърс – изрази  
}
```

Ако има само един израз във *while* тогава скобите могат да се пропуснат. *while* проверява условието на върха на цикъла. По този начин ако условие е *false* в началото цикъла няма да се изпълни нито веднъж. *Условие* може да бъде всякакъв израз. Следващото е пример за *while* цикъл. Той ще прочете 100 символа и ще ги постави във масив от символи:

```
char s [256];  
t = 0;  
while ( t < 100 ) {  
    s [t] = stream . get ( );  
    t++;  
}
```

ГЛАВА 6 – Стандартни C I/O функции.

Тази глава описва стандартните C I/O функции. Тези функции са дефинирани чрез ANSI C стандарт и всички C компилатори са снабдени с тях в техните стандартни библиотеки. Те са също поддържани от C++ да доставят съвместимост със C и няма фундаментална причина да не ги използвате във вашите C++ програми, когато сметнете за необходимо. Понеже функциите в тази глава са специфични за ANSI C стандарта, те ще бъдат обяснени общо като ANSI C I/O системата. Хедърния файл, свързан със стандартните I/O функции е наречен **STDIO . H**. Той дефинира няколко макроса и типа, използвани от файловата система. Най-важният тип е **FILE**, който се използва да декларира файлов указател. Два други типа са *size_t* и *fpos_t*, които са друга форма на *unsigned integer*. Типа *size_t* дефинира обект, който е способен да съдържа размера на най-големия файл, разрешен от операционната среда. Типа *fpos_t* дефинира обект, който може да

съдържа информация, нужна за единствено определяне на всяка позиция във файла. **ANSI C** I/O системата оперира чрез потоци. Потока е логическо устройство, което е свързано към действително физическо устройство, което е отнесено към файла. В **ANSI C** I/O системата всички потоци имат еднакви способности, но файловете могат да имат различаващи се качества. Например дисковия файл разрешава произволен достъп, но модема-не. Така **ANSI C** I/O системата доставя ниво на абстракция между програмиста и физическото устройство. Абстракцията е поток а устройството е файла. По този начин съставния логически интерфейс може да бъде поддържан дори и действителните логически устройства да са различни. Потока се свързва към файл чрез извикване на **fopen()**. Потоците действат върху тях чрез файлов указател (който е указател от типа ***FILE**). В този смисъл файловия указател е закрепен така, че да държи системата заедно. Когато вашата програма започне изпълнение три предефинирани потока се отварят автоматично. Те са **stdin**, **stdout** и **stderr** насочени съответно към стандартния вход, стандартния изход и стандартната грешка. По подразбиране те са свързани към конзолата, но могат да бъдат преадресирани към всякакъв тип устройство. Много от стандартните библиотечни I/O функции установяват вградена глобална целочислена променлива **errno** когато се случи грешка. Вашата програма може да провери тази променлива, когато се получи грешка, за да получи повече информация за грешката. Стойностите, които **errno** може да взима зависят от изпълнението.

clearerr()

```
#include <stdio.h>
```

```
void clearerr (FILE *stream);
```

Функцията **clearerr()** пренастройва (тоест установява на нула) флага за грешка свързан със потока указан чрез **stream EOF** индикатора също се пренастройва. Флаговете за грешка на всеки поток са първоначално установени на 0 чрез успешното извикване на **fopen()**. Когато се случи грешка флаговете се установяват чрез явно извикване на една от функциите **clearerr()** или **rewind()**. Файлови грешки могат да се получат по много причини много от които са системно зависими. Точния вид грешка може да се определи чрез извикване на **perror()**, която показва информация за грешките. (Виж **perror()**) Свързани функции са **feof()**, **ferror()** и **perror()**.

fclose()

```
#include <stdio.h>
```

```
int fclose (FILE * stream);
```

Функцията **fclose()** затваря файла свързан със **stream** и изчиства неговия буфер. След **fclose()** **stream** се освобождава от файла и всеки автоматично заделен буфер се освобождава. Ако **fclose()** е успешна се връща 0 иначе се връща **EOF**. Опит за затваряне на файл който а вече бил затворен генерира грешка. Премахването на съхранените данни преди затварянето на файла също ще генерира грешка, както и липсата на достатъчно дисково пространство. Свързани функции са **fopen()**, **freopen()** и **fflush()**.

feof ()

```
#include <stdio.h>
int feof ( FILE * stream );
```

Функцията **feof ()** проверява индикатора за край на файл за да определи дали края на файла свързан със **stream** е достигнат. Ненулева стойност се връща ако индикатора за край на файл е в края, 0 се връща в противен случай. Веднъж когато е достигнат края на файл следващите четящи операции ще върнат **EOF** докато е извикана една от двете функции **rewind ()** е извикана или позициониращия индикатор е преместен със **fseek ()**. Макроса **EOF** е дефиниран в **STDIO.H**. **feof ()** е особено полезна когато работим със двоични файлове защото маркера за край на файл е също валидно двоично цяло число. Изрично извикване трябва да бъде направено отколкото просто проверяване на върнатата стойност от **getc ()** например за да определим дали е достигнат край на файл. Свързани функции са **clearer ()**, **ferorr ()**, **perror ()**, **putc ()** и **getc ()**.

ferror ()

```
#include <stdio.h>
int ferror ( FILE * stream );
```

Функцията **ferror ()** проверява за грешки файла подаден ѝ във **stream**. Върната стойност 0 показва че не е станала грешка, докато ненулева стойност означава грешка. Флаговете за грешка свързани със **stream** ще останат вдигнати докато или файла се затвори или **rewind ()** или **clearer ()** се извикат. За определяне на точната природа на грешката използвайте **perror ()**. Свързани функции са **clearer ()**, **feof ()** и **perror ()**.

fflush ()

```
#include <stdio.h>
int fflush ( FILE * stream );
```

Ако **stream** е свързан със файл отворен за запис извикването на **fflush ()** ще предизвика съдържанието на изходния буфер да бъде физически записан във файла. Ако **stream** указва входен файл, съдържанието на входния буфер ще се изчисти. Във всички случаи файла остава отворен. Върната стойност 0 означава успех, **EOF** показва че е станала грешка при запис. Всички буфери се изчистват при нормално прекъсване на програмата или когато се напълнят. Също затварянето на файл изчиства неговия буфер. Свързани функции са **fclose ()**, **fopen ()**, **fread ()**, **fwrite ()**, **getch ()** и **putc ()**.

fgetc ()

```
#include <stdio.h>
int ( FILE * stream );
```

Функцията **fgetc ()** връща следващия символ от входния поток на текущата позиция и увеличава позиционния файлов индикатор. Символа се чете като **unsigned char** и тогава се преобразува в цяло число. Ако е достигнат края на файла **fgetc ()** връща **EOF**. Обаче откакто **EOF** е валидна цяла стойност когато работим с двоични файлове трябва да използвате **feof ()** за да проверите дали е достигнат край на файл. Ако **fgetc ()** срещне грешка, **EOF**

се също връща. Ако работите със двоични файлове трябва да използвате *ferror()* за проверка за грешки във файла. Свързани функции са *fputc()*, *getc()*, *putc()* и *fopen()*.

fgetpos()

```
#include <stdio.h>
```

```
int fgetpos ( FILE * stream , fpos_t position );
```

Функцията *fgetpos()* съхранява текущата стойност на файловия позициониращ индикатор в обект указван от *position*. Обекта указван от *position* трябва да бъде от тип *fpos_t* който е дефиниран в *STDIO.H*. Стойността съхранявана там е полезна само в следващото извикване на *fgetpos()*. Ако стане грешка *fgetpos()* връща нула, в противен случай връща 0. Свързани функции са *fsetpos()*, *fseek()* и *ftell()*.

fgets()

```
#include <stdio.h>
```

```
char * fgets ( char * str , int num , FILE * stream );
```

Функцията *fgets()* чете *num - 1* символа от *stream* и ги поставя в масив от символи указван от *str*. Символи се четат докато нов ред или **EOF** се получи или докато е достигнат определения лимит. След като се прочетат символите се поставя 0 в масива веднага след последния прочетен символ. Символа за нов ред ще бъде запазен и ще бъде част от *str*. Ако е успешна *fgets()* връща *str*, нулев указател се връща при срыв. Ако стане грешка при четене съдържанието на масива указван от *str* ще е неопределено. Понеже нулев указател ще се върне или когато стане грешка или когато е достигнат край на файла ще трябва да използвате *feof()* или *ferror()* за да определите какво действително е станало. Свързани функции са *fputs()*, *fgetc()*, *getc()* и *puts()*.

fopen()

```
#include <stdio.h>
```

```
FILE * fopen ( const char * fname , const char * mode );
```

Функцията *fopen()* отваря файл чието име е указано от *fname* и връща поток който е свързан с него. Типа на разрешените операции са определени от стойността на *mode*. Разрешените стойности за *mode* са показани в следващата таблица. Файловото име трябва да бъде низ от символи съставлящи валидно име определено от оперционната система и може да включва специфични пътища ако средата ги поддържа.

режим

“r”

“w”

“a”

“rb”

“wb”

“ab”

“r+”

значение

Отваря текстов файл за четене

Създава текстов файл за писане

Добавя към текстов файл

Отваря двоичен файл за четене

Създава двоичен файл за запис

Добавя към двоичен файл

Отваря текстов файл за четене / запис

“ w+ “	Създава текстов файл за четене / запис
“ a+ “	Отваря текстов файл за четене / запис
“ rb+ “	Отваря двоичен файл за четене / запис
“ wb+ “	Създава двоичен файл за четене / запис
“ ab+ “	Отваря двоичен файл за четене / запис

Ако *fopen()* е успешна в отварянето на определения файл се връща указател **FILE**. Ако файла не може да се отвори се връща нулев указател. Както показва таблицата файл може да бъде отворен или текстов или в двоичен режим. В текстов режим могат да станат някои символни преобразувания. Например нов ред може да бъде преобразуван във връщане на каретка / начало на ред. Никакви подобни преобразувания не стават във двоичните файлове. Правилния начин за отваряне на файл е илюстриран със следния фрагмент от код:

```
FILE *fp;
if ( (fp == fopen("test", "w")) == NULL ) {
    printf("Cannot open file.\n");
    exit(1);
}
```

Този метод открива всяка грешка при отварянето на файл като например защитен срещу запис или пълен диск, преди да опита да пише в него. **NULL** се използва да покаже грешка понеже няма файлов указател който да има такава стойност. **NULL** е дефиниран в **STDIO.H**. Ако използвате *fopen()* да отвори файл за изход всеки предишен файл с това име ще бъде изтрит и ще се запише новия файл. Ако няма файл със такава име той ще бъде създаден. Ако искате да добавите към края на файл трябва да използвате режим “a”. Ако файла не съществува ще бъде върната грешка. Отварянето на файл за четене изисква файла да съществува. Ако не съществува ще се върне грешка. Накрая ако файла е отворен за четене / запис той няма да бъде изтрит ако съществува, обаче ако не съществува ще бъде създаден. Когато имате достъп до файл отворен за четене / запис не можете да редувате изходни със входни операции без обръщане към функциите *fflush()*, *fseek()*, *fsetpos()* или *rewind()*. Също не можете да редувате входна операция със изходна без извикване на една от предишните функции. Свързани функции са *fclose()*, *fread()*, *fwrite()*, *putc()* и *getc()*.

Програмен съвет

Всеки файл може да бъде отворен или като текстов или като двоичен. Няма значение какво в действителност съдържа файла. Например файл който съдържа **ASCII** текст може също да бъде отворен и обработван като двоичен файл. Що се отнася до **ANSI C** файловата система единствената разлика между текстовия и двоичния файл е че в него няма символни преобразувания които ще се заместят когато оперирате върху файл отворен в двоичен режим.

Вие може да искате да отворите файл който съдържа текст като двоичен когато изпълнявате различни не – текстово базирани манипулации върху него . Например за файловете действия като сравняване , компресиране или сортиране ще трябва да имате достъп до файла в двоичен режим . Също файловете кодиращи програми почти винаги ще трябва да работят в двоичен режим . Ключовата точка е че разликата между текстовия и двоичния файл не е какво съдържа файла , а по скоро начина по който сте го отворили .

fprintf ()

```
#include <stdio . h >
```

```
int fprintf ( FILE *stream , const char *format , ... ) ;
```

Функцията *fprintf ()* извежда стойностите на аргументите които съставят списъка със аргументи във низа *format* във поток указван от *stream* .

Върнатата стойност е броя на действително записаните символи . Ако стане грешка се връща отрицателно число . Може да има от 0 до няколко аргумента с максимален брой зависещ от системата . Операциите за форматен контрол и командите са идентични на тези във *printf ()* (виж “*printf*“ за пълно описание). Свързани функции са *printf ()* и *scanf ()* .

fputc ()

```
#include <stdio . h >
```

```
int fputc ( int ch , FILE *stream ) ;
```

Функцията *fputc ()* записва символа *ch* във определения поток във текущата позиция във файла и след това увеличава индикатора на позицията . Въпреки че *ch* е декларирана да бъде *int* по исторически причини , тя се преобразува от *fputc ()* във *unsigned char* понеже всички символни аргументи са повишени в цели числа по време на тяхното извикване , вие сте свободни да използвате символни променливи като аргументи . Ако използвате число , старшият байт просто ще бъде отхвърлен . Стойността върната от *fputc ()* е стойността на записания символ . Ако стане грешка се връща **EOF** . За файлове отворени за двоични операции , **EOF** може да е валиден символ и ще ви трябва функцията *ferror ()* за да определите дали в действителност е станала грешка . Свързани функции са *fgetc ()* , *fopen ()* , *fprintf ()* , *fread ()* и *fwrite ()* .

fputs ()

```
#include <stdio . h >
```

```
int fputs ( const char *str , FILE *stream ) ;
```

Функцията *fputs ()* записва съдържанието на низа указван от *str* в определения поток . Нулевия терминатор не се записва . Функцията *fputs ()* връща неотрицателна стойност при успех и **EOF** при грешка . Ако потока е отворен в текстови режим ще станат определени символни преобразувания . Това означава че може да няма едно към едно съпоставяне на низа във файла . Обаче ако потока е отворен в двоичен режим никакви символни преобразувания няма да има и ще има едно към едно съпоставяне между низа и файла . Свързани функции са *fgets ()* , *gets ()* , *puts ()* , *fprintf ()* и *fscanf ()* .

fread ()

```
#include <stdio.h>
int fread ( void *buf, size_t size, size_t count,
            FILE *stream );
```

Функцията **fread ()** чете **count** брой обекти, всеки обект със **size** байта дължина от поток указван от **stream** и ги поставя в масив указван от **buf**. Индикатора за файлова позиция се увеличава със броя на прочетените символи. Функцията **fread ()** връща броя на действително прочетените обекти. Ако са прочетени по-малко обекти от заявените при извикването, или е станала грешка или е достигнат края на файла. Трябва да използвате **feof ()** или **ferror ()** за да определите какво е станало. Ако потока е отворен за текстови операции могат да станат определени символни преобразувания като връщане на каретка / начало на нов ред. Свързани функции са **fwrite ()**, **fopen ()**, **fscanf ()**, **fgetc ()** и **getc ()**.

freopen ()

```
#include <stdio.h>
FILE *freopen ( const char fname, const char *mode, FILE
                *stream );
```

Функцията **freopen ()** свързва съществуващ поток със различен файл. Новото файлово име е указано от **fname**, режима за достъп е указан чрез **mode**, и потока за който е предназначен е указан от **stream**. Низа **mode** използва същия формат като **fopen ()** пълно разглеждане е направено във секцията “**fopen**”. Когато е извикана, **freopen ()** първо опитва да затвори файла който може текущо да е свързан със **stream**. Обаче ако опита за затваряне на файла пропадне, **freopen ()** още продължава към отваряне на друг файл. Функцията връща указател към **stream** при успех и нулев указател в противен случай. Главната употреба на **freopen ()** е пренасочване на системно дефинираните файлове **stdin**, **stdout** и **stderr** към някакъв друг файл. Свързани функции са **fopen ()** и **fclose ()**.

fscanf ()

```
#include <stdio.h>
int fscanf ( FILE *stream, const char format, ... );
```

Функцията **fscanf ()** работи точно като функцията **scanf ()** с изключение на това че тя чете информация от потока определен от **stream**, вместо от **stdin** (виж “**scanf**” за подробности). Функцията **fscanf ()** връща броя на аргументите с действително присвоени стойности. Този брой не включва прескачаните полета. Върната стойност **EOF** значи че е станала грешка преди да е направено първото присвояване. Свързани функции са **scanf ()** и **fprintf ()**.

fseek ()

```
#include <stdio.h>
int fseek ( FILE *stream, long offset, int origin );
```

Функцията **fseek ()** установява позициониращия индикатор свързан със **stream** съгласно стойностите на **offset** и **origin**. Нейна задача е поддръжката на

операции с произволен I/O достъп *offset* е брой байтове от *origin* който е достигнат. Стойността на *origin* трябва да бъде един от тези макроси (дефинирани във **STDIO.H**) :

Име	Значение
SEEK_SET	Отместване от началото на файл
SEEK_CUR	Отместване от текущото положение
SEEK_END	Отместване от края на файл

Върната стойност 0 значи че *fseek()* е успяла. Ненулева стойност значи грешка. Можете да използвате *fseek()* да местите позициониращия индикатор навсякъде във файла, дори извън края. Обаче е грешка опита да установите индикатора преди началото на файла. Функцията *fseek()* изчиства флага за край на файл свързан със определения поток. Нещо повече, тя анулира всякаква по-ранна *ungetc()* на същия поток (виж “*ungetc()*”). Свързани функции са *ftell()*, *rewind()*, *fopen()*, *fgetpos()* и *fsetpos()*.

fsetpos()

```
#include <stdio.h>
int fsetpos ( FILE *stream , const fpos_t *position ) ;
```

Функцията *fsetpos()* премества позициониращия индикатор към точка определена от обекта указван от *position*. Тази стойност трябва да бъде получена преди това чрез извикване на *fsetpos()*. Типа *fpos_t* е дефиниран в **STDIO.H**. След изпълнението на *fsetpos()* индикатора за край на файл се пренастройва. Също всяко предишно извикване на *ungetc()* се анулира. Ако *fsetpos()* пропадне се връща ненула, ако е успешна връща 0. Свързани функции са *fgetpos()*, *fseek()* и *ftell()*.

ftell()

```
#include <stdio.h>
long ftell ( FILE *stream ) ;
```

Функцията *ftell()* връща текущата стойност на позиционния индикатор на определения поток. В случай на двоични потоци стойността е броя на байтовете до индикатора от началото на файла. За текстови потоци връщаната стойност може да не е от главно значение с изключение за аргумент на *fseek()* понеже възможните символни преобразувания такива като връщане на каретка / начало на ред ще бъдат заместени с нови редове което влияе върху действителната големина на файла. Функцията *ftell()* връща -1 когато стане грешка. Ако потока е неспособен за произволно търсене - ако е модем например – връщаната стойност е неопределена. Свързани функции са *fseek()* и *fgetpos()*.

fwrite()

```
#include <stdio.h>
int fwrite ( const void *buf , size_t size , size_t count ,
FILE *stream ) ;
```

Функцията *fwrite()* записва *count* броя обекти, всеки обект имащ *size* байта дължина в поток указван от *stream* от символния масив указван от *buf*. Позиционния индикатор се увеличава с броя на записаните символи.

Функцията *fwrite()* връща броя на действително записаните обекти които ако функцията е успешна ще са равни на заявения брой. Ако са записани по – малко обекти от заявените се е получила грешка. За текстови потоци могат да станат различни символни преобразувания но те няма да имат ефект върху връщаната стойност. Свързани функции са *fread()*, *fscanf()*, *getc()* и *fgetc()*.

getc ()

```
#include <stdio.h>
int getc ( FILE *stream );
```

Функцията *getc()* връща следващия символ от входния поток на текущата позиция и увеличава позиционния индикатор. Символа се чете като *unsigned char* и се конвертира в *integer*. Ако е достигнат край на файл *getc()* връща **EOF**. Обаче откакто **EOF** е валидна целочислена стойност, когато работите със двоични файлове вие трябва да използвате *feof()* за проверка за символа за край на файл. Ако *getc()* срещне грешка също се връща **EOF**. Ако работите със двоични файлове трябва да използвате *ferror()* за да проверите за грешки. Функциите *getc()* и *fgetc()* са идентични и в повечето изпълнения *getc()* е просто дефинирана като макроса показан тук:

```
#define getc (fp) fgetc (fp)
```

Това предизвиква функцията *fgetc()* да бъде заместена със *getc()* макроса. Свързани функции са *fputc()*, *fgetc()*, *putc()* и *fopen()*.

getchar ()

```
#include <stdio.h>
int getchar ( void );
```

Функцията *getchar()* връща следващия символ от *stdin*. Символа се чете като *unsigned char* и се конвертира в *integer*. Ако е достигнат края на файла *getchar()* връща **EOF**. Обаче откакто **EOF** е валидна цяла стойност когато работите със двоични файлове ще трябва да използвате *feof()* за да провери за края на файла. Ако *getchar()* срещне грешка се също връща **EOF**. Ако работите със двоични файлове трябва да използвате *ferror()* да проверите за грешки. Функцията *getchar()* често се изпълнява като макрос. Свързани функции са *fputc()*, *fgetc()*, *putc()* и *fopen()*.

gets ()

```
#include <stdio.h>
char *gets (char *str);
```

Функцията *gets()* чете символи от *stdin* и ги поставя в масив от символи указван от *str*. Символи се четат докато се получи нов ред или **EOF**. Символа за нов ред не е част от низа, вместо това той се преобразува в 0 за прекъсване на низа. Ако е успешна *gets()* връща *str*, при грешка се връща нулев указател. Ако стане грешка съдържанието на масива указван от *str* е неопределено. Понеже нулев указател ще бъде върнат или когато стане грешка или когато е достигнат края на файла, ще трябва да използвате *feof()* или *ferror()* за да определите какво е станало в действителност. Няма начин да ограничите броя на символите които *gets()* ще прочете и по тази причина ще трябва да направите така че масива указван от *str* да не бъде

препълнен (Вижте “Програмен съвет“ по долу). Свързани функции са *fputc ()* , *fgetc ()* , *fgets ()* и *puts ()* .

Програмен съвет

Когато използвате *gets ()* е възможно да препълните масива който се използва да получи въведените символи понеже *gets ()* не прави проверка на границите . Един начин да заобиколите този проблем е да използвате *fgets ()* определяйки *stdin* за входен поток . Откакто *fgets ()* изисква да определите максимална дължина е възможно да избегнете препълване на масива . Единствения проблем е че *fgets ()* не премахва символа за нов ред , така че вие ще трябва ръчно да го премахнете както е показано в следната програма :

```
#include <stdio.h>
#include <string.h>
int main ( void )
{
    char str [10];
    int i;
    printf ( “ Enter a string : “ );
    fgets ( str , 10 , stdin );
    /* премахва нов ред ако се срещне */
    i = strlen ( str ) - 1;
    if ( str [ i ] == ‘ \n ’ ) str [ i ] = ‘ \0 ’ ;
    printf ( “ This is your string : %s “ , str );
    return 0 ;
}
```

Въпреки че употребата на *fgets ()* изисква малко повече работа нейното предимство пред *gets ()* е че можете да избегнете препълване на входния масив .

perror ()

```
#include <stdio.h>
void perror ( const char *str );
```

Функцията *perror ()* свързва стойността на глобалната променлива *errno* към низ и записва този низ в *stderr* . Ако стойността на *str* не е 0 , низа се записва първи следван от двоеточие и тогава зависимо от изпълнението съобщение за грешка .

printf ()

```
#include <stdio.h>
int printf ( const char *format , ... );
```

Функцията *printf ()* записва в *stdout* аргументите които съставят списъка ѝ от аргументи като определят низа указван от *format* . Низа указван от *format* се състои от два типа съставлящи . Първия тип е съставен от символи които ще бъдат изведени на екрана . Втория тип съдържа форматни команди които определят начина на показване на аргументите . Форматните команди започват със знака за процент и след това форматен код . Трябва да има точно същия

брой аргументи към форматните команди и да се свързват по ред. Например следното извикване на **printf()** показва “**Hi c 10 there!**”:

printf(“Hi %c %d %s “, ‘c’, 10, “there!”);

Ако има несъответстващи аргументи отговарящи на форматните команди, изхода е неопределен. Ако има повече аргументи отколкото форматни команди останалите аргументи се отхвърлят. Форматните команди са показани тук:

Код	Формат
%c	Символ
%d	Десетично цяло число със знак
%i	Десетично цяло число със знак
%e	Научно означение (малко e)
%E	Научно означение (голямо E)
%f	Число със десетична точка
%g	Използва %e или %f което е по – късо (ако %e използва малко e)
%G	Използва %E или %f което е по – късо (ако %e използва голямо E)
%o	Осмично число без знак
%s	Низ от символи
%u	Десетично цяло число без знак
%x	Шестнайсетично цяло число без знак (малки букви)
%X	Шестнайсетично цяло число без знак (големи букви)
%p	Показва указател
%n	Свързания аргумент трябва да бъде указател към integer в който са поставени символи
%%	Отпечатва %

Функцията **printf()** връща броя на действително отпечатаните символи. Отрицателна върната стойност показва че е станала грешка. Форматните команди могат да имат модификатори които определят полетата ширина, точност и ляво – подравнителен флаг. Цяло число поставено между % и форматната команда действа като спецификатор за минимална големина на полето. Това запълва изхода със интервали или 0 като осигурява най – малкия определен минимум дължина. Ако низа от числа е по – голям от минимума той ще бъде отпечатан целия дори ако препълва минимума. Подразбиращото запълване е направено със интервали. Ако искате запълване със 0 сложете 0 преди спецификатора за ширина. Например **%05d** ще запълни число с по – малко от 5 цифри с 0 така че неговата обща дължина да е 5. Точното значение на ограничителя за точност зависи от форматния код който ще се изменя. За да добавите ограничител за точност сложете десетична точка следвана от точността след спецификатора за ширина на полето. За **e**, **E** и **f** форматите ограничителя определя числата след десетичната точка. Например

%10.4f ще покаже най – малко 10 символна големина с четири цифри след точката . Когато ограничителя за точност е добавен към **g** или **G** форматни кодове , той определя максимално число на показаните означени цифри . Когато е добавено към **integer** ограничителя определя минималния брой числа които ще бъдат показани . Водещите 0 се добавят ако е необходимо . Когато точностния ограничител е добавен към низ числото следвано от периода определя максималната дължина на полето . Например **%5.7s** ще покаже низ който ще е най – малко 5 символа дълъг и няма да надхвърля 7 . Ако низа е по – дълъг от максималната ширина на полето символите ще бъдат отрязани в края . По подразбиране целия изход е дясно подравнен ; ако ширината на полето е по – голяма от отпечатваните данни , данните ще бъдат поставени от десния край на полето . Вие можете да направите информацията да бъде ляво подравнена чрез поставяне на знак минус точно след % . Например **% - 10.2f** ще е ляво подравнено число с плаваща точка и със две числа след точката в поле със 10 символа . Има две форматни командни ограничителя които позволяват **printf()** да показва къси и дълги **integers** . Тези модификатори може да бъдат добавени към **d, i, o, u** и **x** типовете спецификатори . Модификатора **l** казва на **printf()** че следва дълъг тип данни . Например **%ld** значи че ще бъде показано дълго цяло число . **h** модификатора казва на **printf()** да покаже късо цяло число . Така **%hu** показва че данните са от тип късо цяло число без знак . **l** модификатора може също да е пред командите **e, f** и **g** и показва че следва **double** . Изхода за **long double** използва **%L** префикс . Командата **%n** педизвиква броя на символите които ще се отпечата когато се срещне **%n** да се постави в целочислена променлива която е указана чрез нейния съответстващ аргумент . Например този фрагмент от код показва число 14 след линията “**This is a test**“ :

```
int i;
printf( "This is a test %n ", &i );
printf( "%d ", i );
```

Символа **#** има специално значение когато се използва с някой **printf()** форматни кодове . Предшествайки **g, f** и **e** кода с **#** осигурява че десетичната точка ще се покаже , дори ако няма числа след нея . Ако поставите пред **x** формата **#** шестнайсетичното число ще бъде отпечатано със префикс **0x** . Ако поставите пред формата **o #** , осмичната стойност ще бъде отпечатана със префикс **0 . #** не се добавя към никакви други форматни спецификатори . Минималната големина на полето и спецификаторите за точност могат да бъдат доставени чрез аргументи на **printf()** вместо чрез константи . За да постигнете това използвайте ***** като съхранител . Когато форматния стринг е сканиран , **printf()** ще съпостави всеки ***** към аргумента в реда в който се срещат . Свързани функции са **scanf()** и **fprintf()** .

```
putc ( )
#include <stdio . h >
int putc ( int ch , FILE *stream ) ;
```

Функцията **putc()** записва символа съдържан в най – малкия означен байт на **ch** към изходния поток указван от **stream** . Понеже символните аргументи се издигат във **integer** по време на тяхното извикване , можете да използвате

символни променливи като аргументи на *putc()*. Функцията *putc()* връща записания символ при успех или **EOF** ако е станала грешка. Ако изходния поток е бил отворен в двоичен режим, **EOF** е валидна стойност за *ch*. Това означава че трябва да използвате *ferror()* за да определите дали е станала грешка. Свързани функции са *fgetc()*, *fputc()*, *getchar()* и *putchar()*.

putchar ()

```
#include <stdio.h>
int putchar ( int ch );
```

Функцията *putchar()* записва символа съдържан в най – малкия означен байт на *ch* в *stdout*. Тя е функционално еквивалентна на *putc(ch, stdout)*. Понеже символните аргументи се издигат в **integer** по време на тяхното извикване, вие може да използвате символни променливи като аргументи на *putchar()*. Функцията *putchar()* връща записания символ при успех или **EOF** при грешка. Ако изходния поток е бил отворен в двоичен режим, **EOF** е валидна стойност за *ch*. Това означава че вие трябва да използвате *ferror()* да определите дали е станала грешка. Свързана функция е *putc()*.

puts ()

```
#include <stdio.h>
int puts ( char *str );
```

Функцията *puts()* записва низа указван от *str* към стандартно изходно устройство. Нулевия терминатор се преобразува в нов ред. Функцията *puts()* връща неотрицателна стойност ако е успешна и **EOF** при срыв. Свързани функции са *putc()*, *gets()* и *printf()*.

remove ()

```
#include <stdio.h>
int remove ( const char *fname );
```

Функцията *remove()* изтрива файла определен от *fname*. Тя връща 0 ако файла е успешно изтрит и ненула ако е станала грешка. Свързана функция е *rename()*.

rename ()

```
#include <stdio.h>
int rename ( const char *oldfname, const char *newfname );
```

Функцията *rename()* променя името на файла определено от *oldfname* на *newfname*. *Newfname* не трябва да е никакво съществуващо име. Функцията връща 0 ако е успешна и ненула ако е станала грешка. Свързана функция е *remove()*.

rewind ()

```
#include <stdio.h>
void rewind ( FILE *stream );
```

Функцията *rewind()* премества позициониращия индикатор към началото на определения поток. Тя също изчиства края на файла и флаговете за грешка свързани със *stream*. Не връща стойност. Свързана функция е *fseek()*.

scanf ()

```
#include <stdio . h >
```

```
int scanf ( const char *format , ... ) ;
```

Функцията *scanf ()* е общо възприетия начин за вход който чете потока *stdin* и съхранява информацията в променливи указвани от списъка със аргументи . Тя може да чете всички вградени типове данни и автоматично да ги преобразува в подходящия вътрешен формат . Контролния низ указван от *format* е съставен от три класификации за символи :

Форматни спецификатори

Символи за празни пространства

Символи за непразни пространства

Форматните спецификатори се предшестват от знака % и казват на *scanf ()* какъв тип данни да бъде прочетен след това . Например *%s* прочита низ докато *%d* прочита **integer** . Кодовете на *scanf ()* се съпоставят в ред със променливите получаващи входа в аргументния списък . Тези кодове са изброени в следната таблица :

код	значение
<i>%c</i>	Чете единствен символ
<i>%d</i>	Чете десетично цяло число
<i>%i</i>	Чете цяло число
<i>%e</i>	Чете число с точка
<i>%f</i>	Чете число с точка
<i>%g</i>	Чете число с точка
<i>%o</i>	Чете осмично число
<i>%s</i>	Чете низ
<i>%x</i>	Чете шестнайсетично число
<i>%p</i>	Чете указател
<i>%n</i>	Получава целочислена стойност равна на броя на прочетените досега символи
<i>%u</i>	Чете цяло число без знак
<i>%[]</i>	Сканира за множество символи
<i>%%</i>	Чете знака %

Форматния низ се чете отляво надясно и форматните кодове се съпоставят в ред с аргументите които включва списъка със аргументи . Празните символи във форматния низ предизвикват *scanf ()* да прескача през един или повече празни символа във входния поток . Празния символ е или интервал или табулация или нов ред . Всъщност един празен символ в контролния низ ще предизвика *scanf ()* да прочете но да не запази никакво число (включително 0) от празните символи до първия непразен символ . Непразен символ във форматния низ предизвиква *scanf ()* да прочете и да отхвърли съответния символ . Например *%d , %d* предизвиква *scanf ()* първо да прочете **integer** , след това да прочете и да отхвърли запетайката и накрая да прочете друго **integer** . Ако определения символ не е открит , *scanf ()* ще прекъсне . Всички променливи използвани да получават стойности чрез *scanf ()* трябва да са подадени по адрес . Това значи че всички аргументи трябва да бъдат

указатели към променливите получаващи вход. Обектите от данни осъществяващи вход трябва да бъдат разделени чрез интервали, табулации или нови редове. Пунктуациите като запетайки, точки и запетайки и други като тях не се броят като разделители. Това означава че:

```
scanf( “ %d%d “, &r, &c );
```

ще приеме вход **10 20** но ще пропадне със **10, 20**. * поставен след % и преди форматния код ще прочете данните от определения тип но ще подтисне присвояванията им. Така следната команда:

```
scanf( “ %d %*c %d “, &x, &y );
```

при подаден вход **10/20** ще постави стойност 10 в **x**, ще отхвърли знака за деление и ще даде на **y** стойност 20. Форматните команди могат да определят модификатора за максимална дължина на полето. Това е число поставено между % и форматния код който ограничава броя на символите прочетени за всяко поле. Например, ако искате да прочетете не повече от 20 символа в **address**, тогава ще трябва да напишете:

```
scanf( “%20s “, address );
```

Ако входния поток има повече от 20 символа, следващото извикване на входа ще започне от там където е прекъснало. Вход за поле може да прекъсне преди достигане на максималната дължина на полето ако се срещне празен интервал. В този случай, **scanf()** се премества на следващото поле. Въпреки че интервалите, табулациите и новите редове се използват като разделители за полета, когато четем единичен символ, те се четат като всеки друг символ. Например със входен поток **x, y**

```
scanf( “ %c%c%c “, &a, &b, &c );
```

ще върне символ **x** във **a**, интервал във **b** и символ **y** във **c**. Внимавайте всеки друг символ в контролния низ, включително интервали, табулации и нови редове, ще бъде използван за съпоставяне и отхвърляне на символи от входния поток. Всеки символ който се срещне ще бъде отхвърлен. Например, подавайки на входния поток **10t20**,

```
scanf( “%st%s “, &x, &y );
```

ще постави 10 в **x** и 20 в **y**. **t** ще се отхвърли понеже **t** е в контролния низ. Друга черта на **scanf()** е наречена множество за сканиране. Множеството дефинира множество от символи които могат да бъдат прочетени от **scanf()** и присвоени на съответстващия символен масив. Множеството за сканиране е дефинирано чрез поставяне на символите които искате да сканирате вътре в квадратни скоби. Началната скоба трябва да има префикс %. Например, това множество за сканиране казва на **scanf()** да чете само символите **A, B** и **C**:

```
%[ABC]
```

Когато е използвано множество за сканиране **scanf()** продължава да чете символи и да ги слага в съответстващия символен масив докато не срещне символ който не е в множеството. Съответната променлива трябва да бъде указател към символен масив. След връщане на **scanf()** масива ще съдържа нулево – терминиран низ състоящ се от прочетените символи. Можете да обърнете множеството ако първия символ в множеството е ^. Когато е подаден ^, той информира **scanf()** да приеме всеки символ който не е

дефиниран в множеството. Вие можете да определите област използвайки тире (-). Например, това казва на *scanf()* да приеме символите от *A* до *Z*:

```
%[A-Z]
```

Важно е да запомните че множеството за сканиране е прецизен случай. По тази причина ако искате да сканирате и за големи и за малки букви, те трябва да бъдат определени поотделно. Функцията *scanf()* връща число равно на броя на полетата със успешно присвоени стойности. Този брой включва полетата които са прочетени но не са присвоени защото *ограничителя е използван да забрани присвояването. EOF се връща ако стане грешка преди първото поле да е присвоено. Свързани функции са *printf()* и *fscanf()*.

setbuf ()

```
#include <stdio.h>
```

```
void setbuf ( FILE *stream , char *buf );
```

Функцията *setbuf()* се използва или да определи буфера който ще използва определения поток или ако е извикана със установен *buf* на 0, ще изключи буферирането. Ако дефинирания от програмиста буфер е бил определен, той трябва да бъде BUFSIZE символа дълъг. BUFSIZE е дефиниран в **STDIO.H**. Функцията *setbuf()* не връща стойност. Свързани функции са *fopen()*, *fclose()* и *setvbuf()*.

setvbuf ()

```
#include <stdio.h>
```

```
int setvbuf ( FILE *stream , char *buf , int mode ,  
size_t size );
```

Функцията *setvbuf()* позволява на програмиста да определи буфера, неговата големина, и режима за определения поток. Символния масив указван от *buf* се използва като поточния буфер за I/O операции. Големината на буфера се определя от *size*, а *mode* определя как ще бъде манипулирано буферирането. Ако *buf* е 0, *setvbuf()* ще отдели собствен буфер. Позволените стойности на *mode* са **_IOFBF**, **_IONBF** и **_IOLBF**. Те са дефинирани в **STDIO.H**. Когато *mode* е установено на **_IOFBF** ще има пълно буфериране. Ако *mode* е **_IOLBF**, потока ще бъде буфериран по редове, което означава че буфера ще бъде записван всеки път когато се срещне символ за нов ред във изходния поток, за входен поток, входа изисква прочитането на всички символи до новия ред. Във всеки случай буфера се записва когато се напълни. Ако режима е **_IONBF** няма буфериране. Стойността на *size* трябва да бъде по-голяма от 0. Функцията *setvbuf()* връща 0 при успех и ненула при срыв. Свързана функция е *setbuf()*.

sprintf ()

```
#include <stdio.h>
```

```
int sprintf ( char *buf , const char *format , ... );
```

Функцията *sprintf()* е идентична на *printf()* с изключение на изхода който се слага в масив указван от *buf*, вместо да бъде записан на конзолата. Виж *printf()* за детайли. Върнатата стойност е равна на броя символи действително поставени в масива. Свързани функции са *printf()* и *fsprintf()*.

sscanf ()

```
#include <stdio.h>
```

```
int sscanf( const char *buf, const char *format, ... );
```

Функцията *sscanf()* е идентична на *scanf()* с изключение на това че данните се четат от масив указван от *buf*, а не от *stdin*. Виж *scanf()* за детайли. Връщаната стойност е равна на броя променливи със действително присвоени стойности. Този брой не включва полетата които са били прескочени чрез употребата на *ограничител. Стойност 0 значи че няма присвоено поле, а EOF показва че е станала грешка при първото присвояване. Свързани функции са *scanf()* и *fscanf()*.

tmpfile ()

```
#include <stdio.h>
```

```
FILE *tmpfile ( void );
```

Функцията *tmpfile()* отваря временен файл за обновяване и връща указател към потока. Функцията автоматично използва уникално файлово име за да избегне конфликти с имената на съществуващи файлове. Функцията *tmpfile()* връща нулев указател при срыв, в противен случай връща указател към потока. Временния файл създаден чрез *tmpfile()* се премахва автоматично когато файла се затвори или програмата се прекъсне. Свързана функция е *tmpnam()*

tmpnam()

```
#include <stdio.h>
```

```
char *tmpnam ( char *name );
```

Функцията *tmpnam()* генерира уникално файлово име и го съхранява в масив указван от *name*. Главната задача на *tmpnam()* е да генерира временно файлово име което е различно от другите в текущата дискова директория. Функцията може да бъде извикана TMP_MAX пъти. TMP_MAX е дефиниран в **STDIO.H** и ще бъде не по-малко от 25. Всеки път когато *tmpnam()* е извикана, тя ще генерира ново временно файлово име. Указател към *name* се връща при успех, в противен случай се връща нулев указател. Свързана функция е *tmpfile()*.

ungetc ()

```
#include <stdio.h>
```

```
int ungetc ( int ch, FILE *stream );
```

Функцията *ungetc()* връща символа определен от нискоразредния байт на *ch* във входния поток *stream*. Този символ ще бъде получен чрез следващата операция за четене в *stream*. Извикването на *fflush()* или *fseek()* премахва *ungetc()* операцията и отхвърля символа. Едносимволно връщане е гарантирано, обаче някои изпълнения ще допуснат повече. Вие не може да не получите EOF. Извикването на *ungetc()* изчиства флага за край на файла свързан със определения поток. Стойността на позициониращия индикатор за текстови поток е неопределена докато всички сложени отзад символи са прочетени в който случай то ще е същото като след първото *ungetc()* извикване. За двоични потоци, всяко *ungetc()* извикване намалява

позициониращия файлов индикатор. Връщаната стойност е равна на *ch* при успех и **EOF** при срыв. Свързана функция е *getc()*.

vprintf () , vfprintf () и vsprintf ()

```
#include <stdarg.h>
#include <stdio.h>
int vprintf (char *format, va_list arg_ptr);
int vfprintf (FILE *stream, const char *format, va_list
               arg_ptr);
int vsprintf (char *buf, char *format, va_list arg_ptr);
```

Функциите *vprintf()*, *vfprintf()* и *vsprintf()* са функционално еквивалентни съответно на *printf()*, *fprintf()* и *sprintf()* с изключение на списъка със аргументи който ще е заместен с указател към списък с аргументи. Този указател трябва да бъде от тип *va_list* и е дефиниран в **STDARG.H**. Свързани функции са *va_arg()*, *va_start()* и *va_end()*.

ГЛАВА 7 – C - низови и символни функции

Стандартната библиотека на C има богато и разнообразно множество от низово и символно – манипулиращи функции. В C/C++ низа е нулево – терминиран масив от символи. Низовите функции изискват хедърния файл **STRING.H** за техните прототипи. Символните функции използват **CTYPE.H** като техен хедър файл. Понеже C/C++ няма проверка за границите при операции с масиви, то отговорност на програмиста е да предотврати препълване на масива. Пренебрегването му може да причини срыв на вашата програма. В C/C++ печатим символ е този който може да бъде показан на терминал. Това са обикновенно символите между интервал (**0x20**) и тилда (**0xFE**). Контролните символи имат стойности между 0 и **0xF** както също и **DEL (0x7F)**. По исторически причини аргументите на символните функции са цели числа. Обаче само нискоразредния байт се използва. Символните функции автоматично конвертират техните аргументи в *unsigned char*. Вие сте свободни да извикате тези функции със символни аргументи, понеже символите се автоматично издигат в цели числа по време на извикване. Хедърния файл **STRING.H** дефинира типа *size_t* който е по същество еднакъв със *unsigned*.

isalnum ()

```
#include <ctype.h>
int isalnum (int ch);
```

Функцията *isalnum()* връща ненула ако нейния аргумент е или буква от азбуката или цифра. Ако не е едно от тях се връща 0. Свързани функции са *isalpha()*, *isctrl()*, *isdigit()*, *isgraph()*, *isprint()*, *ispunct()* и *isspace()*.

isalpha ()

```
#include <ctype.h>
int isalpha (int ch);
```

Функцията *isalpha* () връща ненула ако *ch* е буква от азбуката , в противен случай се връща 0 . Какво се разбира под буква варира в езиците . За английския това са главните и малките букви от *A* до *Z* . Свързани функции са *isalnum* () , *isctrl* () , *isdigit* () , *isgraph* () , *isprint* () , *ispunct* () и *isspace* () .

isctrl ()

```
#include <ctype.h>
int isctrl (int ch);
```

Функцията *isctrl* () връща ненула ако *ch* е между 0 и *0xF* или е равна на *0x7F* (*DEL*) ; иначе връща 0 . Свързани функции са *isalnum* () , *isalpha* () , *isdigit* () , *isgraph* () , *isprint* () , *ispunct* () и *isspace* () .

isdigit ()

```
#include <ctype.h>
int isdigit (int ch);
```

Функцията *isdigit* () връща ненула ако *ch* е цифра , тоест между 0 и 9 . В противен случай връща 0 . Свързани функции са *isalnum* () , *isalpha* () , *isctrl* () , *isgraph* () , *isprint* () , *ispunct* () и *isspace* () .

isgraph ()

```
#include <ctype.h>
int isgraph (int ch);
```

Функцията *isgraph* () връща ненула ако *ch* е всеки печатим символ различен от интервал ; иначе се връща 0 . Печатимите символи са обикновено в интервала от *0x21* до *0x7E* . Свързани функции са *isalnum* () , *isalpha* () , *isctrl* () , *isdigit* () , *isprint* () , *ispunct* () и *isspace* () .

islower ()

```
#include <ctype.h>
int islower (int ch);
```

Функцията *islower* () връща ненула ако *ch* е малка буква , иначе се връща 0 . Свързана функция е *isupper* () .

isprint ()

```
#include <ctype.h>
int isprint (int ch);
```

Функцията *isprint* () връща ненула ако *ch* е печатим символ , включително интервал , иначе се връща 0 . Печатимите символи са често в интервала от *0x20* до *0x7E* . Свързани функции са *isalnum* () , *isalpha* () , *isctrl* () , *isdigit* () , *isgraph* () , *ispunct* () и *isspace* () .

ispunct ()

```
#include <ctype.h>
int ispunct (int ch);
```

Функцията *ispunct* () връща ненула ако *ch* е пунктуационен символ , иначе се връща 0 . Термина пунктуационен , както е дефиниран чрез тази функция , включва всички печатими символи които не са букви , цифри или интервал . Свързани функции са *isalnum* () , *isalpha* () , *isctrl* () , *isdigit* () , *isgraph* () , *isprint* () и *isspace* () .

isspace ()

```
#include <ctype . h >
int isspace (int ch) ;
```

Функцията *isspace ()* връща ненула ако *ch* е или интервал , хоризонтална табулация , вертикална табулация , връщане на каретата или символ за нов ред , иначе се връща 0 . Свързани функции са *isalnum ()* , *isalpha ()* , *iscntrl ()* , *isdigit ()* , *isgraph ()* , *isprintt ()* и *ispunct ()* .

isupper ()

```
#include <ctype . h >
int isupper (int ch) ;
```

Функцията *isupper ()* връща ненула ако *ch* е главна буква , иначе се връща 0 . Свързана функция е *islower ()* .

isxdigit ()

```
#include <ctype . h >
int isxdigit (int ch) ;
```

Функцията *isxdigit ()* връща ненула ако *ch* е шестнайсетично цифра , иначе се връща 0 . Шестнайсетичната цифра ще бъде в един от тези интервали : *A – F* , *a – f* , или *0 – 9* Свързани функции са *isalnum ()* , *isalpha ()* , *iscntrl ()* , *isdigit ()* , *isgraph ()* , *ispunct ()* и *isspace ()* .

memchr ()

```
#include <string . h >
void *memchr (const void *buffer , int ch , size_t count) ;
```

Функцията *memchr ()* претърсва масива указван чрез *buffer* за първото срещане на *ch* в първите *count* символа . Функцията *memchr ()* връща указател към първото срещане на *ch* в *buffer* , или връща нулев указател ако *ch* не е намерен . Свързани функции са *memcpu ()* и *isspace ()* .

memcmp ()

```
#include <string . h >
void *memcmp (const void *buf1 , const void *buf2 ,
size_t count) ;
```

Функцията *memcmp ()* сравнява първите *count* символа от масивите указвани от *buf1* и *buf2* . Функцията *memcmp ()* връща цяло число което се интерпретира както е показано тук :

Стойност	значение
По – малка от 0	<i>buf1</i> е по– малък от <i>buf2</i>
0	<i>buf1</i> е равен на <i>buf2</i>
По – голяма от 0	<i>buf1</i> е по– голям от <i>buf2</i>

Свързани функции са *memchr ()* , *memcpu ()* и *strcmp ()* .

memcpy ()

```
#include <string . h >
void *memcpy (void *to , const void *from , size_t count) ;
```

Функцията *memcpy ()* копира *count* символа от масива указван чрез *from* в масив указван от *to* . Ако масивите излизат извън границите поведението на

memcpy() е неопределено. Функцията *memcpy()* връща *to*. Свързана функция е *memmove()*.

memmove()

#include <string.h>

void *memmove(void *to, const void *from, size_t count);

Функцията *memmove()* копира *count* символа от масив указван чрез *from* в масив указван от *to*. Ако масивите са по-големи копието ще се събере точно, поставяйки правилното съдържание в *to*, но оставяйки *from* изменен. Функцията *memmove()* връща *to*. Свързана функция е *memcpy()*.

memset()

#include <string.h>

void *memset(void *buf, int ch, size_t count);

Функцията *memset()* копира нискоразредния байт от *ch* в първите *count* символа на масива указван от *buf*. Тя връща *buf*. Най-общата употреба на *memset()* е да инициализира регион от паметта с някаква известна стойност. Свързани функции са *memcpy()*, *memcpy()* и *memmove()*.

strcat()

#include <string.h>

char *strcat(char *str1, const char *str2);

Функцията *strcat()* свързва копие от низа указван чрез *str2* към низа указван от *str1* и прекъсва *str1* със 0. Нулевия терминатор първоначално завършващ *str1* се презаписва от първия символ на *str2*. Низа *str2* е непроменен от операцията. Ако масивите се препълнят, поведението на *strcat()* е неопределено. Функцията *strcat()* връща *str1*. Запомнете че не се прави никаква проверка за границите, така че това е задължение на програмиста да осигури достатъчна големина на *str1* да съдържа неговото начално съдържание и това на *str2*. Свързани функции са *strchr()*, *strcmp()* и *strcpy()*.

strchr()

#include <string.h>

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

Функцията *strchr()* връща указател към първото срещане на нискоразредния байт на *ch* в низа указван от *str*. Ако не е открито съвпадение се връща нулев указател. Свързани функции са *strpbrk()*, *strspn()*, *strstr()* и *strtok()*.

strcmp()

#include <string.h>

int strcmp(const char *str1, const char *str2);

Функцията *strcmp()* лексикографски сравнява два низа връща цяло число базирано на изхода както е показано тук:

Стойност	значение
По-малка от 0	<i>str1</i> е по-малък от <i>str2</i>
0	<i>str1</i> е равен на <i>str2</i>
По-голяма от 0	<i>str1</i> е по-голям от <i>str2</i>

Свързани функции са *strchr()*, *strcmp()* и *strcpy()*.

strcoll()

```
#include <string.h>
```

```
int strcoll(const char *str1, const char *str2);
```

Функцията *strcoll()* сравнява низа указван от *str1* със този указван от *str2*. Сравнението се представя в съответствие със местните спецификации използвайки функцията *setlocale()* (Виж ”*setlocale()*“ за детайли). Функцията *strcoll()* връща **integer** който се интерпретира както е показано тук:

Стойност	значение
По – малка от 0	<i>str1</i> е по – малък от <i>str2</i>
0	<i>str1</i> е равен на <i>str2</i>
По – голяма от 0	<i>str1</i> е по – голям от <i>str2</i>

Свързани функции са *memcmp()* и *strcmp()*.

strcpy()

```
#include <string.h>
```

```
int strcpy(char *str1, const char *str2);
```

Функцията *strcpy()* копира съдържанието на низа указван от *str2* в масив указван от *str1*. *str1* трябва да бъде указател към нулево – терминиран низ. Функцията *strcpy()* връща *str1*. Ако *str1* и *str2* се надхвърлят, поведението на функцията *strcpy()* е неопределено. Свързани функции са *memcmp()*, *strchr()*, *strcmp()* и *strncmp()*.

strcspn()

```
#include <string.h>
```

```
size_t strcspn(const char *str1, const char *str2);
```

Функцията *strcspn()* връща дължината на начален подниз от низ указван от *str1* който съставен само от онези символи несъдържани от низа указван от *str2*. Или *strcspn()* връща индекс на първия символ в низа указван от *str1* който отговаря на някой от символите в низа указван от *str2*. Свързани функции са *strrchr()*, *strpbrk()*, *strstr()* и *strtok()*.

strerror()

```
#include <string.h>
```

```
char *strerror(int errnum);
```

Функцията *strerror()* връща указател към имплементационно – зависим низ свързан със стойността на *errnum*. В никакъв случай не трябва да изменяте низа.

strlen()

```
#include <string.h>
```

```
size_t strlen(char *str);
```

Функцията *strlen()* връща дължината на нулево – терминирания низ указван от *str*. Нулата не се брои. Свързани функции са *memcmp()*, *strchr()*, *strcmp()* и *strncmp()*.

strncat ()

```
#include <string . h >
char *strncat ( const char *str1 , const char *str2 , size_t
count ) ;
```

Функцията *strncat ()* свързва *count* символа от низа указван от *str2* към низа *str1* и прекъсва *str1* с 0 . Нулевия терминатор първоначално прекъсващ *str1* се презаписва от първия символ на *str2* . Низа *str2* е непроменен от операцията . Ако низовете се препълнят поведението на функцията е неопределено . Функцията *strncat ()* връща *str1* . Запомнете че няма проверка на границите , така че е отговорност на програмиста да осигури достатъчна големина на *str1* така че да съдържа и първоначалното си съдържание , и това от *str2* . Свързани функции са *strcat ()* , *strchr ()* , *strncmp ()* и *strncpy ()* .

strncmp ()

```
#include <string . h >
int strncmp ( const char *str1 , const char *str2 , size_t count
) ;
```

Функцията *strncmp ()* лексикографски сравнява *count* символа от два нулево – терминирани низа и връща **integer** съгласно таблицата :

Стойност	значение
По – малка от 0	<i>str1</i> е по – малък от <i>str2</i>
0	<i>str1</i> е равен на <i>str2</i>
По – голяма от 0	<i>str1</i> е по – голям от <i>str2</i>

Ако има по – малко от *count* символа в единия низ , сравнението свършва когато се срещне първата нула . Свързани функции са *strcmp ()* , *strchr ()* и *strncpy ()* .

strncpy ()

```
#include <string . h >
char *strncpy ( char *str1 , const char *str2 , size_t count ) ;
```

Функцията *strncpy ()* се използва да копира *count* символа от низа указван чрез *str2* в низа указван от *str1* . *str2* трябва да бъде указател към нулево – терминиран низ . Ако *str1* и *str2* се препълнят , поведението на *strncpy ()* е неопределено . Ако низа указван от *str2* има по – малко символа от *count* , ще бъдат добавени нули в края на *str1* . Съответно ако низа указван от *str2* е по – дълъг от *count* символа , резултатният низ указван от *str1* няма да бъде нулево прекъснат . Функцията *strncpy ()* връща *str1* . Свързани функции са *memcpy ()* , *strchr ()* , *strncat ()* и *strncmp ()* .

strpbrk ()

```
#include <string . h >
char *strpbrk ( const char *str1 , const char *str2 ) ;
```

Функцията *strpbrk ()* връща указател към първия символ в низа указван от *str1* който съвпада със някой символ в низа указван от *str2* . Нулевите терминатори не се включват . Ако не се открият съвпадения се връща нулев указател . Свързани функции са *strspn ()* , *strchr ()* , *strstr ()* и *strtok ()* .

strrchr ()

```
#include <string . h >
char *strrchr ( const char *str , int ch ) ;
```

Функцията *strrchr ()* връща указател към последното срещане на ниско – разрядния байт на *ch* в низа указван от *str* . Ако няма открито съвпадение се връща нулев указател . Свързани функции са *strpbrk ()* , *strspn ()* , *strstr ()* и *strtok ()* .

strspn ()

```
#include <string . h >
size_t strspn ( const char *str1 , const char *str2 ) ;
```

Функцията *strspn ()* връща дължината на началния подниз на низа указван от *str1* който е съставен от тези символи съдържащи в низа указван от *str2* . Или *strspn ()* връща индекса на първия символ в низа указван от *str1* който не съвпада със никой от символите в низа указван от *str2* . Свързани функции са *strpbrk ()* , *strrchr ()* , *strstr ()* и *strtok ()* .

strstr ()

```
#include <string . h >
char *strstr ( const char *str1 , const char *str2 ) ;
```

Функцията *strstr ()* връща указател към първото срещане в низа указван от *str1* на низа указван от *str2* . Тя връща нулев указател ако не е открито съвпадение . Свързани функции са *strchr ()* , *strcspn ()* , *strpbrk ()* , *strspn ()* , *strtok ()* и *strrchr ()* .

strtok ()

```
#include <string . h >
char *strtok ( char *str1 , const char *str2 ) ;
```

Функцията *strtok ()* връща указател към следващия знак в низа указван от *str1* . Символите съставлящи низа *str2* са разграничителите които определят знака . Нулев указател се връща когато няма знак за връщане . За да разделиме низ първото извикване на *strtok ()* трябва да има *str1* указател към низа който ще се разделя . Следващото извикване трябва да използва нулев указател за *str1* . По този начин целия низ може да бъде разделен на отделни знаци . Възможно е да използвате различно множество от разграничители за всяко извикване на *strtok ()* . Свързани функции са *strchr ()* , *strcspn ()* , *strpbrk ()* , *strrchr ()* и *strspn ()* .

Програмен съвет

Функцията *strtok ()* дава начин чрез който можете да разделите низ на неговите съставни части . Например , следната програма разделя низа "One , two , and three " :

```
#include <stdio . h >
#include <string . h >
int main ( void )
{
    char *p ;
    p = strtok ( " One , two , and three . " , " , " , " ) ;
```

```

printf ( p ) ;
do {
    p = strtok ( NULL , “ , . “ ) ;
    if ( p ) printf ( “ | %s “ , p ) ;
} while ( p ) ;
return 0 ;
}

```

Изхода на тази програма е :

One|two|three

Забележете как **strtok ()** се извиква първо със низа който ще бъде разделян , но следващото извикване използва **NULL** за първи аргумент . Функцията **strtok ()** поддържа указател към низа който е бил разделен . Когато първия аргумент на **strtok ()** указва низ , този вътрешен указател се настройва в началото на този низ . Когато първия аргумент е **NULL** , **strtok ()** продължава разделянето на предишния низ от там където е спрял , увеличавайки вътрешния указател със всеки знак . Така **strtok ()** може да раздели целия низ . Също забележете как низа който определя разграничителите е променен между първото и следващите извиквания . Със всяко извикване може да се определят различни разграничители

strxfrm ()

```

#include < string . h >
size_t strxfrm ( char *str1 , const char *str2 , size_t count ) ;

```

Функцията **strxfrm ()** преобразува първите **count** символа от низа указван от **str2** така че той да може да бъде използван чрез функцията **strcmp ()** и слага резултата в низ указван от **str1** . След преобразуването резултата от **strcmp ()** със **str1** и **strcoll ()** , използвайки началния низ на **str2** ще бъде еднакъв . Функцията **strxfrm ()** връща дължината на преобразувания низ . Свързана функция е **strcoll ()**

tolower ()

```

#include < ctype . h >
int tolower ( int ch ) ;

```

Функцията **tolower ()** връща еквивалентната на **ch** малка буква , ако **ch** е буква , иначе **ch** се връща непроменена . Свързана функция е **toupper ()** .

toupper ()

```

#include < ctype . h >
int toupper ( int ch ) ;

```

Функцията **toupper ()** връща еквивалентната на **ch** главна буква , ако **ch** е буква , иначе **ch** се връща непроменена . Свързана функция е **tolower ()** .

ГЛАВА 8 – Математични функции в C

Стандартната библиотека на C съдържа различни математични функции които попадат в следните категории :

- Тригонометрични функции

- Хиперболични функции
- Експоненциални и логаритмични функции
- Разнообразни функции

Всички математични функции изискват хедърния файл **MATH.H**. В добавка към декларираните математични функции този хедър дефинира и три макроса наречени **EDOM**, **ERANGE** и **HUGE_VAL**. Ако аргумента на математична функция не е в областта за която е дефиниран се връща имплементационно – зависима стойност и вградената глобална **integer** променлива **errno** се установява на **EDOM**. Ако функцията изчисли резултат който е прекалено голям за да бъде побран от **double** се получава препълване. Това предизвиква функцията да върне **HUGE_VAL** и **errno** се установява на **ERANGE** показвайки грешка в областта. Ако стане преминаване под долната граница, функцията връща 0 и установява **errno** на **ERANGE**. Всички ъгли са в радиани.

acos ()

```
#include <math.h>
double acos ( double arg );
```

Функцията **acos ()** връща аркускосинус от **arg**. Аргумента на **acos ()** трябва да бъде в интервала от -1 до 1, иначе се получава грешка в областта. Свързани функции са **asin ()**, **atan ()**, **atan2 ()**, **sin ()**, **cos ()**, **tan ()**, **sinh ()**, **cosh ()** и **tanh ()**.

asin ()

```
#include <math.h>
double asin ( double arg );
```

Функцията **asin ()** връща аркуссинус от **arg**. Аргумента на **asin ()** трябва да бъде в интервала от -1 до 1, иначе се получава грешка в областта. Свързани функции са **acos ()**, **atan ()**, **atan2 ()**, **sin ()**, **cos ()**, **tan ()**, **sinh ()**, **cosh ()** и **tanh ()**.

atan ()

```
#include <math.h>
double atan ( double arg );
```

Функцията **atan ()** връща аркустангенс от **arg**. Свързани функции са **asin ()**, **acos ()**, **atan2 ()**, **sin ()**, **cos ()**, **tan ()**, **sinh ()**, **cosh ()** и **tanh ()**.

atan2 ()

```
#include <math.h>
double atan2 ( double y, double x );
```

Функцията **atan2 ()** връща аркустангенс от y/x . Тя използва знаците на аргументите си за да изчисли квадранта на връщаната стойност. Свързани функции са **asin ()**, **acos ()**, **atan ()**, **sin ()**, **cos ()**, **tan ()**, **sinh ()**, **cosh ()** и **tanh ()**.

ceil ()

```
#include <math.h>
double ceil ( double num );
```

Функцията *ceil()* връща най – малкото цяло число (представено като *double*) не по – малко от *num*. Например подаваме *1.02 ceil()* ще върне *2.0*. Подаваме *-1.02 ceil()* ще върне *-1*. Свързани функции са *floor()* и *fmod()*.

cos ()

```
#include <math.h>
double cos ( double arg );
```

Функцията *cos()* връща косинуса от *arg*. Стойността на *arg* трябва да бъде в радиани. Свързани функции са *asin()*, *acos()*, *atan()*, *atan2()*, *tan()*, *sin()*, *sinh()*, *cosh()* и *tanh()*.

cosh ()

```
#include <math.h>
double cosh ( double arg );
```

Функцията *cosh()* връща хиперболичен косинус от *arg*. Свързани функции са *asin()*, *acos()*, *atan()*, *atan2()*, *tan()*, *sin()*, *sinh()*, *cos()* и *tanh()*.

exp ()

```
#include <math.h>
double exp ( double arg );
```

Функцията *exp()* връща естествен логаритъм е повдигнат на степен *arg*. Свързана функция е *log()*.

fabs ()

```
#include <math.h>
double fabs ( double num );
```

Функцията *fabs()* връща абсолютната стойност на *num*. Свързана функция е *abs()*.

floor ()

```
#include <math.h>
double floor ( double num );
```

Функцията *floor()* връща най – голямото число (представено като *double*) не по – голямо от *num*. Например подаваме *1.02 floor()* ще върне *1.0*. Подаваме *-1.02 floor()* ще върне *-2.0*. Свързани функции са *ceil()* и *fmod()*.

fmod ()

```
#include <math.h>
double fmod ( double y, double x );
```

Функцията *fmod()* връща остатък от *x/y*. Свързани функции са *ceil()*, *floor()* и *fabs()*.

frexp ()

```
#include <math.h>
double frexp ( double num, int *exp );
```

Функцията *frexp()* разлага числото *num* на мантия в интервала от *0.5* до *1* и цяла степен така че *num = mantissa * 2^{exp}*. Мантията се връща от функцията, а експонентата се съхранява в променлива указвана от *exp*. Свързана функция е *ldexp()*.

ldexp ()

```
#include <math.h>
double ldexp ( double num , int *exp );
```

Функцията *ldexp ()* връща стойността на **num** * 2^{exp}. Ако стане препълване се връща **HUGE_VAL**. Свързани функции са *frexp ()* и *modf ()*.

log ()

```
#include <math.h>
double log ( double num );
```

Функцията *log ()* връща естествен логаритъм от **num**. Грешка в областта се получава ако **num** е отрицателно, а грешка в интервала става ако аргумента е 0. Свързана функция е *log10 ()*.

log10 ()

```
#include <math.h>
double log10 ( double num );
```

Функцията *log10 ()* връща десетичен логаритъм от **num**. Грешка в областта се получава ако **num** е отрицателно, а грешка в интервала става ако аргумента е 0. Свързана функция е *log ()*.

modf ()

```
#include <math.h>
double modf ( double num , double *i );
```

Функцията *modf ()* разлага **num** на цяла и дробна част. Тя връща дробната част и слага цялата част в променливата указвана от **i**. Свързани функции са *frexp ()* и *ldexp ()*.

pow ()

```
#include <math.h>
double pow ( double base , double exp );
```

Функцията *pow ()* връща **base** повдигната на степен **exp** (**base**^{exp}). Грешка в областта може да стане ако **base** е 0 и **exp** е по-малко или равно на 0. Същото ще се получи ако **base** е отрицателно и **exp** не е цяло. Препълване предизвиква грешка в интервала. Свързани функции са *exp ()*, *log ()* и *sqrt ()*.

sin ()

```
#include <math.h>
double sin ( double arg );
```

Функцията *sin ()* връща синуса от **arg**. Стойността на **arg** трябва да бъде в радиани. Свързани функции са *asin ()*, *acos ()*, *atan ()*, *atan2 ()*, *tan ()*, *cos ()*, *sinh ()*, *cosh ()* и *tanh ()*.

sinh ()

```
#include <math.h>
double sinh ( double arg );
```

Функцията *sinh ()* връща хиперболичен синус от **arg**. Свързани функции са *asin ()*, *acos ()*, *atan ()*, *atan2 ()*, *tan ()*, *cosh ()*, *sin ()*, *cos ()* и *tanh ()*.

sqrt ()

```
#include <math.h>
double sqrt ( double num );
```

Функцията *sqrt ()* връща корен квадратен от *num*. Ако се извика с отрицателен аргумент се получава грешка в областта. Свързани функции са *exp ()*, *log ()* и *pow ()*.

tan ()

```
#include <math.h>
double tan ( double arg );
```

Функцията *tan ()* връща тангенс от *arg*. Стойността на *arg* трябва да бъде в радиани. Свързани функции са *asin ()*, *acos ()*, *atan ()*, *atan2 ()*, *cos ()*, *sin ()*, *sinh ()*, *cosh ()* и *tanh ()*.

tanh ()

```
#include <math.h>
double tanh ( double arg );
```

Функцията *tanh ()* връща хиперболически тангенс от *arg*. Свързани функции са *asin ()*, *acos ()*, *atan ()*, *atan2 ()*, *sin ()*, *cos ()*, *sinh ()*, *cosh ()* и *tan ()*.

ГЛАВА 9 – Функции в С за време , дата и локализационни функции

Тази секция обхваща функциите за време , дата и тези функции които се отнасят за географското разположение на компютъра . С дефинира различни функции които работят със датата и времето на системата като изминалото време . Тези функции изискват хедърния файл **TIME . H** . Този хедър дефинира три свързани със времето типа : *clock_t* , *time_t* и *tm* . Типовете *clock_t* и *time_t* способни да представят системния час и дата като някакво цяло число . Това се нарича календарно време . Структурата тип *tm* съдържа датата и времето разделени на елементи . Структурата *tm* е дефинирана така :

```
struct tm {
    int tm_sec ; /* секунди , 0– 61 */
    int tm_min ; /* минути , 0– 59 */
    int tm_hour ; /* часове , 0– 23 */
    int tm_mday ; /* ден от месеца , 1– 31 */
    int tm_mon ; /* месец от януари , 0– 11 */
    int tm_year ; /* години от 1900 */
    int tm_wday ; /* дни от неделя , 0– 6 */
    int tm_yday ; /* дни от 1 януари , 0– 365 */
    int tm_isdst ; /* индикатор за лятно време */
}
```

Стойността на *tm_isdst* ще бъде положителна ако е в действие лятното часово време , 0 ако не е , и отрицателна ако няма информация . Тази форма на датата и времето е наречена разделено време . В добавка **TIME . H** дефинира макроса **CLOCKS_PER_SEC** , който е броя на тактовете на системния

часовник за секунда. Функциите за географска локализация изискват хедърния файл **LOCALE.H**. Повечето C/C++ компилатори също добавят допълнителни функции за час и дата, така че проверете ръководството на компилатора за повече информация за такива типове функции.

asctime ()

#include <time.h>

char *asctime (const struct tm *ptr);

Функцията **asctime ()** връща указател към низ който конвертира информацията съхранявана в структурата указвана от **ptr** в следната форма:

ден месец дата час : минути : секунди година \n \0

Това е пример:

Wed Jun 19 12 : 05 : 34 1999

Указателя към структура подаден на **asctime ()** е първоначално получен от **localtime ()** или **gmtime ()**. Буфера използван от **asctime ()** да съхрани форматирания изходен низ е постоянно заделен низ от символи и се презаписва всеки път когато се извика функцията. Ако желаете да запазите съдържанието на низа ще трябва да го копирате някъде. Свързани функции са **localtime ()**, **gmtime ()**, **time ()** и **ctime ()**.

clock ()

#include <time.h>

clock_t clock (void);

Функцията **clock ()** връща стойност която е равна на количеството време от пускането на извикващата програма. За да трансформирате тази стойност в секунди, разделете я на **CLOCKS_PER_SEC**. Стойност **-1** се връща ако времето не е достъпно. Свързани функции са **time ()**, **asctime ()** и **ctime ()**.

ctime ()

#include <time.h>

char *ctime (const time_t *time);

Функцията **ctime ()** връща указател към низ със следния формат:

ден месец дата час : минути : секунди година \n \0

даващ указател към календарно време. Календарното време е първоначално получено чрез извикване **time ()**. Буфера използван от **ctime ()** да съхрани форматирания изходен низ е постоянно заделен низ от символи и се презаписва всеки път когато се извика функцията. Ако желаете да запазите съдържанието на низа ще трябва да го копирате някъде. Свързани функции са **localtime ()**, **gmtime ()**, **time ()** и **asctime ()**.

difftime ()

#include <time.h>

double difftime (time_t time2, time_t time1);

Функцията **difftime ()** връща разликата в секунди между **time1** и **time2**. Тоест тя връща разликата **time2 - time1**. Свързани функции са **localtime ()**, **gmtime ()**, **time ()** и **asctime ()**.

gmtime ()

```
#include <time.h>
```

```
struct tm *gmtime (const time_t *time);
```

Функцията **gmtime()** връща указател към разделената форма на **time** във формат на **tm** структура. Времето е във формат Световно координатно време (**UTC**) което всъщност е време по Гринуич. Стойността **time** е първоначално получена чрез извикване на **time()**. Ако системата не поддържа **UTC** се връща **NULL**. Структурата използвана от **gmtime()** да съхранява разделеното време е постоянно заделена и се презаписва всеки път когато се извика функцията. Ако желаете да запазите съдържанието на структурата ще трябва да я копирате някъде. Свързани функции са **localtime()**, **time()** и **asctime()**.

localeconv ()

```
#include <time.h>
```

```
struct lconv *localeconv ( void ) ;
```

Функцията **localeconv()** връща указател към структура от тип **lconv**, която съдържа различна информация за специфичностите на страната отнасящи се до начина на форматиране на числата. Структурата **lconv** е организирана така:

```

1
struct lconv {

```

```
char *decimal_point; /*десетична точка за
немонетарни стойности*/
```

```
char *thousands_sep ; /* разделител за хилядните
                        за немонетарни стойности */
```

```
char *grouping ; /* определя групирането за
                  немонетарни стойности */
```

```
char int_curr_symbol; /* международен символ
за валута */
```

```
char *currency symbol; /* местен символ за валута */
```

```
char *mon_decimal_point; /* символ за десетична точка
                           за монетарни стойности */
```

*char *mon_thousands_sep; /* разделител за хилядните
за монетарни стойности */*

```
char *mon_grouping; /* определя групирането за
                    монетарни стойности */
```

```
char *positive_sign ;      /* знак за положителна стойност
                             за монетарни стойности */
```

```
char *negative_sgn; /* знак за отрицателна стойност
за монетарни стойности */
```

<i>char int_frac_digits ;</i>	/* броя на цифрите показани от дясната страна на десетичната точка за монетарни стойности показан чрез
-------------------------------	---

международен формат */

```
char frac_digits; /* броя на цифрите показани от
дясната страна на десетичната
точка за монетарни стойности
```

```

char p_cs_precedes ; /* 1 ако символа за валута
показан чрез местен формат */
/* 0 ако символа за валута
предхожда положителната
стойност ; 0 ако символа за
за валута е след стойността */
char p_sep_by_space ; /* 1 ако символа за валута е
разделен от стойността чрез
интервал ; 0 в противен случай */
char n_cs_precedes ; /* 1 ако символа за валута
предхожда отрицателната
стойност ; 0 ако символа за
за валута е след стойността */

char n_sep_by_space ; /* 1 ако символа за валута е
разделен от отрицателна
стойност чрез интервал ; 0 в
противен случай */
char p_sign_posn ; /* показва позицията на символа
положителна стойност */
char n_sign_posn ; /* показва позицията на символа
отрицателна стойност */
}

```

Функцията *lconv()* връща указател към *lconv* структура. Вие не трябва да променяте съдържанието на тази структура. Вижте документацията на компилатора за точна информация относно се до *lconv* структурата. Свързана функция е *setlocale()*.

localtime()

```

#include <time.h>
struct tm *localtime(const time_t *time);

```

Функцията *localtime()* връща указател към разделената форма на *time* във форма на *tm* структура. Времето се представя в местно време. Стойността на *time* първоначално е получена чрез извикване на *time()*. Структурата използвана от *localtime()* за съхранение на разделеното време е постоянно заделена и се презаписва всеки път когато се извика функцията. Ако желаете да запазите съдържанието на структурата, ще трябва да го запишете. Свързани функции са *gmtime()*, *time()* и *asctime()*.

mktime()

```

#include <time.h>
time_t mktime(struct tm *time);

```

Функцията *mktime()* връща календарния еквивалент на разделеното време получено от структурата върната от *time()*. Елементите *tm_wday* и *tm_yday* се установяват чрез функцията така че те не се нуждаят от дефиниране по време на извикване. Ако *mktime()* не представя информацията точно календарно време се връща -1. Свързани функции са *time()*, *gmtime()*, *asctime()* и *ctime()*.

Програмен съвет

Функцията *mktime()* е особено полезна когато искате да знаете кой ден от седмицата на коя дата се пада. Например, какъв ден от седмицата е **12 януари, 2012 г.** За да откриете това извикайте *mktime()* с тази дата и разгледайте члена *tm_wday* от структурата *tm* след връщането на функцията. Той ще съдържа деня от седмицата. Следващата програма демонстрира този метод:

```
/* открива кой ден от седмицата е 12 януари 2012 г */
#include <stdio.h>
#include <time.h>
char day[ ][20] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday", "Saturday"
};
int main ( void )
{
    struct tm t;
    t.tm_mday = 12;
    t.tm_mon = 0;
    t.tm_year = 112;
    t.tm_hour = 0;
    t.tm_min = 0;
    t.tm_sec = 0;
    t.tm_isdst = 0;
    mktime ( &t ); /* попълва деня от седмицата*/
    printf ( "Day of week is %s.\n", day[ t.tm_wday ] );
    return 0;
}
```

Когато програмата се пусне *mktime()* автоматично изчислява деня от седмицата който е четвъртък в този случай. Понеже връщаната стойност на *mktime()* не е нужна тя просто се игнорира.

setlocale ()

```
#include <locale.h>
char *setlocale ( int type, const char *locale );
```

Функцията *setlocale()* дава известни параметри които са чувствителни към географското положение на изпълнението на програмата за да бъдат проверени или установени. Например в Европа запетайката се използва на мястото на десетичната точка. Ако *locale* е **0**, *setlocale()* връща указател към текущия локализационен низ. В противен случай, *setlocale()* опитва да използва определения локализационен низ за да установи местните параметри както определя *type*. По време на извикване *type* трябва да бъде един от следните макроси:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
```

LC_NUMERIC
LC_TIME

LC_ALL се отнася за всички локализационни категории. LC_COLLATE действа върху функцията *strcoll()*. LC_MONETARY определя валутния формат. LC_NUMERIC променя символа за десетична точка за форматираните I/O функции. Накрая LC_TIME определя поведението на функцията

strftime(). Стандарта ANSI C определя два възможни низа за *locale*. Първия е “C” който определя минималната среда за C компилация. Втория е “” нулев низ, който определя подразбиращата се зависеща от изпълнението среда. Всички други стойности за *locale* са имплементационно – зависими и ще засегнат преносимостта. Функцията *setlocale()* връща указател към низ свързан със параметъра *type*. Свързани функции са *localeconv()*, *time()*, *strcoll()* и *strftime()*

strftime ()

```
#include <time.h>
size_t strftime ( char *str, size_t maxsize, const char *fmt,
const struct tm *time );
```

Функцията *strftime()* поставя информацията за дата и време покрай друга информация в низ указван от *str* съгласно форматните команди открити в низа указван чрез *fmt* и използвайки разделеното време *time*. Максимум *maxsize* символа ще бъдат поставени в *str*. Функцията *strftime()* работи също като *sprintf()* приемайки множество от форматни команди които започват поставяйки със знак процент (%) и поставяйки форматния изход в низ. Форматните команди се използват да определят точния начин различна информация за времето и датата да се представят в *str*. Всички други символи открити във форматния низ се поставят непроменени в *str*. Показаните време и дата са в местно време. Форматните команди са изброени в следната таблица. Забележете че много от командите са в зависимост от случая :

команда	замества
%a	Съкратеното име на ден от седмицата
%A	Пълно име на ден от седмицата
%b	Съкратено име на месеца
%B	Пълно име на месеца
%c	Стандартен низ за дата и време
%d	Деня от месеца като десетично число (1– 31)
%H	Час (0– 23)
%I	Час (1– 12)
%j	Деня от годината като десетично число (1– 366)
%m	Месеца като десетично число (1– 12)

%M	Минута като десетично число (1– 59)
%p	Местния еквивалент на AM и PM
%S	Секунда като десетично число (1– 61)
%U	Седмица от годината като неделя е първи ден (0 – 53)
%w	Ден от седмицата като десетично число (0– 6 неделя е 0)
%W	Седмица от годината понеделник е първи ден (0– 53)
%x	Стандартен низ за дата
%X	Стандартен низ за време
%y	Годината в десетичен формат без века (0– 99)
%Y	Годината включително века в десетичен формат
%Z	Името на часовата зона
%%	Знак за процент (%)

Функцията *strftime* () връща броя на символите поставени в низа указван от *str* или 0 ако стане грешка . Свързани функции са *time* () , *localtime* () и *gmtime* () .

time ()

#include <time . h >

time_t *time* (*time_t* **time*) ;

Функцията *time* () връща текущото календарно време на системата . Ако системата няма настроено време се връща -1 . Функцията *time* () може да бъде извикана или с нулев указател или с указател към променлива от тип *time_t* . Ако се използва последното , на променливата ще бъде присвоено календарното време . Свързани функции са *localtime* () , *gmtime* () , *strftime* () и *ctime* () .

ГЛАВА 10 – Функции в C за динамично заделяне

Тази секция описва системата на C за динамично заделяне . В основата и са функциите *malloc* () и *free* () . Всеки път когато се извиква *malloc* () част от оставащата свободна памет се заделя . Всеки път когато се извика *free* () паметта се освобождава . Региона от свободна памет от който се заделя паметта се нарича хийп . Прототипите на функциите за динамично заделяне са в **STDLIB.H** . Всички C/C++ компилатори ще включват най – малко тези четири функции за динамично заделяне : *calloc* () , *malloc* () , *free* () и *realloc* ()

. Обаче вашият компилатор почти сигурно съдържа допълнителни варианти на тези функции за да съгласува различни опции и разлики на средата . Ще трябва да проверите документацията на компилатора си . Въпреки че C++ поддържа функциите за динамично заделяне описани тук , не трябва да ги използвате във C++ програми . Причината за това е че C++ доставя специалните оператори за динамично заделяне наречени **new** и **delete** . Има няколко предимства от използването на операторите в C++ за динамично заделяне . Първо **new** автоматично заделя достатъчното количество памет за типа данни за заделяне . Второ , връща правилния тип указател към тази памет . Трето и **new** и **delete** могат да бъдат предефинирани . Откакто **new** и **delete** имат предимство над функциите в C за динамично заделяне , тяхната употреба е препоръчителна за C++ програми .(за разглеждане на **new** и **delete** , виж Глава 5)

calloc ()

```
#include <stdlib . h >
```

```
void *calloc ( size_t num , size_t size ) ;
```

Функцията **calloc ()** заделя памет големината на която е равна **num * size** . Така **calloc ()** заделя достатъчно памет за масив от **num** обекта с размер **size** . Функцията **calloc ()** връща указател към първия байт на заделения регион . Ако няма достатъчно памет за изпълнение на заявката се връща нулев указател . Винаги е важно да проверите че върнатата стойност не е нулев указател преди да опитате да я използвате . Свързани функции са **free ()** , **malloc ()** и **realloc ()** .

free ()

```
#include <stdlib . h >
```

```
void free ( void *ptr ) ;
```

Функцията **free ()** освобождава паметта указвана от **ptr** в хийпа . Това прави паметта свободна за бъдещо заделяне . Наложително е **free ()** да бъде извикана с указател който е бил преди това заделен с една от системните функции за динамично заделяне (една от **malloc ()** или **realloc ()**) . Използването на невалиден указател при извикване най – вероятно ще разруши механизма за управление на паметта и ще причини срив на системата . Свързани функции са **calloc ()** , **malloc ()** и **realloc ()** .

malloc ()

```
#include <stdlib . h >
```

```
void *malloc ( size_t size ) ;
```

Функцията **malloc ()** връща указател към първия байт от регион в паметта с размер **size** който е бил заделен от хийпа . Ако има недостатъчно памет в хийпа за изпълнение на заявката **malloc ()** връща нулев указател . Винаги е важно да проверите че върнатата стойност не е нулев указател преди да опитате да я използвате . Опитът за използване на нулев указател обикновено ще доведе до срив на системата . Свързани функции са **free ()** , **realloc ()** и **calloc ()** .

Програмен съвет

Ако пишете **16** – битови програми за фамилията процесори **8086** (такива като **80486** или **Pentium**), тогава вашия компилатор ще доставя допълнителни функции за заделяне които съгласуват сегментния модел на паметта използван от тези процесори когато работят в **16** – битов режим . Например , това ще са функции които заделят памет от **FAR** хийпа (хийпа който е извън подразбирация се сегмент за данни) които могат да заделят указатели към паметта която е по – голяма от един сегмент и също да освобождават такава памет .

realloc ()

#include <stdlib . h >

void *realloc (void *ptr , size_t size) ;

Функцията **realloc ()** променя размера на преди това заделената памет указвана от **ptr** към такава определена от **size** . Стойността на **size** може да бъде по – голяма или по – малка от първоначалната . Връща се указател към блока памет понеже може да е необходимо за **realloc ()** да премести блока по ред на нарастване на неговия размер . Ако това стане , съдържанието на стария блок се копира в новия блок – няма загуба на информация . Ако **ptr** е **0** , **realloc ()** просто заделя **size** байта от памет и връща указател към тях . Ако **size** е **0** , паметта указвана от **ptr** е освободена . Ако няма достатъчно свободна памет в хийпа за заделянето на **size** байта , се връща нулев указател и първоначалния блок памет остава непроменен . Свързани функции са **free ()** , **malloc ()** и **calloc ()**

ГЛАВА 11 – Разнообразни функции на C

Функциите разглеждани в тази глава не могат лесно да се сложат в никоя друга категория . Те включват различни преобразувания , обработка на аргументи с различна дължина , сортиране и търсене и генериране на случайно число . Много от функциите обхванати тук изискват употребата на хедъра **STDLIB.H** . В този хедър са дефинирани два типа , **div_t** и **ldiv_t** които са типове стойности връщани съответно от **div ()** и **ldiv ()** . Също се дефинира и типа **size_t** които е беззнакова стойност връщана от **sizeof** . Тези макроси също са дефинирани :

Макрос	значение
NULL	Нулев указател
RAND_MAX	Максималната стойност която може да бъде върната от функцията rand ()
EXIT_FAILURE	Стойността върната на извиквания процес ако прекъсването на програмата е неуспешно
EXIT_SUCCESS	Стойността върната на извиквания процес ако

прекъсването на програмата
е успешно

Ако функцията изисква различен хедърен файл от **STDLIB.H** тогава при описанието на функцията ще го обсъдиме .

abort ()

```
#include <stdlib . h >
void abort ( void ) ;
```

Функцията **abort ()** предизвиква незабавно ненормално прекъсване на програмата . Обикновено файлове не се записват . В среди които поддържат това , **abort ()** ще върне имплементационно – зависима стойност на извикващия процес (обикновено операционната система) показваща грешка . Свързани функции са **exit ()** и **atexit ()** .

abs ()

```
#include <stdlib . h >
int abs ( int num ) ;
```

Функцията **abs ()** връща абсолютната стойност на цялото число **num** . Свързана функция е **labs ()** .

assert ()

```
#include <assert . h >
void assert ( int exp ) ;
```

Макроса **assert ()** дефиниран в хедъра **ASSERT.H** пише информация за грешка в **stderr** и след това прекъсва изпълнението на програмата ако израз **exp** се изчислява на **0** . В противен случай **assert ()** не прави нищо . Макар че точния изход е имплементационно – зависим , много компилатори използват съобщение подобно на това :

```
Assertion failed : < expression > , file <file > , line <linenum >
( Грешно твърдение : < израз > , файл <файл > , ред < номер
на ред >
```

Макроса **assert ()** е обикновено използван за да ви помогне да определите дали програмата действа правилно , с израз който е създаден по такъв начин че да се изчислява на true само когато няма възникнали грешки . Не е необходимо да премахвате изразите **assert ()** от сорс кода когато програмата е дебъгната понеже ако макроса **NDEBUG** е дефиниран (като нещо) макросите **assert ()** ще бъдат игнорирани . Свързана функция е **abort ()** .

atexit ()

```
#include <stdlib . h >
int atexit ( void ( *func ) ( void ) ) ;
```

Функцията **atexit ()** предизвиква функцията указвана от **func** да бъде извикана за нормално прекъсване на програмата . Функцията **atexit ()** връща **0** , ако функцията е успешно регистрирана като прекъсваща функция и **0** в противен случай . Най – малко **32** функции могат да бъдат установени и те ще бъдат извикани в обратен ред на тяхното установяване . Свързани функции са **exit ()** и **abort ()** .

atof ()

```
#include <stdlib.h>
double atof (const char *str);
```

Функцията **atof ()** преобразува низа указван от **str** в **double** стойност. Низа трябва да съдържа валидно число с плаваща точка. Ако това не е така, връщаната стойност е неопределена. Числото може да бъде прекъснато от всеки символ който не може да бъде част от валидно число с плаваща точка. Това включва празните символи, пунктуациите (различни от точки) и символите освен "Е" или "е". Това значи че ако **atof ()** е извикана със "100.00HELLO" ще бъде върната стойността **100.00**. Свързани функции са **atoi ()** и **atol ()**.

atoi ()

```
#include <stdlib.h>
int atoi (const char *str);
```

Функцията **atoi ()** преобразува низа указван от **str** в **int** стойност. Низа трябва да съдържа валидно цяло число. Ако това не е така, връщаната стойност е неопределена, обаче повечето изпълнения ще върнат **0**. Числото може да бъде прекъснато от всеки символ който не може да бъде част от цяло число. Това включва празните символи, пунктуациите и символите. Това значи че ако **atoi ()** е извикана със "123.23" ще бъде върната стойността **int 123**, а ".23" ще бъде отхвърлена. Свързани функции са **atof ()** и **atol ()**.

atol ()

```
#include <stdlib.h>
long atol (const char *str);
```

Функцията **atol ()** преобразува низа указван от **str** в **long** стойност. Низа трябва да съдържа валидно дълго цяло число. Ако това не е така, връщаната стойност е неопределена, обаче повечето изпълнения ще върнат **0**. Числото може да бъде прекъснато от всеки символ който не може да бъде част от цяло число. Това включва празните символи, пунктуациите и символите. Това значи че ако **atol ()** е извикана със "123.23" ще бъде върната стойността **long int 123L**, а ".23" ще бъде отхвърлена. Свързани функции са **atof ()** и **atoi ()**.

bsearch ()

```
#include <stdlib.h>
void *bsearch (const void *key, const void *buf,
size_t num, size_t size, int (*compare) (const void *, const void *));
```

Функцията **bsearch ()** изпълнява двоично търсене в сортирания масив указван от **buf** и връща указател към първия член който съвпада със ключа указван от **key**. Броя на елементите в масива е определен от **num**, а размера (в байтове) на всеки елемент е **size**. Функцията указвана чрез **compare** се използва да сравнява елемент от масива с ключа. Формата на **compare** функцията трябва да бъде както следва:

```
int func_name (const void *arg1, const void arg2);
```

Тя трябва да връща стойност както е описано тук :

Сравнение	Върната стойност
<i>arg1</i> е по – малко от <i>arg2</i>	по – малка от 0
<i>arg1</i> е равна на <i>arg2</i>	0
<i>arg1</i> е по – голяма от <i>arg2</i>	по – голяма от 0

Масива трябва да бъде сортиран в нарастващ ред с най – нисък адрес съдържащ най – малкия елемент . Ако масива не съдържа ключа се връща нулев указател . Свързана функция е *qsort ()* .

div ()

```
#include <stdlib . h >


```

Функцията *div ()* връща частното и остатъка от операцията *numerator / denominator* (числител / знаменател) в структура от тип *div_t* . Структурата тип *div_t* е дефинирана в **STDLIB.H** и има само тези две полета :

```
int quot ; /* частно */
int rem ; /* остатък */
```

Свързана функция е *ldiv ()* .

exit ()

```
#include <stdlib . h >
void exit ( int exit_code ) ;
```

Функцията *exit ()* предизвиква незабавно нормално прекъсване на програма . Стойността на *exit_code* се подава на извикващия процес обикновено операционната система , ако средата поддържа това . По конвенция ако стойността на *exit_code* е 0 , или **EXIT_SUCCESS** , означава нормално прекъсване на програмата . Ненулева стойност или **EXIT_FAILURE** се използва да покаже имплементационно – дефинирана грешка . Свързани функции са *atexit ()* и *abort ()* .

getenv ()

```
#include <stdlib . h >
char *getenv ( const char *name ) ;
```

Функцията *getenv ()* връща указател към информация за средата свързана със низ указван от *name* в имплементационно – дефинирана таблица за информация за средата . Средата на програмата може да включва такива неща като наименования на пътища и налични устройства . Точното естество на тези данни зависи от изпълнението . Ще трябва да проверите ръководството на компилатора за подробности . Ако извикването на *getenv ()* е със аргумент който не съвпада със нищо от данните за средата , се връща нулев указател . Свързана функция е *system ()* .

labs ()

```
#include <stdlib . h >
long labs ( long num ) ;
```

Функцията *labs ()* връща абсолютната стойност на *num* . Свързана функция е *abs ()* .

ldiv ()

```
#include <stdlib.h>
```

```
ldiv_t ldiv (long numerator, long denominator);
```

Функцията **ldiv ()** връща частното и остатъка от операцията **numerator / denominator** (числител / знаменател) в структура от тип **div_t**. Структурата тип **div_t** е дефинирана в **STDLIB.H** и има само тези две полета:

```
int quot; /* частно */
```

```
int rem; /* остатък */
```

Свързана функция е **div ()**.

longjmp ()

```
#include <setjmp.h>
```

```
void longjmp (jmp_buf envbuf, int status);
```

Функцията **longjmp ()** предизвиква изпълнението на програмата да се върне в точката на последното извикване на **setjmp ()**. Тези две функции предоставят начин за прескачане между функции. Забележете че се изисква хедъра **SETJMP.H**. Функцията **longjmp ()** действа чрез пренастройване на стека състоянието както е описан в **envbuf**, който трябва да е установен чрез по-ранно извикване на **setjmp ()**. Това предизвиква изпълнението на програмата да се върне на израза следващ извикването на **setjmp ()**. Така компютъра се “измамва” да смята че никога не е напуснал функцията която извиква **setjmp ()**. (Като някакво образно обяснение, функцията **longjmp ()** прескача през времето и (паметта) пространството към предишна точка във програмата без изпълняването на нормалния процес за връщане към функция) Буфера **envbuf** е от тип **jmp_buf** който е дефиниран в хедъра **SETJMP.H**. Буфера трябва да бъде установен чрез извикване на **setjmp ()** преди извикването на **longjmp ()**. Стойността на **status** става връщана стойност на **setjmp ()** и може да бъде изисквана за определяне откъде идва прескачането. Единствената стойност която не е разрешена е **0**. Най-общата употреба на **longjmp ()** е за връщане от дълбоко вложени цикли когато стане грешка. Свързана функция е **setjmp ()**.

qsort ()

```
#include <stdlib.h>
```

```
void qsort (void *buf, size_t num, size_t size,
```

```
int (*compare) (const void *, const void *));
```

Функцията **qsort ()** сортира масива указван от **buf** чрез използване на **Quicksort** (разработен от **C. A. R Hoare**). **Quicksort** е считан за най-добрия общо сортиращ алгоритъм. След връщане, масива ще бъде сортиран. Броя на елементите е определен от **num**, а размера (в байтове) на всеки елемент е **size**. Функцията указвана чрез **compare** се използва да сравни два елемента от масива. Формата на **compare** функцията трябва да бъде както следва:

```
int func_name (const void *arg1, const void arg2);
```

Тя трябва да връща стойност както е описано тук:

Сравнение

arg1 е по-малко от **arg2**

arg1 е равна на **arg2**

Върната стойност

по-малка от **0**

0

arg1 е по – голяма от *arg2* по – голяма от 0

Масива е сортиран в нарастващ ред с най – нисък адрес съдържащ най – малкия елемент . Свързана функция е *bsearch ()* .

Програмен съвет

Когато използвайте *qsort ()* искате да сортирате масива в намаляващ ред (тоест от голям към малък) просто обърнете израза използван в сравняващата функция . Тогава сравняващата функция връща следните стойности :

Сравнение	Върната стойност
<i>arg1</i> е по – малко от <i>arg2</i>	по – голяма от 0
<i>arg1</i> е равна на <i>arg2</i>	0
<i>arg1</i> е по – голяма от <i>arg2</i>	по – малка от 0

Също ако искате да използвате *bsearch ()* в масив който е сортиран в намаляващ ред , ще трябва да използвате обърнатата сравняваща функция .

raise ()

#include < signal . h >

int raise (int signal) ;

Функцията *raise ()* изпраща сигнал определен от *signal* към изпълняваната програма . Тя връща 0 ако е успешна и ненула в противен случай . Тя използва хедъра **SIGNAL.H** . Следните сигнали са дефинирани от стандарта **ANSI C** . Разбира се вашия компилатор може да доставя и допълнителни сигнали .

Макрос	значение
SIGABRT	Прекъсваща грешка
SIGFPE	Грешка в плаващата точка
SIGILL	Лоша инструкция
SIGINT	Потребителя е натиснал Ctrl - C
SIGSEGV	Непозволен достъп до паметта
SIGTERM	Прекъсва програмата

Свързана функция е *signal ()* .

rand ()

#include < stdlib . h >

int rand (void) ;

Функцията *rand ()* генерира поредица от псевдослучайни числа . Всеки път когато се извика се връща цяло число между 0 и **RAND_MAX** . Свързана функция е *srand ()* .

setjmp ()

#include < setjmp . h >

int setjmp (jmp_buf envbuf) ;

Функцията *setjmp ()* запазва съдържанието на системния стек в буфера *envbuf* за следващо извикване на *longjmp ()* . Тя използва хедъра **SETJMP.H** . Функцията *setjmp ()* връща 0 при извикване . Обаче *longjmp ()* подава аргумент на *setjmp ()* когато се изпълни и това е тази стойност (винаги ненула) която ще бъде стойност на *setjmp ()* след като е станало извикване

на *longjmp()* (Виж “*longjmp()*” за допълнителна информация). Свързана функция е *longjmp()* .

signal ()

```
#include <signal . h >
```

```
void ( signal ( int signal , void ( *func ) ( int ) ) ) ( int ) ;
```

Функцията *signal()* регистрира функцията определена от *func* като манипулатор за сигнала определен от *signal* . Така функцията указвана от *func* ще бъде извиквана когато се получи *signal* от програмата . Стойността на *func* може да бъде адреса на сигнал – манипулираща функция или един от следните макроси дефинирани в **SIGNAL.H** :

Макрос

SIG_DFL

SIG_IGN

Значение

Използва подразбиращо се манипулиране на сигнала

Отхвърля сигнала

Ако адреса на функцията е използван , определения манипулатор ще бъде изпълнен когато се получи неговия сигнал . При успех *signal()* връща адреса на предишната дефинирана функция за определения сигнал . При грешка се връща **SIG_ERR** (дефиниран в **SIGNAL.H**) . Свързана функция е *raise()* .

srand ()

```
#include <stdlib . h >
```

```
void srand ( unsigned seed ) ;
```

Функцията *srand()* се използва за установяване на началната точка на поредицата генерирана от *rand()* (Функцията *rand()* връща псевдослучайни числа) . *srand()* се обикновено използва да разреши на множество пуснати програми да използват различни поредици от псевдослучайни числа чрез определяне на различни начални точки . Обратно , вие също можете да използвате *srand()* да генерира същата псевдослучайна поредица отново и отново чрез извикването ѝ със същото число преди началото на всяка поредица . Свързана функция е *rand()* .

strtod ()

```
#include <stdlib . h >
```

```
double strtod ( const char *start , char **end ) ;
```

Функцията *strtod()* конвертира низа представен от число съхранено в низа указван от *start* в *double* и връща резултата . Функцията работи по следния начин : Първо всеки празен символ в низа указван от *start* се отрязва . След това се прочита всеки символ в числото . Всеки символ който не е част от число със плаваща точка ще прекъсне процеса . Това включва празните символи , пунктуациите (различни от точка) и символите без “**E**” и “**e**” . Накрая *end* се установява да сочи остатък от първоначалния низ . Това означава че ако *strtod()* се извика със “**100.00 Pliers**” ще се върне стойността **100.00** а *end* ще сочи интервала който предхожда “**Pliers**” . Ако няма конвертиране се връща **0** . Ако стане препълване *strtod()* връща или **HUGE_VAL** или **-HUGE_VAL** (показващи положително или отрицателно препълване) и глобалната променлива *errno* се установява на **ERANGE** . Свързана функция е *atof()* .

strtol ()

#include <stdlib.h>

long strtol (const char *start, char **end, int radix);

Функцията **strtol ()** конвертира низа представен от число съхранявано в низа указван от **start** в **long** и връща резултата. Основата на числото се определя чрез **radix**. Ако **radix** е **0**, основата се определя по правилата които управляват константните спецификации. Ако **radix** е различно от **0**, то трябва да бъде в интервала от **2** до **36**. Функцията **strtol ()** работи по следния начин: Първо всеки празен символ в низа указван от **start** се отрязва. След това се прочита всеки символ в числото. Всеки символ който не може да е част от **long** ще прекъсне процеса. Това включва празните символи, пунктуациите и символите. Накрая **end** се установява да сочи остатъка от първоначалния низ. Това означава че ако **strtol ()** се извика със “**100 Pliers**“ ще се върне стойността **100** а **end** ще сочи интервала който предхожда “**Pliers**“. Ако резултата не може да се представи чрез **long**, **strtol ()** ще върне или **LONG_MAX** или **LONG_MIN** и глобалната променлива **errno** ще се установи на **ERANGE**, показвайки грешка в интервала. Ако няма конвертиране се връща **0**. Свързана функция е **atol ()**.

strtoul ()

#include <stdlib.h>

unsigned long strtoul (const char *start, char **end, int radix);

Функцията **strtoul ()** конвертира низа представен от число съхранявано в низа указван от **start** в **unsigned long** и връща резултата. Основата на числото се определя чрез **radix**. Ако **radix** е **0**, основата се определя по правилата които управляват константните спецификации. Ако **radix** е различно от **0**, то трябва да бъде в интервала от **2** до **36**. Функцията **strtoul ()** работи по следния начин: Първо всеки празен символ в низа указван от **start** се отрязва. След това се прочита всеки символ в числото. Всеки символ който не може да е част от **unsigned long** ще прекъсне процеса. Това включва празните символи, пунктуациите и символите. Накрая **end** се установява да сочи остатъка от първоначалния низ. Това означава че ако **strtoul ()** се извика със “**100 Pliers**“ ще се върне стойността **100** а **end** ще сочи интервала който предхожда “**Pliers**“. Ако резултата не може да се представи чрез **unsigned long**, **strtoul ()** връща **ULONG_MAX** и глобалната променлива **errno** ще се установи на **ERANGE**, показвайки грешка в интервала. Ако няма конвертиране се връща **0**. Свързана функция е **strtol ()**.

system ()

#include <stdlib.h>

int system (const char *str);

Функцията **system ()** подава низа указван от **str** като команда към командния процесор на операционната система. Ако **system ()** се извика с нулев указател, ще върне ненула. В противен случай се връща **0** (Някакъв C/C++ код ще бъде изпълнен на някои системи без операционна система и командни процесори, така че не трябва да приемате че в системата има команден

процесор). Връщаната стойност на `system()` зависи от изпълнението. Обикновено тя ще върне 0 ако командата бъде успешно изпълнена и ненула в противен случай. Свързана функция е `exit()`.

`va_arg()`, `va_start()` и `va_end()`

```
#include <stdarg.h>
type va_arg(va_list argptr, type);
void va_end(va_list argptr);
void va_start(va_list argptr, last_parm);
```

Макросите `va_arg()`, `va_start()` и `va_end()` работят заедно за да разрешат променлив брой аргументи да бъдат подадени на функция. Най-общия пример на функция която приема променлив брой аргументи е `printf()`. Типа `va_list` е дефиниран в `STDARG.H`. Общата процедура за създаване на функция приемаща променлив брой параметри е следния: Функцията трябва да има поне един известен параметър, но може и повече по-предни параметри в списъка със променливи параметри. Най-десния известен параметър е наречен `last_parm`. Името `last_parm` се използва като втори параметър при извикване на `va_start()`. Преди да бъде използван някой от параметрите с променлива дължина, аргументния указател `argptr` трябва да бъде инициализиран чрез извикване на `va_start()`. След това параметрите се връщат чрез извикване на `va_arg()` с `type` който е типа на следващия параметър. Накрая след като са прочетени всички параметри и по-ранно връщане от функцията, извикването на `va_end()` трябва да осигури че стека е правилно възстановен. Ако не се извика `va_end()` е много вероятен срив на програмата. Свързана функция е `vprintf()`.

Програмен съвет

Точната употреба на `va_start()`, `va_end()` и `va_end()` най-добре се илюстрира с пример. Тази програма използва `sum_series()` за да връща сумата на редица от числа. Първия аргумент съдържа броя на следващите аргументи. В този пример се сумират първите 5 елемента от следния ред:

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^n}$$

Изхода показва "0.968750".

/* Пример за променливи аргументи - сумиране на редица */

```
#include <stdio.h>
#include <stdarg.h>
double sum_series(int, ...);
int main(void)
{
    double d;
    d = sum_series(5, 0.5, 0.25, 0.125, 0.0625, 0.03125);
    printf("Sum of series is %f\n", d);
    return 0;
}
double sum_series(int num, ...)
```

```

{
    double sum = 0.0, t;
    va_list argptr;
    /* инициализиране на argptr */
    va_start ( argptr , num );
    /* сумиране на редицата */
    for ( ; num ; num -- ) {
        t = va_arg ( argptr , double );
        sum += t;
    }
    /* извършва правилно прекъсване */
    va_end ( argptr );
    return sum;
}

```

ГЛАВА 12 – Системата на C++ за I/O в стар стил

Понеже C++ включва цялата библиотека на C, се поддържа и системата за I/O на C. Обаче C++ също дефинира собствена класово – базирана, обектно – ориентирана I/O система която е позната като библиотеката **iostream**. Когато пишете C++ програми обикновено ще искате да използвате по – скоро библиотеката **iostream** отколкото C – базирания I/O. По време на писането на това съществуват две версии на входно – изходната библиотека в употреба: една стара базирана на оригиналните спецификации на C++ и една по – нова дефинирана от стандартизационния комитет **ANSI C++**. Днес повечето C++ компилатори поддържат и двете I/O библиотеки. И за щастие, повечето неща и в двете работят по един начин. Ако знете как да използвате едната, ще можете лесно да използвате и другата. Обаче има няколко важни разлики между двете. Първоначално оригиналните **iostream** класове бяха дефинирани в глобалното именовано пространство. Новата стандартна библиотека се съдържа в именованото пространство **std**. Второ, новата I/O библиотека е дефинирана чрез използване на цялостно, взаимно свързано множество от шаблонни класове и функции. Библиотеката в стар стил използва не – сложна, нешаблонизирана класова йерархия. За щастие, имената на класовете които ще използвате във вашите програми остават същите. Трето, новата I/O библиотека дефинира много нови типове данни. Четвърто, за използване на старата библиотека ще трябва да включите **.H** хедърните файлове такива като **IOSTREAM.H**. Тези хедърни файлове дефинират класовете в стар стил за I/O и ги слагат в глобалното именовано пространство. За сравнение за да използвате новата I/O стандартна библиотека включете хедъра в нов стил **<iostream>** във вашата програма. Поради разликите между новия и стария стил I/O библиотеки те ще бъдат описани отделно в този справочник. Тази секция описва библиотеката в стар стил. Следващата глава описва новата стандартна I/O библиотека.

Основни класове за потоци

Библиотеката за I/O в стар стил използва хедър файла **IOSTREAM.H**. Този файл дефинира основната класова йерархия която поддържа I/O операции. Ако ще извършвате файлов I/O ще трябва да включите **FSTREAM.H**. За да използвате базиран на масиви I/O ще трябва да включите **STRSTREAM.H**. Класът от най – ниско ниво е наречен **streambuf**. Този клас доставя основните входно – изходни операции. Той се използва главно за базов клас на други класове. Освен ако не правите ваши собствени I/O класове няма да използвате директно **streambuf**. Класа **ios** е базов клас на класовата йерархия която обикновено ще използвате когато използвате системата за I/O на C++. Той доставя форматиране, проверка за грешки и информация за състоянието. От **ios** са производни различни класове – понякога през междинни класове. Класовете производни пряко или непряко от **ios** които обикновено ще използвате са изброени тук:

Клас	Цел
istream	Общ вход
ostream	Общ изход
iostream	Общ вход / изход
ifstream	Файлов вход
ofstream	Файлов изход
fstream	Файлов вход / изход
istrstream	Базиран на масиви вход
ostrstream	Базиран на масиви изход
strstream	Базиран на масиви вход / изход

Предефинирани потоци в C++

Когато започне изпълнение на програма на C++ се отварят автоматично четири вградени потока, изброени тук:

Поток	значение	устройство по подразбиране
cin	Стандартен вход	клавиатура
cout	Стандартен изход	екран
cerr	Стандартен изход за грешки	екран
clog	Буферирана версия на cerr	екран

По подразбиране, стандартните потоци се използват да комуникират с конзолата. Обаче в среди които поддържат пренасочване на I/O (като **DOS**, **UNIX** или **Windows**) стандартните потоци могат да бъдат пренасочени към други устройства или файлове.

Форматни флагове

В системата за I/O на C++ всеки поток се свързва със множество от форматни флагове които контролират начина на форматиране на информацията в потока. В **ios** са дефинирани следните изброими стойности. Тези стойности се използват да установят или свалят форматните флагове:

<i>adjustfield</i>	<i>hex</i>	<i>scientific</i>	<i>stdio</i>
<i>basefield</i>	<i>internal</i>	<i>showbase</i>	<i>unitbuf</i>
<i>dec</i>	<i>left</i>	<i>showpoint</i>	<i>uppercase</i>
<i>fixed</i>	<i>oct</i>	<i>showpos</i>	
<i>floatfield</i>	<i>right</i>	<i>skipws</i>	

Понеже тези флагове са дефинирани в класа *ios*, ще трябва изрично да ги определите когато ги използвате в програма. Например за обръщане към *left*, ще трябва да напишете *ios :: left*. Когато е вдигнат флага *skipws* водещите празни символи (интервали, табулации и нови редове) се отхвърлят когато изпълнявате вход в потока. Когато *skipws* е свален празните символи не се отхвърлят. Когато флага *left* е вдигнат, изхода е ляво – подравнен. Когато *right* е вдигнат изхода е дясно – подравнен. Когато *internal* е вдигнат числовата стойност се допълва до запълване на полето със вмъкнати интервали между всеки знак или базов символ. Ако никой от тези флагове не е вдигнат изхода е дясно подравнен по – подразбиране. По подразбиране числовите стойности се извеждат със десетични числа. Обаче е възможно да променим числовата основа. Вдигането на флага *oct* предизвиква изхода да бъде показван в осмични числа. Вдигането на *hex* предизвиква изхода да бъде показван в шестнайсетични числа. За връщане на изхода към десетични числа, вдигнете флага *dec*. Вдигането на *showbase* води до показване на основата на числовите стойности. Например ако основата е шестнайсетична, стойността *1F* ще бъде показана като *0x1F*. По подразбиране когато е показано научно означение “e” е малка буква. Също когато е показана шестнайсетична стойност “x” е малка буква. Когато е вдигнат *uppercase* тези символи се показват със главни букви. Вдигането на флага *showpos* предизвиква водещия знак плюс да бъде показван пред положителните стойности. Вдигането на *showpoint* предизвиква десетичната точка и следващите я нули да бъдат показвани за целия изход със плаваща точка – когато трябва или не. Чрез вдигане на *scientific* числата с плаваща точка се показват със научно означение. Когато е вдигнат *fixed*, стойностите с плаваща точка се показват чрез използване на обикновено означение. Когато никой от тези флагове не е вдигнат компилатора избира подходящия начин за показване. Когато *unitbuf* е вдигнат, буфера се записва след всяка операция за въвеждане. Когато е вдигнат *stdio*, *stdout* и *stderr* се записват след всяко извеждане. Когато е общо обръщането към *oct*, *dec* и *hex* полетата, те могат едновременно да се вдигнат чрез *ios :: basefield*. Също така полетата *left*, *right* и *internal* могат да бъдат реферирани като *ios :: adjustfield*. Накрая полетата *scientific* и *fixed* могат да бъдат представени като *ios :: floatfield*. Форматните флагове се обикновено съхраняват в *long integer* и могат да бъдат вдигани от различни функции на класа *ios*.

I/O Манипулатори

В добавка към вдигането и свалянето на форматните флагове директно, има и друг начин да променим форматиращите параметри на потока. Този начин е чрез употреба на специални функции наречени манипулатори които могат да бъдат включени в израза за I/O. Манипулаторите дефинирани от старата I/O библиотека са показани в следващата таблица:

Манипулатор	Предназначение	Вход / изход
<i>dec</i>	Използва десетични цели числа	Вход / изход
<i>endl</i>	Извежда симол нов ред и записва потока	Изход
<i>ends</i>	Извежда 0	Изход
<i>flush</i>	Записва поток	Изход
<i>hex</i>	Използва шестнайсетични цели числа	Вход / изход
<i>oct</i>	Използва осмични цели числа	Вход / изход
<i>resetiosflags (long f)</i>	Сваля флаговете определени в <i>f</i>	Вход / изход
<i>setbase (int base)</i>	Установява числовата основа на <i>base</i>	Изход
<i>setfill (int ch)</i>	Установява запълващия символ на <i>ch</i>	Изход
<i>setiosflags (long f)</i>	Вдига флаговете определени в <i>f</i>	Вход / изход
<i>setprecision (int p)</i>	Установява броя цифри за точността	Изход
<i>setw (int w)</i>	Установява ширината на полето на <i>w</i>	Изход
<i>ws</i>	Прескача водещия празен интервал	Вход

За да получите достъп до манипулаторите приемащи параметри , такива като *setw ()* ще трябва да включите **IOMANIP.H** във вашата програма .

Програмен съвет

Както знаете има два вида манипулатори : със и без параметри . Докато създаването на параметризирани манипулатори е извън обхвата на тази книга , то е съвсем лесно да създадете ваши манипулатори без параметри . Всички изходни манипулатори без параметри имат тази структура :

```
ostream &manip – name ( ostream &stream ) ;
{
    // вашия код е тук
    return stream ;
}
```

Тук **manip – name** е името на манипулатора . Забележете че се връща псевдоним на поток от тип **ostream** . Това е необходимо ако манипулатора се използва като част от по – голям I/O израз . Важно е да разберете че дори макар че манипулатора има като единствен аргумент псевдоним към потока върху който въздейства не се използва никакъв аргумент когато манипулатора

е включен в изходна операция. Всички входни манипулатори без параметри имат тази структура:

```
istream &manip – name ( istream &stream ) ;
{
    // вашия код е тук
    return stream ;
}
```

Входния манипулатор получава псевдоним за потока за който е бил извикан. Този поток трябва да бъде върнат от манипулатора. Тук е пример за прост изходен манипулатор наречен **setup()**. Той извършва ляво подравняване, установява ширината на полето на **10** и определя знака за долар (\$) като запълващ символ.

```
#include <iostream . h >
#include <iomanip . h >
ostream &setup ( ostream &stream ) ;
{
    stream . setf ( ios : : left ) ;
    stream << setw ( 10 ) << setfill ( ' $ ' ) ;
    return stream ;
}
int main ( )
{
    cout << 10 << “ “ << setup << 10 ;
    return 0 ;
}
```

Запомнете: решаващо е вашия манипулатор да връща **stream**. Ако това не е така, вашия манипулатор няма да може да се използва в серия от входни или изходни операции.

iostream функции в стар стил

Най – често използваните **iostream** функции в стар стил са описани тук:

```
bad ( )
#include <iostream . h >
int bad ( ) const ;
```

Функцията **bad()** е член на **ios**. Функцията **bad()** връща нула ако е станала фатална I/O грешка в свързания поток, в противен случай се връща 0. Свързана функция е **good()**.

```
clear ( )
#include <iostream . h >
void clear ( int flags = 0 ) ;
```

Функцията **clear()** е член на **ios**. Функцията **clear()** изчиства флаговете за състоянието свързани с поток. Ако **flags** е **0** (както е по – подразбиране) тогава всички флагове за грешки се изчистват (установяват се на **0**). В противен случай флаговете ще бъдат установени на стойността на **flags**. Свързана функция е **rdstate()**.

eatwhite ()

```
#include <iostream . h >
void eatwhite ( ) ;
```

Функцията *eatwhite ()* е член на *istream* . Функцията *eatwhite ()* чете и отхвърля всички водещи празни интервали от свързания входен поток и премества *get* указателя на първия не – празен символ . Свързана функция е *ignore ()* .

eof ()

```
#include <iostream . h >
int eof ( ) const ;
```

Функцията *eof ()* е член на *ios* . Функцията *eof ()* връща ненула когато е достигнат края на свързания входен файл , в противен случай се връща **0** . Свързани функции са *bad ()* , *fail ()* , *good ()* , *rdstate ()* и *clear ()* .

fail ()

```
#include <iostream . h >
int fail ( ) const ;
```

Функцията *fail ()* е член на *ios* . Функцията *fail ()* връща ненула ако стане грешка във свързания поток , в противен случай връща **0** . Свързани функции са *good ()* , *eof ()* , *bad ()* , *clear ()* и *rdstate ()* .

fill ()

```
#include <iostream . h >
char fill ( ) const ;
char fill ( char ch ) ;
```

Функцията *fill ()* е член на *ios* . По подразбиране когато трябва да бъде запълнено поле , то се запълва със интервали . Обаче можете да промените запълващия символ чрез използване на функцията *fill ()* и определяйки новия запълващ символ в *ch* . Функцията ще върне стария запълващ символ . За да получите текущия запълващ символ , използвайте първата форма на *fill ()* , която връща текущия запълващ символ . Свързани функции са *precision ()* и *width ()* .

flags ()

```
#include <iostream . h >
long flags ( ) const ;
long flags ( long f ) ;
```

Функцията *flags ()* е член на *ios* . Първата форма на *flags ()* просто връща текущите форматни настройки на флаговете в свързания поток . Втората форма на *flags ()* установява всички форматни флагове свързани с поток чрез *f* . Тази версия също връща предишните настройки . Свързани функции са *unsetf ()* и *setf ()* .

flush ()

```
#include <iostream . h >
ostream &flush ( ) ;
```

Функцията *flush()* е член на *ostream*. Функцията *flush()* предизвиква буфера свързан към определения изходен поток да бъде физически записан на диска. Функцията връща псевдоним към свързания със нея поток. Свързани функции са *put()* и *write()*.

fstream()*, *ifstream()* и *ofstream()

```
#include <fstream.h>
fstream();
fstream(const char *filename, int mode,
         int access = filebuf::openprot);
fstream(int fd);
fstream(int fd, char *buf, int size);
ifstream();
ifstream(const char *filename, int mode = ios::in,
         int access = filebuf::openprot);
ifstream(int fd);
ifstream(int fd, char *buf, int size);
ofstream();
ofstream(const char *filename, int mode = ios::out,
         int access = filebuf::openprot);
ofstream(int fd);
ofstream(int fd, char *buf, int size)
```

fstream(), *ifstream()* и *ofstream()* са конструкторите съответно на класовете *fstream*, *ifstream* и *ofstream*. Версиите без параметри на *fstream()*, *ifstream()* и *ofstream()* създават поток който не е свързан със никакъв файл. Този поток може да бъде свързан към файл чрез използване на *open()*. Версиите на *fstream()*, *ifstream()* и *ofstream()* които приемат име на файл като първи параметър са най-често използвани в приложните програми. Въпреки че те са изцяло предназначени да отворят файл чрез използване на функцията *open()*, в повечето случаи няма да го правите понеже тези *fstream()*, *ifstream()* и *ofstream()* функции – конструктори автоматично отворят файла когато потока е създаден. Функциите конструктори имат еднакви параметри и действие като функцията *open()* (Виж *open()* за детайли). По тази причина най-общия начин за отваряне на файл е този пример:

```
ifstream mystream("myfile");
```

Ако по някаква причина файла не може да бъде отворен, стойността на променливата на свързания поток ще бъде 0. По тази причина и при използване на функция конструктор и при изрично определяне извикване на *open()*, ще трябва да се уверите че файла действително е бил отворен чрез проверка стойността на потока. Версиите на *fstream()*, *ifstream()* и *ofstream()* които приемат само един параметър, винаги валиден файлов описател, създават поток и тогава свързват този поток със файловия описател определен от *fd*. Версиите на *fstream()*, *ifstream()* и *ofstream()* които приемат файлов описател, указател към буфер и размер, създават поток и го свързват със файловия описател определен от *fd*. *buf* трябва да бъде указател към памет която ще служи като буфер, а *size* определя дължината на буфера в байтове (

ако *buf* е 0 и/или *size* е 0, няма буфериране). Свързани функции са *close()* и *open()*.

gcount()

```
#include <iostream.h>
int gcount() const;
```

Функцията *gcount()* е член на *istream*. Функцията *gcount()* връща броя на прочетените символи при последната входна операция. Свързани функции са *get()*, *getline()* и *read()*.

get()

```
#include <iostream.h>
int get();
istream &get(char &ch);
istream &get(char *buf, int num, char delim = '\n');
istream &get(streambuf &buf, char delim = '\n');
```

Функцията *get()* е член на *istream*. Най – общо *get()* чете символи от входния поток. Формата без параметри на *get()* чете единствен символ от свързания поток и връща неговата стойност. Формата на *get()* която приема един псевдоним на символ, чете символ от свързания поток и слага стойността му в *ch*. Тя връща псевдоним за потока (забележете че *ch* може също да бъде от тип **unsigned char*** или **signed char***). Формата на *get()* която приема три параметъра чете символи в масива указван от *buf* докато или бъдат прочетени *num* символа или се срещне символа определен от *delim*. Масива указван от *buf* ще бъде нулево – терминиран от *get()*. Ако няма определен параметър *delim* по – подразбиране символа за нов ред действа като разграничител. Ако се срещне разграничителния символ във входния поток, той не се извлича. Вместо това той остава в потока до следващата водна операция. Тази функция връща псевдоним за потока (забележете че *buf* може също да бъде от тип **unsigned char*** или **signed char***). Формата на *get()* която приема два параметъра чете символи от входния поток в *streambuf* (или произведен) обект. Символите се четат докато не се срещне определения разграничител. Тя връща псевдоним за потока. Свързани функции са *put()*, *read()* и *getline()*.

getline()

```
#include <iostream.h>
istream &getline(char *buf, int num, char delim = '\n');
```

Функцията *getline()* е член на *istream*. Функцията *getline()* чете символи в масива указван от *buf* докато или бъдат прочетени *num* символа или бъде срещнат символа определен от *delim*. Масива указван от *buf* ще бъде нулево – терминиран от *getline()*. Ако няма определен параметър от *delim* по – подразбиране символа за нов ред действа като разграничител. Ако разграничителния символ се срещне във входния поток, той се извлича но не се слага в *buf*. Тази функция връща псевдоним за потока (забележете че *buf* може също да бъде от тип **unsigned char*** или **signed char***). Свързани функции са *get()* и *read()*.

good ()

```
#include <iostream . h >
```

```
int good ( ) const ;
```

Функцията **good ()** е член на **ios**. Функцията **good ()** връща ненула ако не се срещнат I/O грешки във свързания поток, в противен случай тя връща **0**. Свързани функции са **bad ()**, **fail ()**, **eof ()**, **clear ()** и **rdstate ()**.

ignore ()

```
#include <iostream . h >
```

```
istream &ignore ( int num = 1 , int delim = EOF ) ;
```

Функцията **ignore ()** е член на **istream**. Вие може да използвате член-функцията **ignore ()** да чете и отхвърля символи от входния поток. Тя чете и отхвърля символи докато или бъдат отхвърлени **num** символа (**1** по – подразбиране) или се срещне символа определен от **delim** (**EOF** по – подразбиране). Ако се срещне разграничителния символ, той се премахва от входния поток. Функцията връща псевдоним за потока. Свързани функции са **get ()** и **getline ()**.

open ()

```
#include <fstream . h >
```

```
void open ( const char *filename , int mode ,  
            int access = filebuf :: openprot ) ;
```

Функцията **open ()** е член на **fstream**, **ifstream** и **ofstream**. Файл се свързва с поток чрез използване на функцията **open ()**. Тук, **filename** е името на файла което може да включва и пътя. Стойността на **mode** определя как да се отвори файла. Тя трябва да бъде една (или повече) от тези стойности:

```
ios :: app
```

```
ios :: nocreate
```

```
ios :: ate
```

```
ios :: noreplace
```

```
ios :: binary
```

```
ios :: out
```

```
ios :: in
```

```
ios :: trunk
```

Вие можете да комбинирате две или повече от тези стойности като ги оградите заедно във кръгли скоби. Включването на **ios :: app** предизвиква целия изход към файла да бъде добавен в края му. Тази стойност може да бъде използвана само с файлове със възможност за изход. Включването на **ios :: ate** предизвиква търсене към края на файла до съвпадение когато файла е отворен. Въпреки че **ios :: ate** предизвиква отместване към края на файла I/O операциите могат да се извършат навсякъде във файла. Стойността **ios :: binary** предизвиква файла да бъде отворен за двоични I/O операции. По – подразбиране файл се отваря в текстов режим. Стойността **ios :: in** определя че файла има възможност за вход. Стойността **ios :: out** определя че файла има възможност за изход. Обаче създаването на поток със използване на **ifstream** подразбира вход, а създаването на файл чрез използване на **ofstream** е по – подразбиране изход, така че в тези случаи не е необходимо да подавате тези стойности. Стойността **ios :: trunk** предизвиква съдържанието на предишния съществуващ файл с това име да бъде унищожен и файла да бъде скъсен до нулева дължина. Включването на **ios :: nocreate** предизвиква функцията **open ()** да пропадне ако файла не съществува. Стойността **ios ::**

noreplace предизвиква функцията *open()* да пропадне ако файла вече съществува и не са вдигнати също *ios::ate* или *ios::app*. Стойността на *access* определя как да бъде достъпен файла. Негова подразбираща се стойност е *filebuf::openprot* (*filebuf* е базов клас на файловите класове) което означава нормален файл. Проверете документацията на компилатора за други разрешени стойности на *access*. Когато отваряте файл и двете *mode* и *access* са по – подразбиране. Когато отваряте входен файл, *mode* ще е по – подразбиране *ios::in*. Когато отваряте изходен файл, *mode* ще е по – подразбиране *ios::out*. Във всички случаи *access* е по – подразбиране за нормален файл. Например този фрагмент отваря файл наречен *test* за изход:

```
out.open("test"); // подразбира се изход и
                  //нормален файл
```

За отваряне на поток за вход и изход ще трябва да определите и двете стойности на *mode ios::in* и *ios::out* както е показано тук:

```
mystream.open("test", ios::in | ios::out);
```

Никакви подразбиращи стойности не се подават на *mode* когато отваряме файлове за четене/запис. Във всички случаи ако *open()* пропадне потока ще бъде 0. По тази причина преди да използвате файл ще трябва да проверите да се уверите че отварящата операция е успешна. Свързани функции са *close()*, *fstream()*, *ifstream()* и *ofstream()*.

Програмен съвет

За да четете от или да записвате във текстов файл, просто използвайте операторите << и >> с потока който сте отворили. Например следващата програма записва цяло число, стойност със плаваща точка и низ във файла наречен *TEST* и след това ги прочита отново:

```
#include <iostream.h>
#include <fstream.h>
int main()
{
    ofstream out("test");
    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }
    // изходни данни
    out << 10 << " " << 123.33 << "\n";
    out << "This is a short text file.\n";
    out.close();
    // сега ги четеме обратно
    char ch;
    int i;
    float f;
    char str[80];
    ifstream in("test");
    if(!in) {
        cout << "Cannot open file.\n";
```

```

        return 1 ;
    }
    in >> i ;
    in >> f ;
    in >> ch ;
    in >> str ;
    cout << "Here is the data : " ;
    cout << i << " " << f << " " << ch << " \n " ;
    cout << str ;
    in . close ( ) ;
    return 0 ;
}

```

Когато четете текстов файл използвайки оператора >> запомнете че ще се извършат някои символни преобразувания . Например празните символи ще се изпуснат . Ако искате да предотвратите всякакви символни преобразувания , ще трябва да отворите файла за двоичен I/O и да използвате двоичните функции на C++ .

peek ()

```

#include <iostream . h >
int peek ( ) ;

```

Функцията *peek ()* е член на *istream* . Функцията *peek ()* връща следващия символ в потока или EOF ако е достигнат края на файла . Ако не е така при всички случаи тя премахва символа от потока . Свързана функция е *get ()* .

precision ()

```

#include <iostream . h >
int precision ( ) const ;
int precision ( int p ) ;

```

Функцията *precision ()* е член на *ios* . По – подразбиране се показват шест цифри точност когато се извежда стойност със плаваща точка . Обаче със използването на втората форма на *precision ()* ще можете да установите това число на стойността определена в *p* . Връща се първоначалната стойност . Първата версия на *precision ()* връща текущата стойност . Свързани функции са *width ()* и *fill ()* .

put ()

```

#include <iostream . h >
ostream &put ( char ch ) ;

```

Функцията *put ()* е член на *ostream* . Функцията *put ()* записва *ch* във свързания изходен поток . Тя връща псевдоним за потока . Свързани функции са *write ()* и *get ()* .

putback ()

```

#include <iostream . h >
istream &putback ( char ch ) ;

```

Функцията *putback ()* е член на *istream* . Функцията *putback ()* връща *ch* във свързания входен поток .

Забележка : *ch* трябва да бъде последния прочетен символ от потока .
Свързана функция е *peek ()* .

rdstate ()

```
#include <iostream . h >
int rdstate () const ;
```

Функцията *rdstate ()* е член на *ios* . Функцията *rdstate ()* връща състоянието на свързания поток . Системата за I/O на C++ поддържа статус информация за резултата от всяка I/O операция отнасяща се до всеки активен поток . Текущото състояние на I/O системата се съдържа в цяло число в което са кодирани следните флагове :

Име	Значение
<i>ios :: goodbit</i>	Не са станали грешки
<i>ios :: eofbit</i>	Срещнат е края на файла
<i>ios :: failbit</i>	Станала е нефатална I/O грешка
<i>ios :: badbit</i>	Станала е фатална I/O грешка

Тези флагове са изброени във *ios . rdstate ()* връща 0 (*ios :: goodbit*) когато не е станала грешка , в противен случай се установява бита за грешка . Свързани функции са *eof ()* , *good ()* , *bad ()* , *clear ()* и *fail ()* .

read ()

```
#include <iostream . h >
istream &read ( char *buf , int num ) ;
```

Функцията *read ()* е член на *istream* . Функцията *read ()* чете *num* байта от свързания входен поток и ги слага във буфер указван от *buf* (забележете че *buf* може също да бъде от тип **unsigned char *** или **signed char ***) . Ако е достигнат края на файла преди да са прочетени *num* символа , *read ()* просто спира и буфера ще съдържа колкото символа е имало (Виж “ *gcount ()* ”) . *read ()* връща псевдоним за потока . Свързани функции са *gcount ()* , *get ()* , *getline ()* и *write ()* .

seekg () и seekp ()

```
#include <iostream . h >
istream &seekg ( streamoff offset , ios :: seek_dir origin ) ;
istream &seekg ( streampos position ) ;
ostream &seekp ( streamoff offset , ios :: seek_dir origin ) ;
ostream &seekp ( streampos position ) ;
```

Функцията *seekg ()* е член на *istream* , а функцията *seekp ()* е член на *ostream* . В системата за I/O на C++ изпълнявате произволен достъп чрез използване на функциите *seekg ()* и *seekp ()* . Към този момент системата на C++ за I/O управлява два указателя свързани със файл . Единия е **get pointer** който определя къде във файла ще стане следващата входна операция . Другия е **put pointer** който определя къде във файла ще стане следващата изходна операция . Всеки път когато се извърши въвеждане или извеждане съответния указател се увеличава автоматично в последователност . Обаче чрез използване на *seekg ()* и *seekp ()* е възможно да достигнете файла по непоследователен начин . Двупараметровата версия на *seekg ()* премества получаващия указател на *offset* брой байтове от мястото определено от *origin* . Двупараметровата

версия на *seekp()* премества поставящия указател на *offset* брой байтове от мястото определено от *origin*. Параметърът *offset* е от тип *streamoff* който е дефиниран в **IOSTREAM.H**. Обекта *streamoff* е способен да съдържа най-голямата валидна стойност която може да има *offset*. Параметърът *origin* е от тип *ios::seek_dir* и е изброяване което има тези стойности:

<i>ios::beg</i>	Отместване от началото
<i>ios::cur</i>	Отместване от текущата позиция
<i>ios::end</i>	Отместване от края

Еднопараметровите версии на *seekg()* и *seekp()* преместват файловете указатели на положението определено от *position*. Тази стойност трябва да бъде получена преди това чрез извикване на *tellg()* или *tellp()* съответно. *streampos* е тип дефиниран във **IOSTREAM.H** който е способен да съдържа най-голямата валидна стойност която може да има *position*. Тези функции връщат псевдоним за свързания поток. Свързани функции са *tellg()* и *tellp()*

setf()

```
#include <iostream.h>
long setf(long flags);
long setf(long flags1, long flags2);
```

Функцията *setf()* е член на *ios*. Първата версия на *setf()* вдига форматните флагове определени от *flags* (Всички други флагове са непроменени). Например за вдигане на флага *showpos* може да използвате този израз:

```
stream.setf(ios::showpos);
```

Тук *stream* е потока върху когото искате да въздействате. Важно е да разберете че извикването на *setf()* се отнася само за определения поток. Няма концепция за извикване само на *setf()*. Казано различно няма концепция в C++ за глобално форматно състояние. Всеки поток поддържа собствената си форматна статус информация индивидуално. Когато искате да установите повече от един флаг можете да оградите заедно във кръгли скоби стойностите на флаговете които искате да установите. Втората версия на *setf()* въздейства само на флаговете които сте установили във *flags2*. Съответните флагове първо са пренастроени и след това вдигнати съгласно флаговете определени от *flags1*. Важно е да разберете че дори ако *flags1* съдържа друго множество флагове, само онези определени от *flags2* ще бъдат засегнати. И двете версии на *setf()* връщат предишните настройки на форматните флагове свързани със потока. Свързани функции са *unsetf()* и *flags()*.

setmode()

```
#include <fstream.h>
int setmode(int mode = filebuf::text);
```

Функцията *setmode()* е член на *ofstream* и *ifstream*. Функцията *setmode()* установява режима на свързания поток да бъде двоичен или текстови (по подразбиране е текстови). Валидни стойности за *mode* са *filebuf::text* и *filebuf::binary*. Функцията връща предишната настройка на режима или -1 ако стане грешка. Свързана функция е *open()*.

str ()

```
#include <strstream.h>
char *str ( ) ;
```

Функцията *str ()* е член на *strstream*. Функцията *str ()* “замразява” динамично заделения входен масив и връща указател към него. Веднъж щом динамичен масив е замразен, той не може да бъде използван отново за изход. По тази причина няма да искате да замразявате масив докато не изведете изцяло символите към него.

Забележка: Тази функция е за употреба със базиран на масиви I/O. Свързани функции са *strstream ()*, *istrstream ()* и *ostrstream ()*.

strstream () , istrstream () и ostrstream ()

```
#include <strstream.h>
strstream ( ) ;
strstream ( char *buf, int size, int mode ) ;
istrstream ( const char *buf ) ;
istrstream ( const char *buf, int size ) ;
ostrstream ( ) ;
ostrstream ( char *buf, int size, int mode = ios :: out ) ;
```

Конструктора *strstream ()* е член на *strstream*, конструктора *istrstream ()* е член на *istrstream* и конструктора *ostrstream ()* е член на *ostrstream*. Тези конструктори се използват да създадат базирани на масиви потоци които поддържат функциите на C++ за I/O базирани на масиви. За *ostrstream () buf* е указател към масив който събира символите записани във потока. Големината на масива се подава чрез параметъра *size*. По – подразбиране потока е отворен за нормален изход, но можете да определите различен режим чрез използване на параметъра *mode*. Разрешените стойности за *mode* са същите като тези използвани в *open ()*. За повечето цели *mode* ще бъде оставен по – подразбиране. Ако използвате безпараметровата версия на *ostrstream ()* ще бъде заделен автоматично динамичен масив. За еднопараметровата версия на *istrstream () buf* е указател към масива който ще бъде използван като източник на символи всеки път когато се извършва въвеждане във потока. Съдържанието на *buf* трябва да бъде нулево – термилирано. Обаче нулевия терминатор никога не се чете от масива. Ако искате само част от масива да бъде използвана за вход, използвайте двупараметровата форма на конструктора *istrstream ()*. Така само първите *size* елемента от масива указван от *buf* ще бъдат използвани. Този низ не трябва да бъде нулево – прекъснат понеже стойността на *size* определя размера на низа. За създаване на базиран на масиви поток способен за вход и изход използвайте *strstream ()*. В еднопараметровата версия, *buf* сочи към низ който ще бъде използван за I/O операции. Стойността на *size* определя размера на масива. Стойността на *mode* определя как да функционира масива. За нормални входно / изходни операции *mode* ще бъде *ios :: in | ios :: out*. За вход масива трябва да бъде нулево – прекъснат. Ако използвате безпараметровата версия на *strstream ()* буфера използван за I/O ще бъде

динамично заделен и режима ще бъде установен на четящи / записващи операции . Свързани функции са *str ()* и *open ()* .

sync_with_stdio ()

```
#include <iostream . h >
static void sync_with_stdio ( ) ;
```

Функцията *sync_with_stdio ()* е член на *ios* .Извикването на *sync_with_stdio ()* позволява стандартната C – базирана система за I/O да бъде безопасно използвана едновременно със класово – базираната система на C++ за I/O .

tellg () и tellp ()

```
#include <iostream . h >
streampos tellg ( ) ;
streampos tellp ( ) ;
```

Функцията *tellg ()* е член на *istream* , а функцията *tellp ()* е член на *ostream* . Системата за I/O на C++ управлява два указателя свързани със файл . Единия е **get pointer** който определя къде във файла ще стане следващата входна операция . Другия е **put pointer** който определя къде във файла ще стане следващата изходна операция . Всеки път когато се извърши въвеждане или извеждане съответния указател се увеличава автоматично и последователно . Вие можете да определите текущата позиция на **get pointer** използвайки *tellg ()* и на **put pointer** използвайки *tellp ()* .*streampos* е тип дефиниран във **IOSTREAM . H** , който е способен да съдържа най – голямата стойност която всяка от функциите може да върне . Стойностите върнати от *tellg ()* и *tellp ()* могат да бъдат използвани като параметри съответно на *seekg ()* и *seekp ()* . Свързани функции са *seekg ()* и *seekp ()* .

unsetf ()

```
#include <iostream . h >
long unsetf ( long flags ) ;
```

Функцията *unsetf ()* е член на *ios* .Функцията *unsetf ()* се използва да сваля един или повече форматни флага . Флаговете определени от *flags* се свалят (всички други флагове остават непроменени) . Връщат се предишните флагови настройки . Свързани функции са *setf ()* и *flags ()* .

width ()

```
#include <iostream . h >
int width ( ) const ;
int width ( int w ) ;
```

Функцията *width ()* е член на *ios* . За да получите текущата ширина на полето използвайте първата форма на *width ()* . Тази версия връща текущата ширина на полето . За да установите ширината на полето използвайте втората форма . Така *w* става новата ширина на полето , а се връща старата стойност . Свързани функции са *precision ()* и *fill ()* .

write ()

```
#include <iostream . h >
```

```
ostream &write ( const char *buf, int num ) ;
```

Функцията *write ()* е член на *ostream*. Функцията *write ()* записва *num* байта в определения изходен поток от буфера указван чрез *buf* (забележете че *buf* може също да бъде от тип **unsigned char *** или **signed char ***). Тя връща псевдоним за потока. Свързани функции са *read ()* и *put ()*.

ГЛАВА 13 – Стандартната система на C++ за I/O

Както обясних в предишната секция има две версии на библиотеката за I/O на C++. Библиотеката в стар стил беше описана в предишната секция. Новата библиотека за I/O както е дефинирана от стандартизационния комитет ANSI / ISO е описана тук.

Използване на новата *iostream* библиотека

Има две основни разлики между старата и новата *iostream* библиотеки. Първо библиотеката в стар стил беше дефинирана в глобалното именовано пространство. Новата стандартна *iostream* библиотека се съдържа във именованото пространствено *std*. Второ, старата библиотека използва *.H* хедърни файлове. Новата библиотека използва хедъри в нов стил (които не използват *.H*). За да използвате новата стандартна *iostream* библиотека включете хедъра в нов стил **<iostream>** във вашата програма. След като направите това обикновено ще искате да поставите библиотеката във текущото именовано пространство чрез използване израз като този:

```
using namespace std ;
```

След израза *using* и двете библиотеки старата и новата работят по еднакъв начин. Така не трябва да конвертирате много (или нищо) от вашия съществуващ код. Не е необходимо да използвате израза *using* само описателно. Вместо това може да включите точно определен определител за именовано пространство всеки път когато се обръщате към членовете на класовете за I/O. Например следното изрично реферира *cout*:

```
std : : cout << " This is a test " ;
```

Разбира се ако извършвате широка употреба на *iostream* библиотеката включете израза *using* правещ нещата по – къси.

Базови класове за I/O

Новата *iostream* библиотека е дефинирана чрез комплексна йерархия от шаблонни класове. Вътрешно новата библиотека не използва същите имена на класове като старата библиотека. Например класовете от най – ниско ниво в старата библиотека са наречени *streambuf* и *ios*. В новата библиотека те са шаблонни класове наречени *basic_streambuf* и *basic_ios*. За щастие повечето от имената на класовете дефинирани в старата *iostream* библиотека се запазват от поредица *typedef* изрази които създават символно – базирани версии на I/O

класовете . Това е частичен списък със съпоставящите се шаблонни имена на класове със техните символно – базирани версии :

Шаблонен клас	Символно – базиран клас
<i>basic_streambuf</i>	<i>streambuf</i>
<i>basic_ios</i>	<i>ios</i>
<i>basic_istream</i>	<i>istream</i>
<i>basic_ostream</i>	<i>ostream</i>
<i>basic_iostream</i>	<i>iostream</i>
<i>basic_fstream</i>	<i>fstream</i>
<i>basic_ifstream</i>	<i>ifstream</i>
<i>basic_ofstream</i>	<i>ofstream</i>

Базираните на масиви I/O класове са също поддържани но не се одобряват . Нов код ще трябва да използва контейнерите описани в следващата секция . В добавка към символно – базирани класове , широко – символни I/O класове се също доставят със новата *iostream* библиотека . Техните имена са същите като на символно – базирани класове с изключение на това че започват със “w” . Например *wios* е широко – символната версия на *ios* . Понеже огромното множество от програмистите ще използват символно – базиран I/O , това са класовете описани в тази книга . Така когато се обръщаме към I/O класовете просто ще използваме техните *typedef* символно – базирани имена по – често отколкото техните вътрешни шаблонни имена . В случая тази книга ще използва името *ios* отколкото *basic_ios* .

Предефинирани потоци в C++

Когато използвате новата *iostream* библиотека се отварят автоматично следните потоци :

Поток	Значение
<i>cin</i>	Стандартен вход
<i>cout</i>	Стандартен изход
<i>cerr</i>	Стандартен изход за грешки
<i>clog</i>	Буферирана версия на <i>cerr</i>
<i>wcin</i>	Широко символна версия на <i>cin</i>
<i>wcout</i>	Широко символна версия на <i>cout</i>
<i>wcerr</i>	Широко символна версия на <i>cerr</i>
<i>wclog</i>	Широко символна версия на <i>clog</i>

По подразбиране , стандартните потоци се използват за комуникация със конзолата . Обаче в среди които поддържат пренасочване на I/O (като **DOS** , **UNIX** и **Windows**) стандартните потоци могат да бъдат пренасочени към други устройства или файлове .

Типа *streamsize*

Новата *iostream* библиотека дефинира няколко нови типа данни . Един от най – общите е наречен *streamsize* който е някаква форма на *integer* . Обект от тип *streamsize* е способен да съдържа най – големия брой байтове който ще бъде трансфериран в единствена операция за I/O .

Типа iostate

Текущото състояние на I/O потока е описано чрез обект от тип *iostate*, който е изброяване дефинирано в *ios* и включва тези членове:

Име	Значение
<i>goodbit</i>	Не са станали грешки
<i>eofbit</i>	Срещнат е края на файла
<i>failbit</i>	Станала е нефатална I/O грешка
<i>badbit</i>	Станала е фатална I/O грешка

Форматни флагове и типа fmtflags

Всеки поток е свързан със множество от форматни флагове които контролират начина на форматиране на информацията в потока. Новата *iostream* библиотека декларира изброяване наречено *fmtflags* в което са декларирани следните стойности:

<i>adjustfield</i>	<i>floatfield</i>	<i>right</i>	<i>skipws</i>
<i>basefield</i>	<i>hex</i>	<i>scientific</i>	<i>unitbuf</i>
<i>boolalpha</i>	<i>internal</i>	<i>showbase</i>	<i>uppercase</i>
<i>dec</i>	<i>left</i>	<i>showpoint</i>	
<i>fixed</i>	<i>oct</i>	<i>showpos</i>	

Тези стойности се използват за да установят или изчистят форматните флагове. Тези стойности са дефинирани в *ios* (технически, изброяването *fmtflags* е дефинирано в *ios_base*, който е базов клас на *basic_ios* но тази разлика не е важна за повечето програмни ситуации). Когато флага *skipws* е вдигнат, водещите празни символи (интервали, табулации и нови редове) се отхвърлят когато изпълняваме вход в потока. Когато *skipws* е свален, празните символи не се отхвърлят. Когато *left* е вдигнат изхода е ляво подравнен. Когато е вдигнат *right* изхода е дясно подравнен. Когато е вдигнат *internal* числото се допълва до запълване на полето чрез вмъкване на интервали между всеки знак или основен символ. Ако никой от тези флагове не е вдигнат изхода е дясно подравнен по – подразбиране. По – подразбиране числените стойности се извеждат в десетичен формат. Обаче е възможно да променим числовата основа. Вдигането на флага *oct* предизвиква изхода да се показва в осмични числа. Вдигането на *hex* причинява изхода да бъде показван в шестнайсетични числа. За връщане към десетичната база вдигнете флага *dec*. Вдигането на *showbase* предизвиква да бъде показвана базата на числата. Например, ако установената база е шестнайсетична, стойността **1F** ще бъде показана като **0x1F**. По – подразбиране когато е показано научно означение **e** – то е малка буква. Също когато е показана шестнайсетична стойност, **x** е малка буква. Когато е вдигнат *uppercase* тези символи се показват с главни букви. Вдигането на *showpos* е причина водещия знак плюс да се показва пред положителните стойности. Вдигането на *showpoint* причинява десетичната точка и следващите я нули да бъдат показвани за всички извеждани числа с плаваща точка – когато е нужно или не е. Чрез вдигане на флага *scientific* числовите стойности със плаваща точка се показват чрез използване на научно означение. Когато *fixed* е вдигнат стойностите със плаваща точка се показват със използване на нормално

означение. Когато никой флаг не е вдигнат, компилатора избира подходящия метод. Когато е вдигнат *unitbuf*, буфера се записва след всяка вмъкваща операция. Когато *boolalpha* е вдигнат, булевите стойности могат да бъдат въвеждани и извеждани чрез използване на ключовите думи *true* и *false*. Понеже обръщането към полетата *oct*, *dec* и *hex* общо, те могат да бъдат общо представени като *ios :: basefield*. Също полетата *left*, *right* и *internal* могат да бъдат представени като *ios :: adjustfield*. Накрая, полетата *scientific* и *fixed* могат да бъдат представени като *ios :: floatfield*.

I/O манипулатори

В добавка към директното вдигане и сваляне на форматните флагове, можете да промените форматните параметри на потока чрез употребата на специални функции наречени манипулатори, които могат да бъдат включени в израз за I/O. Стандартните манипулатори са показани в следващата таблица (някои от тези манипулатори не са поддържани от старата библиотека):

Манипулатор	Предназначение	Вход / изход
<i>boolalpha</i>	Вдига флага <i>boolalpha</i>	Вход / изход
<i>dec</i>	Вдига флага <i>dec</i>	Вход / изход
<i>endl</i>	Извежда символ нов ред и записва потока	Изход
<i>ends</i>	Извежда 0	Изход
<i>fixed</i>	Вдига флага <i>fixed</i>	Изход
<i>flush</i>	Записва потока	Изход
<i>hex</i>	Вдига флага <i>hex</i>	Вход / изход
<i>internal</i>	Вдига флага <i>internal</i>	Изход
<i>left</i>	Вдига флага <i>left</i>	Изход
<i>noboolalpha</i>	Сваля флага <i>boolalpha</i>	Вход / изход
<i>noshowbase</i>	Сваля флага <i>showbase</i>	Изход
<i>nshowpoint</i>	Сваля флага <i>showpoint</i>	Изход
<i>nshowpos</i>	Сваля флага <i>showpos</i>	Изход
<i>noskipws</i>	Сваля флага <i>skipws</i>	Вход
<i>nunitbuf</i>	Сваля флага <i>unitbuf</i>	Изход
<i>nuppercase</i>	Сваля флага <i>uppercase</i>	Изход
<i>oct</i>	Вдига флага <i>oct</i>	Вход / изход
<i>resetiosflags (long f)</i>	Сваля флаговете определени в <i>f</i>	Вход / изход
<i>right</i>	Вдига флага <i>right</i>	Изход
<i>scientific</i>	Вдига флага <i>scientific</i>	Изход
<i>setbase (int base)</i>	Установява числовата основа на <i>base</i>	Изход
<i>setfill (int ch)</i>	Установява запълващия символ на <i>ch</i>	Изход
<i>setiosflags (long f)</i>	Вдига флаговете определени в <i>f</i>	Вход / изход
<i>setprecision (int p)</i>	Установява броя цифри за точността	Изход

<i>setw (int w)</i>	Установява ширината на полето на <i>w</i>	Изход
<i>showbase</i>	Вдига флага <i>showbase</i>	Изход
<i>showpoint</i>	Вдига флага <i>showpoint</i>	Изход
<i>showpos</i>	Вдига флага <i>showpos</i>	Изход
<i>skipws</i>	Вдига флага <i>skipws</i>	Вход
<i>unitbuf</i>	Вдига флага <i>unitbuf</i>	Изход
<i>uppercase</i>	Вдига флага <i>uppercase</i>	Изход
<i>ws</i>	Прескача водещите празни интервали	Вход

За достъп до манипулаторите които приемат параметри такива като *setw ()*, ще трябва да включит *<iomanip>* във програмата .

Програмен съвет

Един от най – интересните форматни флагове добавени в новата **iostream** библиотека е **boolalpha** .Този флаг може да бъде вдигнат или директно или чрез вдигане на манипулаторите **boolalpha ()** или **noboolalpha ()** . Това което прави **boolalpha ()** интересен е че установяването му позволява да въвеждате и извеждате булеви стойности чрез използването на ключовите думи **true** и **false** . Обикновено трябва да въвеждате **1** за **true** и **0** за **false** .Например разгледайте следната програма :

```
// Демонстрира форматния флаг boolalpha
#include <iostream>
using namespace std;

int main ( )
{
    bool b ;
    cout << " Before setting boolalpha flag : \n " ;
    b = true ;
    cout << b << " " ;
    b = false ;
    cout << b << endl ;
    cout << " After setting boolalpha flag : \n " ;
    b = true ;
    cout << boolalpha << b << " " ;
    b = false ;
    cout << b << endl ;
    cout << " Enter a Boolean value : " ;
    cin >> boolalpha >> b ;
    cout << " You entered " << b ;

    return 0 ;
}
```

Тук е примерен изход :

```
Before setting boolalpha flag : 1 0
After setting boolalpha flag : true false
```

Enter a Boolean value : true

You entered true

Както може да видите щом веднъж флага *boolalpha* бъде установен булевите стойности могат да се въвеждат и извеждат чрез думите *true* и *false*. Както показва програмата ще трябва да установите флага отделно за *cin* и *cout*. Като всички форматни флагове установявайки *boolalpha* за един поток не предполага че той също е установен за другите.

Стандартни *iostream* функции

Най – често употребяваните функции в нов стил са описани тук.

bad ()

```
#include <iostream>
```

```
bool bad ( ) const ;
```

Функцията *bad ()* е член на *ios*. Функцията *bad ()* връща *true* ако е станала фатална I/O грешка в свързания поток, в противен случай се връща *false*. Свързана функция е *good ()*.

clear ()

```
#include <iostream>
```

```
void clear ( iostate flags = goodbit ) ;
```

Функцията *clear ()* е член на *ios*. Функцията *clear ()* изчиства флаговете за състоянието свързани с поток. Ако *flags* е *goodbit* (както е по – подразбиране) тогава всички флагове за грешки се изчистват (установяват се на **0**). В противен случай флаговете ще бъдат установени на стойността на *flags*. Свързана функция е *rdstate ()*.

eof ()

```
#include <iostream>
```

```
bool eof ( ) const ;
```

Функцията *eof ()* е член на *ios*. Функцията *eof ()* връща *true* когато е достигнат края на свързания входен файл, в противен случай се връща *false*. Свързани функции са *bad ()*, *fail ()*, *good ()*, *rdstate ()* и *clear ()*.

fail ()

```
#include <iostream>
```

```
bool fail ( ) const ;
```

Функцията *fail ()* е член на *ios*. Функцията *fail ()* връща *true* ако стане I/O грешка във свързания поток, в противен случай връща *false*. Свързани функции са *good ()*, *eof ()*, *bad ()*, *clear ()* и *rdstate ()*.

fill ()

```
#include <iostream>
```

```
char fill ( ) const ;
```

```
char fill ( char ch ) ;
```

Функцията *fill ()* е член на *ios*. По подразбиране когато трябва да бъде запълнено поле, то се запълва със интервали. Обаче можете да промените запълващия символ чрез използване на функцията *fill ()* и определяйки новия запълващ символ в *ch*. Функцията ще върне стария запълващ символ. За да

получите текущия запълващ символ, използвайте първата форма на *fill()*, която връща текущия запълващ символ. Свързани функции са *precision()* и *width()*.

flags()

```
#include <iostream>
fmtflags flags() const;
fmtflags flags(fmtflags f);
```

Функцията *flags()* е член на *ios*. Първата форма на *flags()* просто връща текущите форматни настройки на флаговете в свързания поток. Втората форма на *flags()* установява всички форматни флагове свързани с поток чрез *f*. Когато използвате тази версия битовия шаблон открит в *f* се копира във форматните флагове свързани със потока. Тази версия също връща предишните настройки. Свързани функции са *unsetf()* и *setf()*.

flush()

```
#include <iostream>
ostream &flush();
```

Функцията *flush()* е член на *ostream*. Функцията *flush()* предизвиква буфера свързан към определения изходен поток да бъде физически записан на диска. Функцията връща псевдоним към свързания със нея поток. Свързани функции са *put()* и *write()*.

fstream(), ifstream() и ofstream()

```
#include <fstream>
fstream();
fstream(const char *filename,
         openmode mode = ios::in | ios::out);
ifstream();
ifstream(const char *filename,
         openmode mode = ios::in);
ofstream();
ofstream(const char *filename,
         openmode mode = ios::out | ios::trunc);
```

fstream(), *ifstream()* и *ofstream()* са конструкторите съответно на класовете *fstream*, *ifstream* и *ofstream*. Версиите без параметри на *fstream()*, *ifstream()* и *ofstream()* създават поток който не е свързан със никакъв файл. Този поток може да бъде свързан към файл чрез използване на *open()*. Версиите на *fstream()*, *ifstream()* и *ofstream()* които приемат име на файл като първи параметър са най-често използвани в приложните програми. Въпреки че те са изцяло предназначени да отварят файл чрез използване на функцията *open()*, в повечето случай няма да го правите понеже тези *fstream()*, *ifstream()* и *ofstream()* функции – конструктори автоматично отварят файла когато потока е създаден. Функциите конструктори имат еднакви параметри и действие като функцията *open()* (Виж “*open()*” за детайли). Например, това е най-общия начин за отваряне на файл:

```
ifstream mystream("myfile");
```

Ако по някаква причина файла не може да бъде отворен, стойността на променливата на свързания поток ще бъде **0**. По тази причина и при използване на функция конструктор и при изрично определяне извикване на **open()**, ще трябва да се уверите че файла действително е бил отворен чрез проверка стойността на потока. Типа **openmode** е дефиниран в класа **ios_base**. Виж “**open()**” за детайли. Свързани функции са **close()** и **open()**.

gcount()

```
#include <iostream>
streamsize gcount() const;
```

Функцията **gcount()** е член на **istream**. Функцията **gcount()** връща броя на прочетените символи при последната входна операция. Свързани функции са **get()**, **getline()** и **read()**.

get()

```
#include <iostream>
int get();
istream &get(char &ch);
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
istream &get(streambuf &buf);
istream &get(streambuf &buf, char delim);
```

Функцията **get()** е член на **istream**. Най – общо **get()** чете символи от входния поток. Формата без параметри на **get()** чете единствен символ от свързания поток и връща неговата стойност.

get(char &ch) прочита символ от свързания поток и слага стойността му в **ch**. Тя връща псевдоним за потока.

get(char *buf, streamsize num) прочита символите в масив указван от **buf** докато или **num – 1** символа бъдат прочетени, достигнат е нов ред или е срещнат края на файла. Масива указван от **buf** ще бъде нулево – терминиран от **get()**. Ако е срещнат символа нов ред във входния поток, той не се извлича. Вместо това остава в потока до следващата входна операция. Тази функция връща псевдоним за потока.

get(char *buf, streamsize num, char delim) прочита символите в масива указван от **buf** докато или **num – 1** символа бъдат прочетени, символа определен от **delim** бъде открит или е срещнат края на файла. Масива указван от **buf** ще бъде нулево – терминиран от **get()**. Ако се срещне разграничителния символ във входния поток, той не се извлича. Вместо това той остава в потока до следващата водна операция. Тази функция връща псевдоним за потока.

get(streambuf &buf) чете символи от входния поток във **streambuf** обект. Символите се четят докато се достигне нов ред или се срещне края на файла. Тази функция връща псевдоним за потока.

get(streambuf &buf, char delim) чете символи от входния поток във **streambuf** обект. Символите се четят докато се открие символа определен от **delim** или се срещне края на файла. Тази функция връща псевдоним за потока.

Тя връща псевдоним за потока. Ако се срещне разграничителния символ във входния поток, той не се извлича. Свързани функции са *put()*, *read()* и *getline()*.

getline ()

```
#include <iostream>
istream &getline ( char *buf, streamsize num );
istream &getline ( char *buf, streamsize num, char delim );
```

Функцията *getline()* е член на *istream*.

*getline (char *buf, streamsize num)* чете символите в масив указван от *buf* докато или *num* – 1 символа бъдат прочетени, достигнат е нов ред или е срещнат края на файла. Масива указван от *buf* ще бъде нулево – термилиран от *getline()*. Ако се срещне символ за нов ред във входния поток, той се извлича, но не се слага във *buf*. Тази функция връща псевдоним за потока *getline (char *buf, streamsize num, char delim)* чете смволи в масив указван от *buf* докато или *num* – 1 символа бъдат прочетени, символа определен от *delim* бъде открит или е срещнат края на файла. Масива указван от *buf* ще бъде нулево – термилиран от *getline()*. Ако се срещне разграничителния символ във входния поток, той се извлича но не се слага във *buf*. Тази функция връща псевдоним за потока. Свързани функции са *get()* и *read()*.

good ()

```
#include <iostream>
bool good ( ) const;
```

Функцията *good()* е член на *ios*. Функцията *good()* връща *true* ако не се срещнат I/O грешки във свързания поток, в противен случай връща *false*. Свързани функции са *bad()*, *fail()*, *eof()*, *clear()* и *rdstate()*.

ignore ()

```
#include <iostream>
istream &ignore ( streamsize num = 1, int delim = EOF );
```

Функцията *ignore()* е член на *istream*. Вие може да използвате член-функцията *ignore()* да чете и отхвърля символи от входния поток. Тя чете и отхвърля символи докато или бъдат отхвърлени *num* символа (1 по – подразбиране) или се срещне символа определен от *delim* (EOF по – подразбиране). Ако се срещне разграничителния символ, той се премахва от входния поток. Функцията връща псевдоним за потока. Свързани функции са *get()* и *getline()*.

open ()

```
#include <fstream>
void fstream :: open ( const char *filename,
                      openmode mode = ios :: in | ios :: out );
void ifstream :: open ( const char *filename,
                       openmode mode = ios :: in );
void ofstream :: open ( const char *filename,
                       openmode mode = ios :: out | ios :: trunc );
```

Функцията *open ()* е член на *fstream*, *ifstream* и *ofstream*. Файл се свързва с поток чрез използване на функцията *open ()*. Тук, *filename* е името на файла което може да включва и пътя. Стойността на *mode* определя как да се отвори файла. Тя трябва да бъде една (или повече) от тези стойности:

<i>ios :: app</i>	<i>ios :: in</i>
<i>ios :: ate</i>	<i>ios :: out</i>
<i>ios :: binary</i>	<i>ios :: trunc</i>

Вие можете да комбинирате две или повече от тези стойности като ги оградите заедно във кръгли скоби. Включването на *ios :: app* предизвиква целия изход към файла да бъде добавен в края му. Тази стойност може да бъде използвана само с файлове със възможност за изход. Включването на *ios :: ate* предизвиква търсене към края на файла до съвпадение когато файла е отворен. Въпреки че *ios :: ate* предизвиква отместване към края на файла I/O операциите могат да се извършат навсякъде във файла. Стойността *ios :: binary* предизвиква файла да бъде отворен за двоични I/O операции. По – подразбиране файл се отваря в текстов режим. Стойността *ios :: in* определя че файла има възможност за вход. Стойността *ios :: out* определя че файла има възможност за изход. Обаче създаването на поток със използване на *ifstream* подразбира вход, а създаването на файл чрез използване на *ofstream* е по – подразбиране изход и отварянето на файл със *fstream* означава вход / изход. Във всички случай ако *open ()* пропадне потока ще бъде 0. По тази причина преди да използвате файл ще трябва да проверите да се уверите че отварящата операция е успешна. Свързани функции са *close ()*, *fstream ()*, *ifstream ()* и *ofstream ()*.

Програмен съвет

В библиотеката в стар стил, конструктора *fstream* не съдържа подразбираща се стойност за параметъра *mode*. Така той не отваря автоматично поток за входни и изходни операции. Така когато използвате старата библиотека за да отваряте поток за вход / изход и двете стойности *ios :: in* и *ios :: out* трябва да бъдат изрично определени. Запомнете го ако ви трябва обратна съвместимост.

peek ()

```
#include <iostream >
int peek ( ) ;
```

Функцията *peek ()* е член на *istream*. Функцията *peek ()* връща следващия символ в потока или EOF ако е достигнат края на файла. Ако не е така при всички случаи тя премахва символа от потока. Свързана функция е *get ()*.

precision ()

```
#include <iostream >
streamsize precision ( ) const ;
streamsize precision ( streamsize p ) ;
```

Функцията *precision ()* е член на *ios*. По – подразбиране се показват шест цифри точност когато се извежда стойност със плаваща точка. Обаче със използването на втората форма на *precision ()* ще можете да установите това

число на стойността определена в *p*. Връща се първоначалната стойност. Първата версия на *precision()* връща текущата стойност. Свързани функции са *width()* и *fill()*.

put ()

```
#include <iostream >
ostream &put ( char ch );
```

Функцията *put()* е член на *ostream*. Функцията *put()* записва *ch* във свързания изходен поток. Тя връща псевдоним за потока. Свързани функции са *write()* и *get()*.

putback ()

```
#include <iostream >
istream &putback ( char ch );
```

Функцията *putback()* е член на *istream*. Функцията *putback()* връща *ch* във свързания входен поток. Свързана функция е *peek()*.

rdstate ()

```
#include <iostream >
iosstate rdstate ( ) const ;
```

Функцията *rdstate()* е член на *ios*. Функцията *rdstate()* връща състоянието на свързания поток. Системата за I/O на C++ поддържа статус информация за резултата от всяка I/O операция отнасяща се до всеки активен поток. Текущото състояние на I/O системата се съдържа в обект от тип *iosstate* в който са дефинирани следните флагове:

Име	Значение
<i>goodbit</i>	Не са станали грешки
<i>eofbit</i>	Срещнат е края на файла
<i>failbit</i>	Станала е нефатална I/O грешка
<i>badbit</i>	Станала е фатална I/O грешка

Тези флагове са изброени във *ios*. *rdstate()* връща *goodbit* когато не е станала грешка, в противен случай се установява бита за грешка. Свързани функции са *eof()*, *good()*, *bad()*, *clear()* и *fail()*.

read ()

```
#include <iostream >
istream &read ( char *buf, streamsize num );
```

Функцията *read()* е член на *istream*. Функцията *read()* чете *num* байта от свързания входен поток и ги слага във буфер указван от *buf*. Ако е достигнат края на файла преди да са прочетени *num* символа, *read()* просто спира и буфера ще съдържа колкото символа е имало (Виж "*gcount()*"). *read()* връща псевдоним за потока. Свързани функции са *gcount()*, *get()*, *getline()* и *write()*.

seekg () и seekp ()

```
#include <iostream>
istream &seekg ( off_type offset , ios :: seekdir origin ) ;
istream &seekg ( pos_type position ) ;
ostream &seekp ( off_type offset , ios :: seekdir origin ) ;
ostream &seekp ( pos_type position ) ;
```

Функцията **seekg ()** е член на **istream** , а функцията **seekp ()** е член на **ostream** . В системата за I/O на C++ изпълнявате произволен достъп чрез използване на функциите **seekg ()** и **seekp ()** . Към този момент системата на C++ за I/O управлява два указателя свързани със файл . Единия е **get pointer** който определя къде във файла ще стане следващата входна операция . Другия е **put pointer** който определя къде във файла ще стане следващата изходна операция . Всеки път когато се извърши въвеждане или извеждане съответния указател се увеличава автоматично в последователност . Обаче чрез използване на **seekg ()** и **seekp ()** е възможно да достигнете файла по непоследователен начин . Двупараметровата версия на **seekg ()** премества **get pointer** на **offset** брой байтове от мястото определено от **origin** . Двупараметровата версия на **seekp ()** премества **put pointer** на **offset** брой байтове от мястото определено от **origin** . Параметърът **offset** е от тип **off_type** който е способен да съдържа най – голямата валидна стойност която **offset** може да има . Параметърът **origin** е от тип **seekdir** и е изброяване което има тези стойности :

ios :: beg	Отместване от началото
ios :: cur	Отместване от текущата позиция
ios :: end	Отместване от края

Еднопараметровите версии на **seekg ()** и **seekp ()** преместват файловите указатели на положението определено от **position** . Тази стойност трябва да бъде получена преди това чрез извикване на **tellg ()** или **tellp ()** съответно . **pos_type** е тип който е способен да съдържа най – голямата валидна стойност която може да има **position** . Тези функции връщат псевдоним за свързания поток . Свързани функции са **tellg ()** и **tellp ()** .

setf ()

```
#include <iostream>
fmtflags setf ( fmtflags flags ) ;
fmtflags setf ( fmtflags flags1 , fmtflags flags2 ) ;
```

Функцията **setf ()** е член на **ios** . **setf ()** вдига форматните флагове свързани с поток . Виж разглеждането на форматните флагове по – рано в тази секция . Първата версия на **setf ()** вдига форматните флагове определени от **flags** (всички други флагове са непроменени) . Например за вдигане на флага **showpos** за **cout** може да използвате този израз :

```
cout . setf ( ios :: showpos ) ;
```

Когато искате да установите повече от един флаг можете да оградите заедно във кръгли скоби стойностите на флаговете които искате да установите . Важно е да разберете че извикването на **setf ()** се отнася само за определения поток . Няма концепция за самостоятелно извикване на **setf ()** . Казано различно

няма концепция в C++ за глобално форматно състояние. Всеки поток поддържа собствената си форматна статус информация индивидуално. Втората версия на *setf()* въздейства само на флаговете които сте установили във *flags2*. Съответните флагове първо са пренастроени и след това вдигнати съгласно флаговете определени от *flags1*. Дори ако *flags1* съдържа друго множество флагове, само онези определени от *flags2* ще бъдат променени. И двете версии на *setf()* връщат предишните настройки на форматните флагове свързани със потока. Свързани функции са *unsetf()* и *flags()*.

sync_with_stdio ()

```
#include <iostream>
```

```
static bool sync_with_stdio ( bool sync = true );
```

Функцията *sync_with_stdio()* е член на *ios*. Извикването на *sync_with_stdio()* позволява стандартната C – базирана система за I/O да бъде безопасно използвана едновременно със класово – базираната система на C++ за I/O. За премахване на синхронизацията със *stdio*, подайте *false* на *sync_with_stdio()*. Връща се предишната настройка – *true* за синхронизиране и *false* ако няма синхронизация.

tellg () и tellp ()

```
#include <iostream>
```

```
pos_type tellg ( );
```

```
pos_type tellp ( );
```

Функцията *tellg()* е член на *istream*, а функцията *tellp()* е член на *ostream*. Системата за I/O на C++ управлява два указателя свързани със файл. Единия е **get pointer** който определя къде във файла ще стане следващата входна операция. Другия е **put pointer** който определя къде във файла ще стане следващата изходна операция. Всеки път когато се извърши въвеждане или извеждане съответния указател се увеличава автоматично и последователно. Вие можете да определите текущата позиция на **get pointer** използвайки *tellg()* и на **put pointer** използвайки *tellp()*. *pos_type* е тип който е способен да съдържа най – голямата стойност която всяка от функциите може да върне. Стойностите върнати от *tellg()* и *tellp()* могат да бъдат използвани като параметри съответно на *seekg()* и *seekp()*. Свързани функции са *seekg()* и *seekp()*.

unsetf ()

```
#include <iostream>
```

```
void unsetf ( fmtflags flags );
```

Функцията *unsetf()* е член на *ios*. Функцията *unsetf()* се използва да сваля един или повече форматни флага. Флаговете определени от *flags* се свалят (всички други флагове остават непроменени). Свързани функции са *setf()* и *flags()*.

width ()

```
#include <iostream>
```

```
streamsize width ( ) const ;
```

```
streamsize width ( streamsize w );
```

Функцията *width()* е член на *ios*. За да получите текущата ширина на полето използвайте първата форма на *width()*. Тази версия връща текущата ширина на полето. За да установите ширината на полето използвайте втората форма. Тук, *w* става новата ширина на полето, а се връща предишната стойност. Свързани функции са *precision()* и *fill()*.

write()

```
#include <iostream>
```

```
ostream &write( const char *buf, streamsize num );
```

Функцията *write()* е член на *ostream*. Функцията *write()* записва *num* байта в определения изходен поток от буфера указван чрез *buf*. Тя връща псевдоним за потока. Свързани функции са *read()* и *put()*.

ГЛАВА 14 – Стандартната библиотека с шаблони на C++

Едно от главните постижения които станаха през стандартизационния процес на C++ беше включването на стандартната библиотека с шаблони или **STL**. **STL** предоставя общо предназначени, шаблонни класове и функции които изпълняват множество популярни и общо използвани алгоритми и структури за данни. Например, тя включва поддръжка на вектори, списъци, опашки и стекове. Тя също дефинира и разнообразни начини за достъп до тях. Понеже **STL** е изградена от шаблонни класове, алгоритмите и структурите за данни могат да бъдат приложени към почти всеки тип данни. **STL** е голяма библиотека и не всички от нейните качества могат да бъдат пълно описани в тази книга. Също версията на **STL** описана тук е тази определена от стандартизационния комитет **ANSI / ISO**. Вашия компилатор може да предоставя по – различна версия на **STL** така че трябва да проверите неговата документация.

Преглед на контейнери, алгоритми и итератори

STL е съставена от три компонента: контейнери, алгоритм и итератори. Те работят заедно за да доставят основни решения за разнообразни програмни проблеми. Всеки е описан кратко тук.

Контейнери

Контейнерите са обекти които съдържат други обекти. Има няколко различни типа контейнери. Например класа *vector* дефинира динамичен масив, *queue* създава опашка и *list* изгражда линеен списък. В добавка към основните контейнери **STL** също дефинира и асоциативни контейнери които позволяват ефикасно получаване на стойности базирани на ключове. Например, *map* доставя достъп до стойности с уникални ключове. По този начин *map* съхранява двойк ключ / стойост и позволява стойността да бъде върната чрез подаване на нейния ключ. Всеки контейнерен клас дефинира множество от

функции които могат да бъдат приложени към контейнера . Например , *list* контейнера включва функции които вмъкват , изтриват и сливат елементи .

Алгоритми

Алгоритмите действат върху контейнерите . Те включват възможности за инициализиране , сортиране , претърсване и преобразуване на съдържанието на контейнерите . Много алгоритми действат в последователност , което е линеен списък от елементи в контейнера .

Итератори

Итераторите са обекти които са повече или по – малко указатели . Те дават възможност да имаме достъп до съдържанието на контейнер по много подобен начин при който използваме указател за достъп до масив . Има 5 типа итератори :

Итератор	Разрешен достъп
Произволен достъп	Съхранява и извлича стойности; елементите могат да бъдат достъпвани произволно
Двупосочен	Съхранява и извлича стойности; движение напред и назад
Прав	Съхранява и извлича стойности; движение само напред
Входен	Извлича , но не съхранява стойности ; движение само напред
Изходен	Съхранява , но не извлича стойности ; движение само напред

Най – общо итератор който има най – големи възможности за достъп може да бъде използван на мястото на такъв със по – малки възможности . Например правия итератор може да бъде използван на мястото на входния . Итераторите се манипулират като указатели . Може да се увеличават и намаляват . Може да се прилага оператора * към тях . Итераторите са декларираны чрез използване на типа *iterator* дефиниран за различни контейнери . Когато се отнася за различните типове контейнери общо , тази книга използва следните имена :

Име	Представя
BiIter	Двупосочен итератор
ForIter	Прав итератор
InIter	Входен итератор
OutIter	Изходен итератор
RandIter	Итератор с произволен достъп

STL също поддържа и обратни итератори . Обратните итератори са или двупосочни или с произволен достъп които се преместват в последователност в обратна посока . Така ако обратния итератор сочи края на поредицата , увеличавайки го ще предизвикаме той да сочи към един елемент преди края .

Алокатори

Всеки контейнер дефинира за себе си алокатор, който е обект от клас **allocator**. Алокаторите управляват заделянето на памет когато се създава нов контейнер.

Предикати и сравнителни функции

Някои от алгоритмите и контейнерите използват специален тип функции наречени предикати. Има две вариации на предикатите - унарни и бинарни. Унарния предикат приема един аргумент. Бинарният приема два аргумента. Тези функции връщат резултат **true** / **false**. Но точното условие което ги кара да върнат **true** или **false** се дефинира от вас. В следващите описания, когато се изисква унарна предикатна функция, тя ще се означава чрез типа **UnPred**. Когато се изисква бинарен предикат ще използваме типа **BinPred**. В бинарния предикат, аргументите са винаги в ред първи, втори отнасящи се за функцията която извиква предиката. И за двата предиката, аргументите ще съдържат стойности от типа на обектите които се съхраняват в контейнер. Някои алгоритми и класове използват специален тип бинарен предикат който сравнява два елемента. Сравняващите функции връщат **true** ако техният първи аргумент е по-малък от втория. Сравняващите функции ще ги представяме чрез типа **comp**.

Инструменти и функционални хедъри

В добавка към хедърите изисквани от различните класове на STL стандартната библиотека на C++ включва **<utility>** и **<functional>** хедъри, които доставят поддръжка за STL. Например в **<utility>** е дефиниран шаблонният клас **pair**, който може да съдържа двойка от стойности. Шаблонната функция **less()** декларирана във **<functional>** определя кога един обект е по-малък от друг. Шаблоните във **<functional>** ви позволяват да изградите обекти които дефинират **operator()**. Те са наречени функционални обекти и могат да бъдат използвани на мястото на указателите към функции на много места. Те са извън обсега на този бърз справочник за да ги описваме.

Програмен съвет

Контейнерите, алгоритмите и итераторите работят заедно. Най-добрият начин да разберете това е да видите пример. Следващата програма демонстрира контейнера **vector**. **vector** е подобен на масив. Обаче той има предимство че автоматично манипулира своите размери за съхраняване, нараствайки ако е необходимо. **vector** предоставя методи за да можете да определите размера му и да добавяте или премахвате елементи. Следващата програма демонстрира използването на класа **vector**:

```
// къс пример демонстриращ vector
#include <iostream>
#include <vector>
using namespace std;

int main ( )
{
```

```

vector<int> v; // създава vector с нулева дължина
int i;
// показва първоначалната дължина на v
cout << "size = " << v.size() << endl;
/* поставя стойности към края на v –
   vector ще нарасне ако е необходимо */
for (i = 0; i < 10; i++) v.push_back(i);
// показва текущата дължина на v
cout << "size now = " << v.size() << endl;
// можеме да достъпваме съдържанието на vector
// чрез индекс
for (i = 0; i < 10; i++) cout << v[i] << " ";
cout << endl;
// можеме да достъпваме първия и последния елемент
cout << "front = " << v.front() << endl;
cout << "back = " << v.back() << endl;
// достъп чрез итератор
vector<int>::iterator p = v.begin();
while (p != v.end()) {
    cout << *p << " ";
    p++;
}

return 0;
}

```

Изхода от тази програма е :

```

size = 0
size now = 10
0 1 2 3 4 5 6 7 8 9
front = 0
back = 9
0 1 2 3 4 5 6 7 8 9

```

В тази програма вектора е първоначално създаден с нулева дължина. Член функцията *push_back()* слага стойности в края на вектора, увеличавайки размера му ако е необходимо. Функцията *size()* показва размера на вектора. Вектора може да бъде индексирен като нормален масив. Също може да бъде достъпван чрез итератор. Функцията *begin()* връща итератор към началото на вектора. Функцията *end()* връща итератор към края на вектора. От друга страна: Забележете как е деклариран итератора *p*. Типа *iterator* е дефиниран за няколко контейнерни класа.

Контейнерни класове

Контейнерите дефинирани от STL са показани тук :

Контейнер	Описание	Изискван хедър
<i>bitset</i>	Множество от битове	< <i>bitset</i> >
<i>deque</i>	Опашка с два края	< <i>deque</i> >
<i>list</i>	Линеен списък	< <i>list</i> >

<i>map</i>	Съхранява двойки ключ / стойност в които на един ключ могат да отговарят две или повече стойности	< <i>map</i> >
<i>multimap</i>	Съхранява двойки ключ / стойност в които на един ключ могат да отговарят две или повече стойности	< <i>map</i> >
<i>multiset</i>	Множество в което всеки елемент не е необходимо да е уникален	< <i>set</i> >
<i>priority_queue</i>	Опашка с приоритети	< <i>queue</i> >
<i>queue</i>	Опашка	< <i>queue</i> >
<i>set</i>	Множество в което всеки елемент е уникален	< <i>set</i> >
<i>stack</i>	Стек	< <i>stack</i> >
<i>vector</i>	Динамичен масив	< <i>list</i> >

Класа ***string*** който управлява символните низове е също контейнер но е обсъден в Глава 15. Всеки от контейнерите е обобщен в следващите секции. Понеже контейнерите са изпълнени чрез шаблонни класове се използват различни типове съхранители на данни. В описанията, общия тип **T** представя типа на данните съхранявани от контейнера, а **Size** представя някакъв целочислен тип. Понеже имената на съхранителните типове в шаблонен клас са произволни, контейнерните класове декларират ***typedef*** версии на тези типове. Това прави имената на типовете конкретни. Някой от най – общите ***typedef*** имена са показани тук:

size_type	Някакъв целочислен тип еквивалентен на <i>size_t</i>
reference	Псевдоним на елемент
const_reference	const псевдоним на елемент
iterator	Итератор
const_iterator	const итератор
reverse_iterator	Обратен итератор
const_reverse_iterator	<i>const</i> обратен итератор
value_type	Тип на стойността съхранявана в контейнер
allocator_type	Типа на алокатора
key_type	Типа на ключа
key_compare	Типа на функцията която сравнява два ключа
value_compare	Типа на функцията която сравнява две стойности

bitset

Класа **bitset** поддържа операции върху множество от битове. Неговата шаблонна спецификация е:

```
template <size_t N> class bitset
```

Тук, N определя дължината на **bitset** в битове. Той има следните конструктори:

```
bitset ();
bitset ( unsigned long bits );
explicit bitset ( const string &s , size_t i = 0 , size_t num = -1 );
```

Първата форма конструира празен **bitset**. Втората форма конструира **bitset** който има собствено множество битове съгласно тези определени в **bits**.

Третата форма конструира **bitset** чрез използване на низа s , започвайки от i . Низа трябва да съдържа само единици или нули. Само num или $s.size() - i$ стойности се използват, което е по-малко. Извеждащите оператори $<<$ и $>>$ се дефинират за **bitset**. **bitset** съдържа следните член-функции:

Член	Описание
<code>bool any () const ;</code>	Връща true ако всеки бит в извикания bitset е 1; в противен случай връща 0
<code>size_type count () const ;</code> <code>bitset <N> &flip () ;</code>	Връща броя на битовете с 1 Инвертира състоянието на всички битове в извикания bitset и връща *this
<code>bitset <N> &flip (size_t i) ;</code>	Инвертира бита на позиция i в извикания bitset и връща *this
<code>bool none () const ;</code>	Връща true ако няма установени битове в извикания bitset
<code>bool operator != (const bitset <N> &op2) const ;</code>	Връща true ако извикания bitset се различава от този определен в дясната страна на оператора op2
<code>bool operator == (const bitset <N> &op2) const ;</code>	Връща true ако извикания bitset е еднакъв със този определен в дясната страна на оператора op2
<code>bitset <N></code> <code>&operator &= (const bitset <N> &op2);</code>	Операция AND върху всеки бит на извикания bitset със съответния бит в op2 и поставя резултата в извикания bitset ; връща *this
<code>bitset <N></code> <code>&operator ^= (const bitset <N> &op2);</code>	Операция XOR върху всеки бит на извикания bitset със съответния бит в op2 и поставя резултата в извикания bitset ; връща *this
<code>bitset <N></code> <code>&operator = (const bitset <N> &op2);</code>	Операция OR върху всеки бит на извикания bitset със съответния бит в op2 и поставя резултата в извикания bitset ; връща *this
<code>bitset <N> &operator ~ = () const ;</code>	Инвертира състоянието на всички битове в извикания

bitset <N> &operator <<= (*size_t num*);

bitset <N> &operator >>= (*size_t num*);

reference operator [] (*size_type i*);

bitset <N> &reset ();

bitset <N> &reset (*size_t i*);

bitset <N> &set ();

bitset <N> &set (*size_t i*, *int val = 1*);

size_t size () *const* ;

bool test (*size_t i*);

string to_string () *const* ;

unsigned long to_ulong () *const* ;

bitset и връща **this*

Премества наляво всеки бит в извикания *bitset* на *num* позиции и поставя резултата в извикания *bitset*; връща **this*

Премества надясно всеки бит в извикания *bitset* на *num* позиции и поставя резултата в извикания *bitset*; връща **this*

Връща псевдоним за бит *i* в извикания *bitset*

Изчиства всички битове в извикания *bitset* и връща **this*

Изчиства бита на позиция *i* в извикания *bitset* и връща **this*

Установява всички битове в извикания *bitset* и връща **this*

Установява бита на позиция *i* на стойността определена от *val* в извикания *bitset* и връща **this*
Всяка ненулева стойност за *val* е приета да бъде *1*

Връща броя на битове който може да съдържа дадения *bitset*

Връща състоянието на бита на позиция *i*

Връща низ който съдържа представяне на битовия шаблон в извикания *bitset*

Конвертира извикания *bitset* в **unsigned long integer**

deque

Класа *deque* поддържа опашка със два края. Неговата шаблонна спецификация е:

```
template < class T, class Allocator = allocator < T >>
    class deque
```

Тук *T* е типа данни съхранявани в *deque*. Той има следните конструктори:

```
explicit deque ( const Allocator &a = Allocator ( ) );
explicit deque ( size_type num, const T &val = T ( ),
                const Allocator &a = Allocator ( ) );
deque ( const deque < T, Allocator > &ob );
template < class InIter > deque ( InIter start, InIter end,
                                const Allocator &a = Allocator ( ) );
```

Първата форма конструира празна *deque*. Втората форма конструира *deque* която има *num* елемента със стойност *val*. Третата форма изгражда *deque* която има еднакви елементи със *ob*. Четвъртата форма изгражда опашка която съдържа елементите в интервала определен от *start* и *end*. Следващите сравнителни оператори са дефинирани за *deque*: *==*, *<*, *<=*, *!=*, *>* и *>=*. *deque* съдържа следните член функции:

Член

```
template < class InIter >
    void assign ( InIter start, InIter end );
```

Описание
 Присвоява на *deque* поредицата дефинирана от *start* и *end*

```

template < class Size , class T >
    void assign ( Size num ,
        const T &val = T () ) ;
reference at ( size_type i ) ;
const_reference at ( size_type i ) const ;
reference back () ;
const_reference back () const ;
iterator begin () ;
const_iterator begin () const ;
void clear () ;

bool empty () const ;

iterator end () ;
const_iterator end () const ;
iterator erase ( iterator i ) ;

iterator erase ( iterator start , iterator end ) ;

reference front () ;
const_reference front () const ;
allocator_type get_allocator () const ;
iterator insert ( iterator i ,
    const T &val = T () ) ;

void insert ( iterator i , size_type num ,
    const T &val ) ;

template < class InIter >
    void insert ( iterator i ,
        InIter start , InIter end ) ;

size_type max_size () const ;

reference operator [] ( size_type i ) const ;
const_reference
    operator [] ( size_type i ) const ;
void pop_back () ;

void pop_front () ;

void push_back ( const T &val ) ;

void push_front ( const T &val ) ;

reverse_iterator rbegin () ;
const_reverse_iterator rbegin () const ;
reverse_iterator rend () ;
const_reverse_iterator rend () const ;
void resize ( size_type num ,
    T val = T () ) ;

```

Присвоява на *deque num*
елемента със стойност *val*

Връща псевдоним за елемента
определен чрез *i*

Връща псевдоним за последния
елемент в *deque*

Връща итератор за първия
елемент в *deque*

Премахва всички елементи
от *deque*

Връща true ако извиканата deque
е празна и *false* в противен случай

Връща итератор към края
на *deque*

Премахва елемента указван от *i*
връща итератор към елемента
след премахнатия

Премахва елементите в
интервала *start – end*, връща итератор
към елемента след последния
премахнат елемент

Връща псевдоним към първия
елемент на *deque*

Връща алокатор за *deque*

Вмъква *val* непосредствено

преди елемента определен от *i*

Връща се итератор към елемента

Вмъква *num* копия от *val*

непосредствено преди елемента
определен от *i*

Вмъква поредицата определена
от *start – end* непосредствено
преди елемента определен
от *i*

Връща максималния брой
елементи които може да
съдържа *deque*

Връща псевдоним за *i* - тия
елемент

Премахва последния елемент
в *deque*

Премахва първия елемент
в *deque*

Добавя елемент със стойност
val към края на *deque*

Добавя елемент със стойност
val към началото на *deque*

Връща обратен итератор към
края на *deque*

Връща обратен итератор към
началото на *deque*

Променя размера на *deque* към
този определен от *num*. Ако
deque трябва да бъде удължена,

```
size_type size () const ;
```

```
void swap ( deque < T , Allocator > &ob ) ;
```

тогава елементите със стойността
определена от *val* се добавят
към края
Връща текущия брой елементи
в *deque*
Разменя елементите съхранявани
в извикваната *deque* със тези
в *ob*

list

Класа *list* поддържа списък. Неговата шаблонна спецификация е :

```
template < class T , class Allocator = allocator < T >>
class list
```

Тук *T* е типа данни съхранявани в списъка. Той има следните конструктори :

```
explicit list ( const Allocator &a = Allocator () ) ;
explicit list ( size_type num , const T &val = T () ,
               const Allocator &a = Allocator () ) ;
list ( const list < T , Allocator > &ob ) ;
template < class InIter > list ( InIter start , InIter end ,
                               const Allocator &a = Allocator () ) ;
```

Първата форма конструира празен списък. Втората форма конструира списък който има *num* елемента със стойност *val*. Третата форма изгражда списък който има еднакви елементи със *ob*. Четвъртата форма изгражда списък който съдържа елементите в интервала определен от *start* и *end*. Следващите сравнителни ператори са дефинирани за *list* : *==* , *<* , *<=* , *!=* , *>* и *>=*. *list* съдържа следните член функции :

Член

```
template < class InIter >
void assign ( InIter start , InIter end ) ;
template < class Size , class T >
void assign ( Size num ,
             const T &val = T () ) ;
reference back () ;
const_reference back () const ;
iterator begin () ;
const_iterator begin () const ;
void clear () ;

bool empty () const ;

iterator end () ;
const_iterator end () const ;
iterator erase ( iterator i ) ;

iterator erase ( iterator start , iterator end ) ;

reference front () ;
const_reference front () const ;
allocator_type get_allocator () const ;
iterator insert ( iterator i ,
```

Описание

Присвоява на списъка
поредицата дефинирана от *start* и *end*
Присвоява на списъка *num*
елемента със стойност *val*

Връща псевдоним за последния
елемент в списъка
Връща итератор за първия
елемент в списъка
Премахва всички елементи
от списъка
Връща *true* ако извикания списък
е празен и *false* в противен случай
Връща итератор към края
на списъка
Премахва елемента указван от *i*
връща итератор към елемента
след премахнатия
Премахва елементите в
интервала *start – end*, връща итератор
към елемента след последния
премахнат елемент
Връща псевдоним към първия
елемент на списъка
Връща алокатор за списъка
Вмъква *val* непосредствено

```

        const T &val = T();

void insert ( iterator i, size_type num,
             const T &val );

template < class InIter >
void insert ( iterator i,
             InIter start, InIter end );

size_type max_size () const;

void merge ( list < T, Allocator > &ob );
template < class Comp >
void merge ( list < T, Allocator > &ob
            Comp cmpfn );

void pop_back ();

void pop_front ();

void push_back ( const T &val );

void push_front ( const T &val );

reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
void remove ( const T &val );

template < class UnPred >
void remove_if ( UnPred pr );
const T &val = T();
reverse_iterator rend ();
const_reverse_iterator rend () const;
void resize ( size_type num,
             T val = T() );

void reverse ();
size_type size () const;

void sort ();
template < class Comp >
void sort ( Comp cmpfn );

void splice ( iterator i,
             list < T, Allocator > &ob );

```

преди елемента определен от *i*
 Връща се итератор към елемента
 Вмъква *num* копия от *val*
 непосредствено преди елемента
 определен от *i*
 Вмъква поредицата определена
 от *start – end* непосредствено
 преди елемента определен
 от *i*
 Връща максималния брой
 елементи които може да
 съдържа списъка
 Слива подредения списък
 съдържа в *ob* със подредения
 извикван списък. Резултатния
 списък е подреден. След
 сливането, списък съдържа
 от *ob* е празен. Във втората
 форма сравнителната функция
 може да бъде определена така
 че да определя кога един
 елемент е по – малък от друг
 Премахва последния елемент
 в списъка
 Премахва първия елемент
 в списъка
 Добавя елемент със стойност
val към края на списъка
 Добавя елемент със стойност
val към началото на списъка
 Връща обратен итератор към
 края на списъка
 Премахва елементите със
 стойност *val* от списъка
 Премахва елементите за които
 унарния предикат *pr* е *true*
 Връща обратен итератор към
 началото на списъка
 Променя размера на списъка към
 този определен от *num*. Ако
 списъка трябва да бъде удължен,
 тогава елементите със стойността
 определена от *val* се добавят
 към края
 Обръща извиквания списък
 Връща текущия брой елементи
 в списъка
 Сортира списъка. Втората форма
 сортира списъка използвайки
 сравнителната функция *fn* за
 определяне кога един елемент е
 по – малък от друг
 Съдържанието на *ob* се вмъква
 в извиквания списък на
 позицията определена от *i*. След

```

void splice ( iterator i ,
              list < T , Allocator > &ob ,
              iterator el ) ;

void splice ( iterator i ,
              list < T , Allocator > &ob ,
              iterator start , iterator end ) ;

void swap ( list < T , Allocator > &ob ) ;

void unique ( ) ;
template < class BinPred >
void unique ( BinPred pr ) ;

```

операцията *ob* е празен
 Елемента указван от *el* е
 премахнат от списъка *ob* и
 съхранен в извиквания списък на
 позицията определена от *i*.
 Интервала определен от *start* и
end е премахнат от *ob* и
 съхранен в извиквания списък
 започвайки от позицията определена от
i
 Разменя елементите съхранявани
 в извиквания списък със тези
 в *ob*
 Премахва дублиращите се
 елементи от извиквания списък.
 Втората форма използва *pr* за да
 определи уникалността

map

Класа *map* поддържа асоциативен контейнер в който уникални ключове се съпоставят на стойности. Неговата шаблонна спецификация е:

```

template < class Key , class T , class Comp = less < Key > ,
          class Allocator = allocator < T >> class map

```

Тук, *Key* е типа данни на ключовете, *T* е типа данни на стойностите които ще се съхраняват (съпоставят) и *Comp* е функция която сравнява два ключа. Той има следните конструктори:

```

explicit map ( const Comp &cmpfn = Comp ( ) ,
               const Allocator &a = Allocator ( ) ) ;
map ( const map < Key , T , Comp , Allocator > &ob ) ;
template < class InIter > map ( InIter start , InIter end ,
                               const Comp &cmpfn = Comp ( ) ,
                               const Allocator &a = Allocator ( ) ) ;

```

Първата форма конструира празна карта. Втората форма изгражда карта която съдържа еднакви елементи със *ob*. Третата форма изгражда карта която съдържа елементите в интервала определен от *start* и *end*. Функцията определена чрез *cmpfn*, ако е представена, определя нареждането на картата. Следващите сравнителни оператори са дефинирани за *map*: *==*, *<*, *<=*, *!=*, *>* и *>=*.

Тук са показани член-функциите на *map*. В описанията, *key_type* е типа на ключа, а *value_type* представя *pair < Key , T >*:

```

Член
iterator begin ( ) ;
const_iterator begin ( ) const ;
void clear ( ) ;

```

```

size_type count ( const key_type &k )
const ;
bool empty ( ) const ;

```

```

iterator end ( ) ;
const_iterator end ( ) const ;
pair < iterator , iterator >

```

Описание

Връща итератор за първия
 елемент в картата
 Премахва всички елементи
 от картата
 Връща броя на срещанията на
k в картата (1 или 0)
 Връща *true* ако извиканата карта
 е празна и *false* в противен случай
 Връща итератор към края
 на картата
 Връща двойка итератори които

```

    equal_range ( const key_type &k ) ;
pair < const_iterator , const_iterator >
    equal_range ( const key_type &k ) const ;
void erase ( iterator i ) ;
void erase ( iterator start , iterator end ) ;

size_type erase ( const key_type &k ) ;

iterator find ( const key_type &k ) ;
const_iterator find ( const key_type &k )
    const ;

allocator_type get_allocator ( ) const ;
iterator insert ( iterator i ,
    const value_type &val ) ;

template < class InIter >
    void insert ( InIter start , InIter end ) ;
pair < iterator , bool >
    insert ( const value_type &val ) ;

key_compare key_comp ( ) const ;

iterator lower_bound ( const key_type &k ) ;
const_iterator
    lower_bound ( const key_type &k ) const ;
size_type max_size ( ) const ;

reference operator[] ( const key_type &i ) ;

reverse_iterator rbegin ( ) ;
const_reverse_iterator rbegin ( ) const ;
reverse_iterator rend ( ) ;
const_reverse_iterator rend ( ) const ;
size_type size ( ) const ;

void swap ( map < Key , T , Comp ,
    Allocator > &ob ) ;

iterator upper_bound ( const key_type &k ) ;
const_iterator
    upper_bound ( const key_type &k ) const ;
value_compare value_comp ( ) const ;

```

указват първия и последния
елемент в картата която съдържа
определен ключ
Премахва елемента указван от *i*
Премахва елементите в
интервала *start – end*
Премахва от картата елементите
които имат ключове със стойност *k*
Връща итератор към определен
ключ . Ако ключа не е открит се
връща итератор към края на
картата
Връща алокатор за картата
Вмъква *val* на или след
елемента определен от *i* .
Връща се итератор към елемента
Вмъква интервал от елементи

Вмъква *val* в извикваната карта .
Връща се итератор към елемента
Елемента се вмъква ако още не
е съществувал . Ако елемента се
вмъкне се връща двойка
< *iterator* , *true* > . В противен случай се
връща
< *iterator* , *false* >
Връща функция обект която
сравнява ключове
Връща итератор към първия
елемент в картата със ключ
равен или по – голям от *k*
Връща максималния брой
елементи които може да
съдържа картата
Връща псевдоним за елемента
определен от *i* . Ако този
елемент не съществува , той
се вмъква
Връща обратен итератор към
края на картата
Връща обратен итератор към
началото на картата
Връща текущия брой елементи
в картата
Разменя елементите съхранявани
в извикваната карта със тези
в *ob*
Връща итератор към първия
елемент в картата със ключ
по – голям от *k*
Връща функция обект която
сравнява стойности

multimap

Класа **multimap** поддържа асоциативен контейнер в който е възможно еднакви ключове да се съпоставят на стойности. Неговата шаблонна спецификация е:

```
template < class Key , class T , class Comp = less < Key > ,
          class Allocator = allocator < T >> class multimap
```

Тук, **Key** е типа данни на ключовете, **T** е типа данни на стойностите които ще се съхраняват (съпоставят) и **Comp** е функция която сравнява два ключа. Той има следните конструктори:

```
explicit multimap ( const Comp &cmpfn = Comp ( ) ,
                  const Allocator &a = Allocator ( ) ) ;
multimap ( const multimap < Key , T , Comp , Allocator > &ob )
;
template < class InIter > multimap ( InIter start , InIter end ,
                                   const Comp &cmpfn = Comp ( ) ,
                                   const Allocator &a = Allocator ( ) ) ;
```

Първата форма конструира празен **multimap**. Втората форма изгражда **multimap** която съдържа еднакви елементи със **ob**. Третата форма изгражда **multimap** която съдържа елементите в интервала определен от **start** и **end**. Функцията определена чрез **cmpfn**, ако е представена, определя нареждането на **multimap**. Следващите сравнителни оператори са дефинирани за **multimap**: **==**, **<**, **<=**, **!=**, **>** и **>=**.

Тук са показани член-функциите на **multimap**. В описанията, **key_type** е типа на ключа, **T** е стойността, а **value_type** представя **pair < Key, T >**:

Член	Описание
iterator begin () ;	Връща итератор за първия елемент в multimap
const_iterator begin () const ;	Премахва всички елементи от multimap
void clear () ;	Връща броя на срещанията на k в multimap
size_type count (const key_type &k) const ;	Връща true ако извикания multimap е празен и false в противен случай
bool empty () const ;	Връща итератор към края на multimap
iterator end () ;	Връща двойка итератори които указват първия и последния елемент в multimap който съдържа определен ключ
const_iterator end () const ;	Премахва елемента указван от i
pair < iterator , iterator > equal_range (const key_type &k) ;	Премахва елементите в интервала start – end
pair < const_iterator , const_iterator > equal_range (const key_type &k) const ;	Премахва от multimap елементите които имат ключове със стойност k
void erase (iterator i) ;	Връща итератор към определен ключ. Ако ключа не е открит се връща итератор към края на multimap
void erase (iterator start , iterator end) ;	Връща алокатор за multimap
size_type erase (const key_type &k) ;	Вмъква val на или след
iterator find (const key_type &k) ;	
const_iterator find (const key_type &k) const ;	
allocator_type get_allocator () const ;	
iterator insert (iterator i ,	

```

        const value_type &val );

template < class InIter >
    void insert ( InIter start , InIter end );
iterator insert ( const value_type &val );

key_compare key_comp ( ) const;

iterator lower_bound ( const key_type &k );
const_iterator
    lower_bound ( const key_type &k ) const;
size_type max_size ( ) const;

reverse_iterator rbegin ( );
const_reverse_iterator rbegin ( ) const;
reverse_iterator rend ( );
const_reverse_iterator rend ( ) const;
size_type size ( ) const;

void swap ( multimap < Key , T , Comp ,
            Allocator > &ob );

iterator upper_bound ( const key_type &k );
const_iterator
    upper_bound ( const key_type &k ) const;
value_compare value_comp ( ) const;

```

елемента определен от *i* .
Връща се итератор към елемента
Вмъква интервал от елементи

Вмъква *val* в извиквания
multimap
Връща функция обект която
сравнява ключове
Връща итератор към първия
елемент в **multimap** със ключ
равен или по – голям от *k*
Връща максималния брой
елементи които може да
съдържа **multimap**
Връща обратен итератор към
края на **multimap**
Връща обратен итератор към
началото на **multimap**
Връща текущия брой елементи
в **multimap**
Разменя елементите съхранявани
в извиквания **multimap** със тези
в *ob*
Връща итератор към първия
елемент в **multimap** със ключ
по – голям от *k*
Връща функция обект която
сравнява стойности

multiset

Класа **multiset** поддържа множество в което е възможно еднакви ключове да се съпоставят на стойности . Неговата шаблонна спецификация е :

```

template < class Key , class Comp = less < Key > ,
class Allocator = allocator < Key >> class multiset

```

Тук , *Key* е типа данни на ключовете , а *Comp* е функция която сравнява два ключа . Той има следните конструктори :

```

explicit multiset ( const Comp &cmpfn = Comp ( ) ,
                    const Allocator &a = Allocator ( ) ) ;
multiset ( const multiset < Key , Comp , Allocator > &ob ) ;
template < class InIter > multiset ( InIter start , InIter end ,
                                    const Comp &cmpfn = Comp ( ) ,
                                    const Allocator &a = Allocator ( ) ) ;

```

Първата форма конструира празен **multiset** . Втората форма изгражда **multiset** който съдържа еднакви елементи със *ob* . Третата форма изгражда **multiset** който съдържа елементите в интервала определен от *start* и *end* . Функцията определена чрез *cmpfn* , ако е представена , определя нареждането на множеството . Следващите сравнителни оператори са дефинирани за **multiset** :
== , < , <= , != , > и >=

Тук са показани член – функциите на **multiset** . В описанията , и двата *key_type* и *value_type* са *typedef* за *Key* :

Член	Описание
<i>iterator begin () ;</i>	Връща итератор за първия

```

const_iterator begin () const ;
void clear () ;

size_type count ( const key_type &k )
    const ;
bool empty () const ;

iterator end () ;
const_iterator end () const ;
pair < iterator , iterator >
    equal_range ( const key_type &k ) const ;

void erase ( iterator i ) ;
void erase ( iterator start , iterator end ) ;

size_type erase ( const key_type &k ) ;

iterator find ( const key_type &k ) const ;

allocator_type get_allocator () const ;
iterator insert ( iterator i ,
    const value_type &val ) ;

template < class InIter >
    void insert ( InIter start , InIter end ) ;
iterator insert ( const value_type &val ) ;

key_compare key_comp () const ;

iterator lower_bound ( const key_type &k )
    const ;

size_type max_size () const ;

reverse_iterator rbegin () ;
const_reverse_iterator rbegin () const ;
reverse_iterator rend () ;
const_reverse_iterator rend () const ;
size_type size () const ;

void swap ( multiset < Key , Comp ,
    Allocator > &ob ) ;

iterator upper_bound ( const key_type &k )
    const ;

value_compare value_comp () const ;

```

елемент в *multiset*

Премахва всички елементи
от *multiset*

Връща броя на срещанията на
k в *multiset*

Връща *true* ако извикания
multiset е празен и *false* в противен
случай

Връща итератор към края
на *multiset*

Връща двойка итератори които
указват първия и последния

елемент в *multiset* който
съдържа определен ключ

Премахва елемента указван от *i*

Премахва елементите в
интервала *start – end*

Премахва от *multiset* елементите
които имат ключове със стойност *k*

Връща итератор към определен
ключ. Ако ключа не е открит се

връща итератор към края на

multiset

Връща алокатор за *multiset*

Вмъква *val* на или след
елемента определен от *i*.

Връща се итератор към елемента

Вмъква интервал от елементи

Вмъква *val* в извиквания

multiset

Връща се итератор към елемента

Връща функция обект която
сравнява ключове

Връща итератор към първия

елемент в *multiset* със ключ

равен или по – голям от *k*

Връща максималния брой

елементи които може да

съдържа *multiset*

Връща обратен итератор към
края на *multiset*

Връща обратен итератор към

началото на *multiset*

Връща текущия брой елементи

в *multiset*

Разменя елементите съхранявани
в извиквания *multiset* със тези

в *ob*

Връща итератор към първия

елемент в *multiset* със ключ

по – голям от *k*

Връща функция обект която

сравнява стойности

queue

Класа *queue* поддържа опашка със един край. Неговата шаблонна спецификация е:

```
template < class T, class Container = deque < T >>
    class queue
```

Тук *T* е типа данни които се съхраняват, а *Container* е типа контейнер използван да съдържа опашката. Той има следния конструктор:

```
explicit queue ( const Container &cnt = Container () );
```

Конструктора *queue()* създава празна опашка. По – подразбиране той използва *deque* като контейнер, но *queue* може само да бъде достъпвана по **FIFO** начин. Също можете да използвате *list* като контейнер за опашка. Контейнера се съдържа в защитен обект наречен с от тип *Container*. Следващите сравнителни оператори са дефинирани за *queue*: $=$, $<$, $<=$, $!=$, $>$ и $>=$. *queue* съдържа следните член – функции:

Член	Описание
<i>value_type &back();</i>	Връща псевдоним за последния елемент в опашката
<i>const value_type &back() const;</i>	Връща <i>true</i> ако извиканата опашка е празна и <i>false</i> в противен случай
<i>bool empty() const;</i>	
<i>value_type &front();</i>	Връща псевдоним за първия елемент в опашката
<i>const value_type &front() const;</i>	Премахва първия елемент в опашката
<i>void pop();</i>	Добавя елемент със стойност <i>val</i> към края на опашката
<i>void push (const T &val);</i>	Връща текущия брой елементи в опашката
<i>size_type size() const;</i>	

priority_queue

Класа *priority_queue* поддържа приоритетна опашка със един край. Неговата шаблонна спецификация е:

```
template < class T, class Container = vector < T >,
    class Comp = less < Container :: value_type >>
    class priority_queue
```

Тук *T* е типа на съхраняваните данни. *Container* е типа контейнер използван да съдържа опашката, а *Comp* определя сравнителната функция която определя кога един член на приоритетната опашка е със по – нисък приоритет от друг. Той има следните конструктори:

```
explicit priority_queue ( const Comp &cmpfn = Comp (),
    Container &cnt = Container () );
template < class InIter > priority_queue ( InIter start, InIter
    end, const Comp &cmpfn = Comp (),
    Container &cnt = Container () );
```

Първия *priority_queue()* конструктор създава празна приоритетна опашка. Втория изгражда опашка която съдържа елементите в интервала определен от *start* и *end*. По – подразбиране той използва *vector* като контейнер. Може да използвате *deque* като контейнер за опашка. Контейнера се съдържа в защитен обект наречен с от тип *Container*. *priority_queue* съдържа следните

член – функции :

Член
bool empty () const ;

void pop () ;
void push (const T &val) ;

size_type size () const ;

value_type &top () ;
const_value_type &top () const ;

Описание
 Връща *true* ако извиканата приоритетна опашка е празна и *false* в противен случай
 Премахва първия елемент в приоритетната опашка
 Добавя елемент към приоритетната опашка
 Връща текущия брой елементи в приоритетната опашка
 Връща псевдоним за елемента с най – висок приоритет . Елемента не се премахва

set

Класа *set* поддържа множество в което е възможно уникални ключове да се съпоставят на стойности . Неговата шаблонна спецификация е :

template < class Key , class Comp = less < Key > ,
class Allocator = allocator < Key >> class set

Тук , *Key* е типа данни на ключовете , а *Comp* е функция която сравнява два ключа .Той има следните конструктори :

explicit set (const Comp &cmpfn = Comp () ,
const Allocator &a = Allocator ()) ;
set (const multiset < Key , Comp , Allocator > &ob) ;
template < class InIter > set (InIter start , InIter end ,
const Comp &cmpfn = Comp () ,
const Allocator &a = Allocator ()) ;

Първата форма конструира празно множество .Втората форма изгражда множество което съдържа еднакви елементи със *ob* .Третата форма и згражда множество което съдържа елементите в интервала определен от *start* и *end* . Функцията определена чрез *cmpfn* , ако е представена , определя нареждането на множеството . Следващите сравнителни оператори са дефинирани за *set* :
 == , < , <= , != , > и >=

Тук са показани член – функциите на *set* .В описанията , и двете *key_type* и *value_type* са *typedef* за *Key* :

Член
iterator begin () ;
const_iterator begin () const ;
void clear () ;

size_type count (const key_type &k)
const ;
bool empty () const ;

iterator end () ;
const_iterator end () const ;
pair < iterator , iterator >
equal_range (const key_type &k) const ;

Описание
 Връща итератор за първия елемент в множеството
 Премахва всички елементи от множеството
 Връща броя на срещанията на *k* в множеството
 Връща *true* ако извиканото множество е празно и *false* в противен случай
 Връща итератор към края на списъка
 Връща двойка итератори които указват първия и последния елемент в множеството което съдържа определен ключ

```
void erase ( iterator i );
void erase ( iterator start , iterator end );
```

```
size_type erase ( const key_type &k );
```

```
iterator find ( const key_type &k ) const ;
```

```
allocator_type get_allocator () const ;
iterator insert ( iterator i ,
                 const value_type &val ) ;
```

```
template < class InIter >
void insert ( InIter start , InIter end ) ;
```

```
pair < iterator , bool >
insert ( const value_type &val ) ;
```

```
key_compare key_comp () const ;
```

```
iterator lower_bound ( const key_type &k )
const ;
```

```
size_type max_size () const ;
```

```
reverse_iterator rbegin () ;
const_reverse_iterator rbegin () const ;
reverse_iterator rend () ;
const_reverse_iterator rend () const ;
size_type size () const ;
```

```
void swap ( multiset < Key , Comp ,
            Allocator > &ob ) ;
```

```
iterator upper_bound ( const key_type &k )
const ;
```

```
value_compare value_comp () const ;
```

Премахва елемента указван от *i*
Премахва елементите в интервала *start – end*
Премахва от множеството елементите които имат ключове със стойност *k*.
Връща се броя на премахнатите елементи
Връща итератор към определен ключ. Ако ключа не е открит се връща итератор към края на множеството
Връща алокатор за множеството
Вмъква *val* на или след елемента определен от *i*.
Дублиращите се елементи не се вмъкват. Връща се итератор към елемента
Вмъква интервал от елементи.
Дублиращите се елементи не се вмъкват
Вмъква *val* в извикваното множество. Връща се итератор към елемента. Елемента се вмъква ако все още не е съществувал. Ако елемента се вмъкне се връща двойка *< iterator , true >*. В противен случай се връща *< iterator , false >*
Връща функция обект която сравнява ключове
Връща итератор към първия елемент в множеството със ключ равен или по – голям от *k*
Връща максималния брой елементи които може да съдържа множеството
Връща обратен итератор към края на множеството
Връща обратен итератор към началото на множеството
Връща текущия брой елементи в множеството
Разменя елементите съхранявани в извикваното множество със тези в *ob*
Връща итератор към първия елемент в множеството със ключ по – голям от *k*
Връща функция обект която сравнява стойности

stack

Класа *stack* поддържа стек. Неговата шаблонна спецификация е:

```
template < class T , class Container = deque < T >>
```

class stack

Тук **T** е типа на съхраняваните данни, а **Container** е типа контейнер използван да съдържа опашката. Той има следния конструктор:

explicit stack (const Container &cnt = Container());

Конструктора **stack()** създава празен стек. По подразбиране той използва **deque** като контейнер, но **stack** може само да бъде достъпван по **LIFO** начин. Контейнера се съдържа в защитен обект наречен с от тип **Container**.

Следващите сравнителни оператори са дефинирани за **stack**: **==**, **<**, **<=**, **!=**, **>** и **>=**.

stack съдържа следните член – функции:

Член	Описание
bool empty() const;	Връща true ако извикания стек е празен и false в противен случай.
void pop();	Премахва върха на стека който технически е последния елемент в контейнера.
void push(const T &val);	Слага елемент към края на стека. Последния елемент в контейнера представлява върха на стека.
size_type size() const;	Връща текущия брой елементи в стека.
value_type &top(); const_value_type &top() const;	Връща псевдоним за върха на стека, който е последния елемент в контейнера. Елемента не се премахва.

vector

Класа **vector** поддържа динамичен масив. Неговата шаблонна спецификация е:

template < class T, class Allocator = allocator < T >>

class vector

Тук **T** е типа на съхраняваните данни, а **Allocator** определя алокатора. Той има следните конструктори:

explicit vector (const Allocator &a = Allocator());
explicit vector (size_type num, const T &val = T(),
const Allocator &a = Allocator());
vector (const vector < T, Allocator > &ob);
template < class InIter > vector (InIter start, InIter end,
const Allocator &a = Allocator());

Първата форма конструира празен вектор. Втората форма конструира вектор който има **num** елемента със стойност **val**. Третата форма изгражда вектор който има еднакви елементи със **ob**. Четвъртата форма изгражда вектор който съдържа елементите в интервала определен от **start** и **end**. Следващите сравнителни оператори са дефинирани за **vector**: **==**, **<**, **<=**, **!=**, **>** и **>=**.

vector съдържа следните член функции:

Член	Описание
template < class InIter > void assign (InIter start, InIter end);	Присвоява на вектора поредицата дефинирана от start и end .
template < class Size, class T > void assign (Size num, const T &val = T());	Присвоява на вектора num елемента със стойност val .

```

reference at ( size_type i );
const_reference at ( size_type i ) const ;
reference back ( ) ;
const_reference back ( ) const ;
iterator begin ( ) ;
const_iterator begin ( ) const ;
size_type capacity ( ) const ;

void clear ( ) ;

bool empty ( ) const ;

iterator end ( ) ;
const_iterator end ( ) const ;
iterator erase ( iterator i ) ;

iterator erase ( iterator start , iterator end ) ;

reference front ( ) ;
const_reference front ( ) const ;
allocator_type get_allocator ( ) const ;
iterator insert ( iterator i ,
                  const T &val = T ( ) ) ;

void insert ( iterator i , size_type num ,
              const T &val ) ;

template < class InIter >
void insert ( iterator i ,
              InIter start , InIter end ) ;

size_type max_size ( ) const ;

reference operator [] ( size_type i ) const ;
const_reference
    operator [] ( size_type i ) const ;
void pop_back ( ) ;

void push_back ( const T &val ) ;

reverse_iterator rbegin ( ) ;
const_reverse_iterator rbegin ( ) const ;
reverse_iterator rend ( ) ;
const_reverse_iterator rend ( ) const ;
void reserve ( size_type num ) ;

void resize ( size_type num , T val = T ( ) ) ;

```

Връща псевдоним за елемента определен чрез *i*
Връща псевдоним за последния елемент във вектора
Връща итератор за първия елемент във вектора
Връща текущия капацитет на вектора. Това е броя елементи които може да съдържа преди да трябва да заделва повече памет
Премахва всички елементи от вектора
Връща *true* ако извикания вектор е празен и *false* в противен случай
Връща итератор към края на вектора
Премахва елемента указван от *i* връща итератор към елемента след премахнатия
Премахва елементите в интервала *start – end*, връща итератор към елемента след последния премахнат елемент
Връща псевдоним към първия елемент във вектора
Връща алокатор за вектора
Вмъква *val* непосредствено преди елемента определен от *i*
Връща се итератор към елемента
Вмъква *num* копия от *val* непосредствено преди елемента определен от *i*
Вмъква поредицата определена от *start – end* непосредствено преди елемента определен от *i*
Връща максималния брой елементи които може да съдържа вектора
Връща псевдоним за *i*-тия елемент
Премахва последния елемент във вектора
Добавя елемент със стойност *val* към края на вектора
Връща обратен итератор към края на вектора
Връща обратен итератор към началото на вектора
Установява капацитета на вектора така че да е равен най – малко на *num*
Променя размера на вектора към този определен от *num*. Ако вектора трябва да бъде удължен,

size_type size () const ;

void swap (vector < T , Allocator > &ob) ;

STL също съдържа специализиран вектор за Булеви стойности . Той включва цялата функционалност на vector и добавя тези два члена :

void flip () ;

static void swap (reference i , reference j) ;

тогава елементите със стойността
определена от *val* се добавят към края
Връща текущия брой елементи
във вектора
Разменя елементите съхранявани
във извиквания вектор със тези
в *ob*

Инвертира всички битове във
вектора

Разменя битовете определени
от *i* и *j*

Алгоритми

Стандартните алгоритми са описани тук . Всички алгоритми са шаблонни функции . За опростяване на описанията няма да бъде представяна шаблонната спецификация . Вместо това описанията ще използват навсякъде следните общи типови имена :

Общо име

BiIter

ForIter

InIter

OutIter

RandIter

T

Size

Func

Generator

BinPred

UnPred

Comp

Представя

Двупосочен итератор

Прав итератор

Входен итератор

Изходен итератор

Итератор с произволен
достъп

Някакъв тип данни

Някакъв тип **integer**

Някакъв тип функция

Функция която генерира
обекти

Бинарен предикат

Унарнен предикат

Сравнителна функция

която връща резултата от
arg1 < arg2

adjacent_find

ForIter adjacent_find (ForIter start , ForIter end) ;

*ForIter adjacent_find (ForIter start , ForIter end ,
BinPred pfn) ;*

Алгоритъма *adjacent_find ()* търси съседни съответстващи елементи в поредица определена от *start* и *end* и връща итератор за първия елемент . Ако не е открита съседна двойка се връща *end* . Първата версия търси за еквивалентни елементи . Втората версия позволява да определите метод за определяне на съвпадащите елементи .

binary_search

```

bool binary_search ( ForIter start, ForIter end,
                    const T &val );
bool binary_search ( ForIter start, ForIter end,
                    const T &val, Comp cmpfn );

```

Алгоритъма *binary_search* () извършва двоично търсене в подредената редица започваща от *start* и завършваща със *end* за стойността определена от *val*. Тя връща *true* ако открие *val* и *false* в противен случай. Първата версия сравнява елементите в определената редица за еднаквост. Втората версия позволява да определите ваша сравнителна функция.

copy

```

OutIter copy ( InIter start, InIter end, OutIter result );

```

Алгоритъма *copy* () копира последователността започваща от *start* и завършваща със *end* слагайки резултата в редицата указвана от *result*. Тя връща указател за края на резултатната последователност. Интервала който ще бъде копиран не трябва да превишава *result*.

copy_backward

```

BiIter2 copy_backward ( BiIter start, BiIter end,
                      BiIter2 result );

```

Алгоритъма *copy_backward* () е същия като *copy* () с изключение на това че премества елементите от края в началото на поредицата.

count

```

size_t count ( InIter start, InIter end, const T &val );

```

Алгоритъма *count* () връща броя елементи в поредицата започваща от *start* и завършваща със *end* които съвпадат със *val*.

count_if

```

size_t count_if ( InIter start, InIter end, UnPred pfn );

```

Алгоритъма *count_if* () връща броя елементи в поредицата започваща от *start* и завършваща със *end* за които унарния предикат *pfn* връща *true*.

equal

```

bool equal ( InIter1 start1, InIter1 end1, InIter2 start2 );

```

Алгоритъма *equal* () определя дали два интервала са еднакви. Интервала определен от *start1* и *end1* се сравнява със поредицата указвана от *start2*. Ако са еднакви се връща *true*, в противен случай се връща *false*.

equal_range

```

pair < ForIter, ForIter > equal_range ( ForIter start,
                                       ForIter end, const T &val );
pair < ForIter, ForIter > equal_range ( ForIter start,
                                       ForIter end, const T &val, Comp cmpfn );

```

Алгоритъма *equal_range* () връща интервал в който може да бъде вмъкнат елемент в последователността без да се наруши подреждането на редицата. Региона в който се търси за такъв интервал е определен от *start* и *end*.

Стойността се подава във *val*. За да определите ваш търсещ критерии, определете сравнителната функция *cmpfn*. Шаблонния клас *pair* е помощен който може да съдържа двойка обекти със техни *first* и *second* членове.

fill и fill_n

```
void fill ( ForIter start , ForIter end , const T &val ) ;  
void fill_n ( ForIter start , Size num , const T &val ) ;
```

Алгоритмите *fill()* и *fill_n()* попълват интервал със стойността определена от *val*. За *fill()* интервала е определен от *start* и *end*. За *fill_n()* интервала започва от *start* и продължава *num* елемента.

find

```
InIter find ( InIter start , InIter end , const T &val ) ;
```

Алгоритъма *find()* претърсва интервала от *start* до *end* за стойността определена от *val*. Той връща итератор към първото съвпадение на елемент или *end* ако стойността не е в редицата.

find_end

```
FwdIter1 find_end ( ForIter1 start1 , ForIter1 end1 ,  
ForIter2 start2 , ForIter2 end2 ) ;  
FwdIter1 find_end ( ForIter1 start1 , ForIter1 end1 ,  
ForIter2 start2 , ForIter2 end2  
BinPred pfn ) ;
```

Алгоритъма *find_end()* търси последния итератор от подниза определен от *start2* и *end2* в интервала *start1 - end1*. Ако редицата е открита се връща итератор към последния елемент в редицата. В противен случай се връща итератора *end1*. Втората форма ви позволява да определите бинарен предикат който определя кога елементите съвпадат.

find_first_of

```
FwdIter1 find_first_of ( ForIter1 start1 , ForIter1 end1 ,  
ForIter2 start2 , ForIter2 end2 ) ;  
FwdIter1 find_first_of ( ForIter1 start1 , ForIter1 end1 ,  
ForIter2 start2 , ForIter2 end2  
BinPred pfn ) ;
```

Алгоритъма *find_first_of()* открива първия елемент в поредицата определена от *start1* и *end1* който съвпада със елемент от интервала *start2* и *end2*. Ако няма съвпадащ елемент се връща итератора *end1*. Втората форма ви позволява да определите бинарен предикат който определя кога елементите съвпадат.

find_if

```
InIter find_if ( InIter start , InIter end , UnPred pfn ) ;
```

Алгоритъма *find_if()* претърсва интервала *start - end* за елемент за който унарния предикат *pfn* връща *true*. Той връща итератор към първото срещане или *end* ако стойността не е в поредицата.

for_each

```
Func for_each ( InIter start , InIter end , Func fn ) ;
```

Алгоритъмът *for_each()* добавя функцията *fn* към интервала от елементи определен от *start* – *end*. Тя връща *fn*.

generate и generate_n

```
void generate ( ForIter start , ForIter end , Generator fngen ) ;  
void generate_n ( OutIter start , Size num , Generator fngen ) ;
```

Алгоритмите *generate()* и *generate_n()* присвояват елементи в интервала от стойности върнати от генераторната функция. За *generate()* интервала който е бил присвоен е определен от *start* и *end*. За *generate_n()* интервала започва от *start* и продължава *num* елемента. Генераторната функция се подава във *fngen*. Тя няма параметри.

includes

```
bool includes ( InIter1 start1 , InIter1 end1 , InIter2 start2 ,  
                InIter2 end2 ) ;  
bool includes ( InIter1 start1 , InIter1 end1 , InIter2 start2 ,  
                InIter2 end2 , Comp cmpfn ) ;
```

Алгоритъмът *includes()* определя дали поредицата *start1* – *end1* включва всички елементи от поредицата *start2* – *end2*. Тя връща *true* ако са открити всички елементи и *false* в противен случай. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

inplace_merge

```
void inplace_merge ( Biter start , Biter mid , Biter end ) ;  
void inplace_merge ( Biter start , Biter mid , Biter end ,  
                    Comp cmpfn ) ;
```

Алгоритъмът *inplace_merge()* слива в единствена редица интервала определен от *start* и *mid* със интервала определен от *mid* и *end*. И двата интервала трябва да бъдат сортирани във нарастващ ред. След изпълнението, резултатната поредица е сортирана във нарастващ ред. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

iter_swap

```
void iter_swap ( ForIter1 i , ForIter2 j ) ;
```

Алгоритъмът *iter_swap()* разменя стойностите указвани чрез неговите два итератора аргументи.

lexicographical_compare

```
bool lexicographical_compare ( InIter1 start1 , InIter1 end1 ,  
                             InIter2 start2 , InIter2 end2 ) ;  
bool lexicographical_compare ( InIter1 start1 , InIter1 end1 ,  
                             InIter2 start2 , InIter2 end2  
                             Comp cmpfn ) ;
```

Алгоритъмът *lexicographical_compare()* сравнява по азбучен ред редицата определена от *start1* и *end1* със редицата определена от *start2* и *end2*. Тя връща *true* ако първата редица е лексикографски по – малка от втората (Така

е ако първата редица ще бъде преди втората във речников ред). Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

lower_bound

```
ForIter lower_bound ( ForIter start , ForIter end ,  
                      const T &val ) ;  
ForIter lower_bound ( ForIter start , ForIter end ,  
                      const T &val Comp cmpfn ) ;
```

Алгоритъма *lower_bound()* открива първата точка в редицата определена от *start* и *end* която не е по – малка от *val*. Тя връща итератор за тази точка. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

make_heap

```
void make_heap ( RandIter start , RandIter end ) ;  
void make_heap ( RandIter start , RandIter end ,  
                Comp cmpfn ) ;
```

Алгоритъма *make_heap()* изгражда хийп от редицата определена от *start* и *end*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

max

```
const T &max ( const T &i , const T &j ) ;  
const T &max ( const T &i , const T &j , Comp cmpfn ) ;
```

Алгоритъма *max()* връща максималната от две стойности. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

max_element

```
ForIter max_element ( ForIter start , ForIter last ) ;  
ForIter max_element ( ForIter start , ForIter last ,  
                    Comp cmpfn ) ;
```

Алгоритъма *max_element()* връща итератор към максималния елемент в интервала *start* и *last*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

merge

```
OutIter merge ( InIter1 start1 , InIter1 end1 , InIter2 start2 ,  
                InIter2 end2 , OutIter result ) ;  
OutIter merge ( InIter1 start1 , InIter1 end1 , InIter2 start2 ,  
                InIter2 end2 , OutIter result , Comp cmpfn ) ;
```

Алгоритъма *merge()* слива две подредени редици, поставяйки резултата в трета поредица. Редиците които се сливат се дефинират от *start1*, *end1* и *start2*, *end2*. Резултата се слага в редица указвана от *result*. Връща се итератор за края на резултатната редица. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

min

```
const T &min ( const T &i , const T &j ) ;  
const T &min ( const T &i , const T &j , Comp cmpfn ) ;
```

Алгоритъмът *min ()* връща минималната от две стойности. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

min_element

```
ForIter min_element ( ForIter start , ForIter last ) ;  
ForIter min_element ( ForIter start , ForIter last ,  
Comp cmpfn ) ;
```

Алгоритъмът *min_element ()* връща итератор към минималния елемент в интервала *start* и *last*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

mismatch

```
pair < InIter1 , InIter2 > mismatch ( InIter1 start1 ,  
InIter1 end1 , InIter2 start2 ) ;  
pair < InIter1 , InIter2 > mismatch ( InIter1 start1 ,  
InIter1 end1 , InIter2 start2 , BinPred pfn ) ;
```

Алгоритъмът *mismatch ()* открива първото несъвпадение между елементите в две редици. Връща се итератор за двата елемента. Ако не е открито несъвпадение, се връщат итератори за двата последни елемента във всяка поредица. Втората форма ви позволява да определите бинарен предикат който определя кога един елемент е равен на друг. Шаблонния клас *pair* съдържа два члена данни наречени *first* и *second* които съдържат двойка стойности.

next_permutation

```
bool next_permutation ( BIter start , BIter end ) ;  
bool next_permutation ( BIter start , BIter end ,  
Comp cmpfn ) ;
```

Алгоритъмът *next_permutation ()* изгражда следваща пермутация на редица. Пермутациите се генерират присвоявайки сортирана редица от ниско към високо представяне на първата пермутация. Ако не съществува следваща пермутация *next_permutation ()* сортира поредицата на нейната първа пермутация и връща *false*. В противен случай се връща *true*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

nth_element

```
void nth_element ( RandIter start , RandIter element ,  
RandIter end ) ;  
void nth_element ( RandIter start , RandIter element ,  
RandIter end , Comp cmpfn ) ;
```

Алгоритъмът *nth_element ()* подрежда редицата определена от *start* и *end* така че всички елементи по – малки от *element* идват преди елемента и всички елементи по – големи от *element* идват след него. Втората форма ви позволява

да определите сравнителна функция която определя кога един елемент е по – малък от друг .

partial_sort

```
void partial_sort ( RandIter start , RandIter mid ,  
                  RandIter end ) ;  
void partial_sort ( RandIter start , RandIter mid ,  
                  RandIter end , Comp cmpfn ) ;
```

Алгоритъма *partial_sort ()* сортира редицата *start – end* .Обаче след изпълнението само елементите в поредицата *start – mid* ще бъдат сортирани в ред . Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг .

partial_sort_copy

```
RandIter partial_sort_copy ( InIter start , InIter end ,  
                           RandIter res_start , RandIter res_end ) ;  
RandIter partial_sort_copy ( InIter start , InIter end ,  
                           RandIter res_start , RandIter res_end ,  
                           Comp cmpfn ) ;
```

Алгоритъма *partial_sort_copy ()* сортира интервала *start – end* и след това копира елементите които ще бъдат сложени в резултатната редица определена от *res_start* до *res_end* . Връща се итератор за последния елемент копиран в резултатната поредица . Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг .

partition

```
BIter partition ( BIter start , BIter end , UnPred pfn ) ;
```

Алгоритъма *partition ()* подрежда редицата определена от *start* до *end* така че всички елементи за които предиката *pfn* връща *true* идват преди онези за които предиката връща *false* .Тя връща итератор за началото на елементите за които предиката е *false* .

pop_heap

```
void pop_heap ( RandIter start , RandIter end ) ;  
void pop_heap ( RandIter start , RandIter end , Comp cmpfn ) ;
```

Алгоритъма *pop_heap ()* разменя *first* и *last* – елементите и тогава преизгражда хийпа . Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг .

prev_permutation

```
bool prev_permutation ( BIter start , BIter end ) ;  
bool prev_permutation ( BIter start , BIter end ,  
                      Comp cmpfn ) ;
```

Алгоритъма *prev_permutation ()* конструира предишната пермутация на редицата . Пермутациите се генерират приемайки сортирана редица от ниско към високо представяне за първа пермутация . Ако не съществува предишна пермутация , *prev_permutation ()* сортира редицата като нейна последна пермутация и връща *false* .В противен случай връща *true* .Втората форма ви

позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг .

push_heap

```
void push_heap ( RandIter start , RandIter end ) ;  
void push_heap ( RandIter start , RandIter end ,  
Comp cmpfn ) ;
```

Алгоритъма *push_heap ()* слага елемент в края на хийпа . Интервала определен от *start* до *ende* приет да представлява валиден хийп . Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг .

random_shuffle

```
void random_shuffle ( RandIter start , RandIter end ) ;  
void random_shuffle ( RandIter start , RandIter end ,  
Generator rand_gen ) ;
```

Алгоритъма *random_shuffle ()* разбърква редицата определена от *start – end* . Втората форма ви позволява да определя потребителски генератор на случайни числа . Тази функция трябва да има следната обща форма :

```
rand_gen ( num ) ;
```

Тя трябва да връща случайно число между 0 и *num* .

remove , remove_if , remove_copy и

remove_copy_if

```
ForIter remove ( ForIter start , ForIter end , const T &val ) ;  
ForIter remove_if ( ForIter start , ForIter end , UnPred pfn ) ;  
OutIter remove_copy ( InIter start , InIter end , OutIter result ,  
const T &val ) ;  
OutIter remove_copy_if ( InIter start , InIter end ,  
OutIter result , const T &val ,  
UnPred pfn ) ;
```

Алгоритъма *remove ()* премахва елементи от определения интервал които са равни на *val* . Връща итератор за края на останалите елементи .

Алгоритъма *remove_if ()* премахва елементите от определения интервал за които предиката *pfn* е *true* . Връща итератор за края на останалите елементи .

Алгоритъма *remove_copy ()* копира елементите от определения интервал които са равни на *val* и слага резултата в поредицата определена от *result* . Връща итератор за края на *result* .

Алгоритъма *remove_copy_if ()* копира елементите от определения интервал за които предиката *pfn* е *true* и слага резултата в поредицата определена от *result* . Връща итератор за края на *result* .

replace , replace_copy , replace_if и**replace_copy_if**

```

void replace ( ForIter start , ForIter end , const T &old ,
               Const T &new ) ;
void replace_if ( ForIter start , ForIter end , UnPred pfn ,
                  Const T &new ) ;
OutIter replace_copy ( InIter start , InIter end ,
                       OutIter result ,
                       const T &old , Const T &new ) ;
OutIter replace_copy_if ( InIter start , InIter end ,
                          OutIter result ,
                          UnPred pfn , Const T &new ) ;

```

В определения интервал алгоритъта **replace ()** разменя елементите които имат стойност **old** със елементите които имат стойност **new** .

В определения интервал алгоритъта **replace_if ()** разменя елементите за които предиката **pfn** е **true** със елементите които имат стойност **new** .

В определения интервал алгоритъта **replace_copy ()** копира елементи в **result** .

В този процес разменя елементите със стойност **old** със елементи със стойност **new** . Първоначалния интервал е непроменен . Връща итератор за края на **result** .

В определения интервал алгоритъта **replace_copy_if ()** копира елементи в **result** . В този процес разменя елементите за които предиката **pfn** връща **true** със елементи със стойност **new** . Първоначалния интервал е непроменен .

Връща итератор за края на **result** .

reverse и reverse_copy

```

void reverse ( BIter start , BIter end ) ;
OutIter reverse_copy ( BIter first , BIter last ,
                       OutIter result ) ;

```

Алгоритъта **reverse ()** обръща реда на интервала определен от **start** и **end** .

Алгоритъта **reverse_copy ()** копира в обърнат ред интервала определен от **start** и **end** и съхранява резултата в **result** . Връща итератор за края на **result** .

rotate и rotate_copy

```

void rotate ( ForIter start , ForIter mid , ForIter end ) ;
OutIter rotate_copy ( ForIter start , ForIter mid , ForIter end ,
                      OutIter result ) ;

```

Алгоритъта **rotate ()** ротира наляво елементите в интервала от **start** до **end** така че елемента определен от **mid** става новия първи елемент .

Алгоритъта **rotate_copy ()** копира интервала от **start** до **end** съхранявайки резултата във **result** . По време на процеса ротира наляво елементите така че елемента определен от **mid** става новия първи елемент . Връща итератор за края на **result** .

search

```

ForIter1 search ( ForIter1 start1 , ForIter1 end1 ,
                  ForIter2 start2 , ForIter2 end2 ) ;
ForIter1 search ( ForIter1 start1 , ForIter1 end1 ,
                  ForIter2 start2 , ForIter2 end2 , BinPred pfn );

```

Алгоритъмът *search* () търси за поредица във редица. Редицата която се претърсва е определена от *start1* и *end1*. Поредицата която се търси е определена от *start2* и *end2*. Ако поредицата е открита се връща итератор за нейното начало. В противен случай се връща *end1*. Втората форма ви позволява да определите бинарен предикат който определя кога един елемент е равен на друг.

search_n

```

ForIter search_n ( ForIter start , ForIter end , Size num ,
                  Const T &val ) ;
ForIter search_n ( ForIter start , ForIter end , Size num ,
                  Const T &val , BinPred pfn ) ;

```

Алгоритъмът *search_n* () търси за редица от подобни на *num* елементи в редица. Редицата която се претърсва е определена от *start* и *end*. Ако поредицата е открита се връща итератор за нейното начало. В противен случай се връща *end*. Втората форма ви позволява да определите бинарен предикат който определя кога един елемент е равен на друг.

set_difference

```

OutIter set_difference ( InIter1 start1 , InIter1 end1 ,
                        InIter2 start2 , InIter2 end2 ,
                        OutIter result ) ;
OutIter set_difference ( InIter1 start1 , InIter1 end1 ,
                        InIter2 start2 , InIter2 end2 ,
                        OutIter result , Comp cmpfn ) ;

```

Алгоритъмът *set_difference* () произвежда редица която съдържа разликата между две подредени множества определени от *start1*, *end1* и *start2*, *end2*. Така множеството определено от *start2*, *end2* се изважда от множеството *start1*, *end1*. Резултатът е подреден и сложен във *result*. Връща итератор за края на *result*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по-малък от друг.

set_intersection

```

OutIter set_intersection ( InIter1 start1 , InIter1 end1 ,
                          InIter2 start2 , InIter2 end2 ,
                          OutIter result ) ;
OutIter set_intersection ( InIter1 start1 , InIter1 end1 ,
                          InIter2 start2 , InIter2 end2 ,
                          OutIter result , Comp cmpfn ) ;

```

Алгоритъмът *set_intersection* () произвежда редица която съдържа сечението на две подредени множества определени от *start1*, *end1* и *start2*, *end2*. Това са елементите открити и във двете множества. Резултатът е подреден и сложен

във *result*. Връща итератор за края на *result*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

set_symetric_difference

```
OutIter set_symetric_difference ( InIter1 start1 , InIter1 end1 ,
                                   InIter2 start2 , InIter2 end2 ,
                                   OutIter result ) ;
OutIter set_symetric_difference ( InIter1 start1 , InIter1 end1 ,
                                   InIter2 start2 , InIter2 end2 ,
                                   OutIter result , Comp cmpfn ) ;
```

Алгоритъма *set_symetric_difference ()* произвежда редица която съдържа симетричната разлика между две подредени множества определени от *start1* , *end1* и *start2* , *end2*. Това е резултатно множество съдържа само онези елементи които не са общи за двете множества. Резултата е подреден и сложен във *result*. Връща итератор за края на *result*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

set_union

```
OutIter set_union ( InIter1 start1 , InIter1 end1 ,
                    InIter2 start2 , InIter2 end2 , OutIter result ) ;
OutIter set_union ( InIter1 start1 , InIter1 end1 ,
                    InIter2 start2 , InIter2 end2 , OutIter result
                    Comp cmpfn ) ;
```

Алгоритъма *set_union ()* произвежда редица която съдържа обединението на две подредени множества определени от *start1* , *end1* и *start2* , *end2*. Така резултатното множество съдържа тези елементи които са и във двете множества. Резултата е подреден и сложен във *result*. Връща итератор за края на *result*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

sort

```
void sort ( RandIter start , RandIter end ) ;
void sort ( RandIter start , RandIter end , Comp cmpfn ) ;
```

Алгоритъма *sort ()* сортира интервала определен от *start* и *end*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

sort_heap

```
void sort_heap ( RandIter start , RandIter end ) ;
void sort_heap ( RandIter start , RandIter end , Comp cmpfn ) ;
```

Алгоритъма *sort_heap ()* сортира хийп определен от *start* и *end*. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

stable_partition

*Biliter stable_partition (Biliter start , Biliter end ,
BinPred pfn) ;*

Алгоритъма *stable_partition ()* подрежда редицата определена от *start* и *end* така че всички елементи за които предиката определен от *pfn* връща *true* идват преди тези за които предиката връща *false*. Разделянето е постоянно. Това означава че съответното подреждане се запазва. Връща итератор към началото на елементите за които предиката е *false*.

stable_sort

*void stable_sort (RandIter start , RandIter end) ;
void stable_sort (RandIter start , RandIter end , Comp cmpfn) ;*

Алгоритъма *stable_sort ()* подрежда интервала *start – end*. Сортирането е постоянно. Това означава че еднаквите елементи не се пренареждат. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

swap

void swap (T &i , T &j) ;

Алгоритъма *swap ()* разменя стойностите отнасящи се за *i* и *j*.

swap_ranges

*ForIter2 swap_ranges (ForIter1 start1 , ForIter1 end1 ,
ForIter2 start2) ;*

Алгоритъма *swap_ranges ()* разменя елементите в интервала определен от *start1* и *end1* със елементите в редицата започваща със *start2*. Връща указател към края на редицата определена от *start2*.

transform

*OutIter transform (InIter start , InIter end , OutIter result ,
Func unaryfunc) ;
OutIter transform (InIter1 start1 , InIter1 end1 , InIter2
start2 , OutIter result , Func binaryfunc) ;*

Алгоритъма *transform ()* прибавя функция към интервала от елементи и съхранява резултата във *result*. В първата форма интервала е определен от *start* и *end*. Функцията която ще бъде добавена е определена от *unaryfunc*. Тази функция получава стойност на елемент като нейн параметър и трябва да връща неговата трансформация. Във втората форма, трансформацията е добавена чрез използване на бинарна операторна функция която получава елемент от редицата и го преобразува във нейн първи параметър и елемент от втората редица за нейн втори параметър. И двете версии връщат итератор за края на резултатната редица.

Програмен съвет

Един от най – интересните алгоритми е *transform ()* понеже той изменя всеки елемент в интервала съгласно предоставената функция. Например, следващата програма използва проста преобразуваща функция наречена *xform ()* да вдигне

на квадрат съдържанието на списък. Забележете че резултатната редица се съхранява в същия списък който доставя първоначалната редица :

```
// пример за трансформиращ алгоритъм
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
// проста преобразуваща функция
int xform (int i) {
    return i * i; // първоначалната стойност на квадрат
}
int main ( )
{
    list<int> xl;
    int i;
    // слага стойности във списъка
    for (i = 0; i < 10; i++) xl.push_back(i);
    cout << "Original contents of xl: ";
    list<int>::iterator p = xl.begin();
    while (p != xl.end()) {
        cout << *p << " ";
        p++;
    }
    cout << endl;
    // трансформира xl
    p = transform(xl.begin(), xl.end(), xl.begin(),
        xform);
    cout << "Transformed contents of xl: ";
    p = xl.begin();
    while (p != xl.end()) {
        cout << *p << " ";
        p++;
    }
    return 0;
}
```

Изхода от тази програма е :

Original contents of xl: 0 1 2 3 4 5 6 7 8 9

Transformed contents of xl: 0 1 4 9 16 25 36 49 64 81

Както можете да видите всеки елемент в *xl* е бил повдигнат на квадрат.

unique и unique_copy

```
ForIter unique (ForIter start, ForIter end);
ForIter unique (ForIter start, ForIter end, BinPred pfn);
OutIter unique_copy (ForIter start, ForIter end,
    OutIter result);
OutIter unique_copy (ForIter start, ForIter end,
    OutIter result, BinPred pfn);
```

Алгоритъмът *unique()* елиминира дублиращите се елементи от определения интервал. Втората форма ви позволява да определите бинарен предикат който определя кога един елемент е равен на друг. *unique()* връща итератор за края на интервала. Алгоритъмът *unique_copy()* копира интервала определен от *start* и *end*, елиминирайки дублиращите се елементи по време на процеса. Резултатът е сложен във *result*. Втората форма ви позволява да определите бинарен предикат който определя кога един елемент е равен на друг. *unique_copy()* връща итератор за края на интервала.

upper_bound

```
ForIter upper_bound ( ForIter start , ForIter end ,
                      const T &val ) ;
ForIter upper_bound ( ForIter start , ForIter end ,
                      const T &val , Comp cmpfn ) ;
```

Алгоритъмът *upper_bound()* открива последната точка в редицата определена от *start* и *end* която не е по – голяма от *val*. Връща итератор за тази точка. Втората форма ви позволява да определите сравнителна функция която определя кога един елемент е по – малък от друг.

ГЛАВА 15 – C++ Низове и изключения

В добавка към *iostream* библиотеката и *STL* стандартната C++ библиотека дефинира няколко други класа които манипулират разнообразие от ситуации. Например, има класове които поддържат числови операции, комплексна аритметика, локализация. Докато много от тези класове не са често използвани във всекидневното програмиране, два са: низове и изключения. Те са описани тук.

Низове

C++ поддържа символните низове по два начина. Първия е нулево – прекъснат символен масив. Той понякога се представя като C – низ. Втория начин е като класов обект от тип *basic_string*. Има две разновидности на *basic_string*: *string* който поддържа символни низове и *wstring* който поддържа широко – символни низове. Най – често ще използвате низови обекти от тип *string* и те са разгледани тук. Класа използван от *basic_string* е *char_traits* който дефинира няколко атрибута на символите които съставят низа. Важно е да разберете че докато най – общите низове са изградени от или *char* или *wchar_t* символи, *basic_string* може да работи със всеки обект който може да бъде използван да представя текстов символ. Класа *basic_string* е в основата си контейнер. Това значи че той поддържа алгоритмите описани във секцията за *STL*. Обаче низовете имат допълнителни възможности. За да използвате *string* обекти ще трябва да включите *<string>*. Шаблонната спецификация за *basic_string* е:

```
template < class charT , class Traits = char_traits < charT > ,
          class Allocator = allocator < T > >
class basic_string
```

Тук, *charT* е тип на символите които се използват, *Traits* е класа който описва символите и *Allocator* определя алокатора. Той има следните конструктори:

```

explicit basic_string ( const Allocator &a = Allocator () ) ;
basic_string ( size_type len , charT ch ,
               const Allocator &a = Allocator () ) ;
basic_string ( const charT *str ,
               const Allocator &a = Allocator () ) ;
basic_string ( const charT *str , size_type len ,
               const Allocator &a = Allocator () ) ;
basic_string ( const basic_string &str , size_type indx = 0 ,
               size_type len , const Allocator &a = Allocator () ) ;
template < class InIter >
basic_string ( InIter start , InIter end ,
               const Allocator &a = Allocator () ) ;

```

Първата форма конструира празен низ. Втората форма конструира низ който има *len* символа със стойност *ch*. Третата форма изгражда низ който съдържа еднакви елементи като *str*. Четвъртата форма изгражда низ който съдържа подниз *str* започващ от 0 и дълг *len* символа. Петата форма изгражда низ от друг *basic_string* използвайки подниз който започва от *indx* и е дълг *len* символа. Шестата форма изгражда низ който съдържа елементите в интервала определен от *start* и *end*. Следващите сравнителни оператори са дефинирани за *basic_string*: *==*, *<*, *<=*, *!=*, *>* и *>=*. Също е дефиниран и оператора *+* който дава резултата от свързването на един низ със друг и I/O операторите *<<* и *>>* които могат да бъдат използвани за въвеждане и извеждане на низове. Оператора *+* може да бъде използван за свързване на низов обект със друг низов обект или низов обект със низ във C-стил. Така се поддържат следните случаи:

```

string + string
string + C – string
C – string + string

```

Оператора *+* може да бъде използван да свързва символ към края на низ. Класа *basic_string* дефинира константа *npos* която обикновено е -1. Тази константа представя дължината на най-дългия възможен низ. В описанията общия тип *charT* представя типа на символите съхранявани от низа. Понеже имената на плейсхолдърите са случайни, контейнерните класове декларират *typedef* версии на тези типове. Това прави имената на типовете точни. Общо използваните типове дефинирани от *basic_string* са изброени тук:

<i>size_type</i>	Някакъв целочислен тип еквивалентен на <i>size_t</i>
<i>reference</i>	Псевдоним на символ
<i>const_reference</i>	const псевдоним на символ
<i>iterator</i>	Итератор
<i>const_iterator</i>	const итератор
<i>reverse_iterator</i>	Обратен итератор
<i>const_reverse_iterator</i>	const обратен итератор
<i>value_type</i>	Тип на символите съхранявани в низа

allocator_type
pointer

const_pointer

Типа на алокатора
Указател към символ
в низа

const указател към символ
в низа

basic_string съдържа следните член функции :

Член

Описание

```
basic_string
    &append ( const basic_string &str );

basic_string
    &append ( const basic_string &str ,
              size_type indx , size_type len );

basic_string &append ( const charT *str );

basic_string &append ( const charT *str
                    size_type num );

basic_string &append ( size_type len
                    char T ch );

template < class InIter >
    basic_string &append ( InIter start ,
                        InIter end );

basic_string
    &assign ( const basic_string &str );

basic_string
    &assign ( const basic_string &str ,
            size_type indx , size_type len );

basic_string &assign ( const charT *str );

basic_string &assign ( const charT *str ,
                    size_type len );

basic_string &assign ( size_type len
                    char T ch );

template < class InIter >
    basic_string &assign ( InIter start ,
                        InIter end );

reference at ( size_type indx );
const_reference at ( size_type indx ) const ;
iterator begin ( ) ;
const_iterator begin ( ) const ;
const charT *c_str ( ) const ;

size_type capacity ( ) const ;
```

Добавя *str* в края на извиквания
низ ; връща **this*

Добавя подниза *str* в края на
извиквания низ . Подниза който
се присвоява започва от *indx* и
продължава *len* символа ;
връща **this*

Добавя *str* в края на
извиквания низ ; връща **this*
Добавя първите *num* символа в
края на извиквания низ ;
връща **this*

Добавя *len* символа определени
от *ch* в края на извиквания низ
връща **this*

Добавя редицата определена от
start и *end* в края на
извиквания низ ; връща **this*

Присвоява *str* на извиквания
низ връща **this*

Присвоява подниза *str* на
извиквания низ . Подниза който
се присвоява започва от *indx* и
продължава *len* символа ;
връща **this*

Присвоява *str* на извиквания
низ ; връща **this*

Присвоява първите *len* символа
от *str* на извиквания низ ;
връща **this*

Присвоява *len* символа
определени от *ch* към края на
извиквания низ ; връща **this*

Присвоява поредицата
определена от *start* и *end* към
края на извиквания низ ;

Връща псевдоним за символа
определен от *indx*

Връща итератор за първия
елемент в низа

Връща указател към C – стил
(тоест нулево – терминиран)
вид на извиквания низ

Връща текущия капацитет на

```

int compare ( const basic_string &str )
    const ;

int compare ( size_type indx , size_type len ,
    const basic_string &str ) const ;

int compare ( size_type indx , size_type len ,
    const basic_string &str ,
    size_type indx2 , size_type len2 ) const ;

int compare ( const char T *str ) const ;

int compare ( size_type indx , size_type len ,
    const char T *str ,
    size_type len2 = npos ) const ;

size_type copy ( char T *str , size_type len ,
    size_type indx = 0 ) const ;

const charT *data ( ) const ;

bool empty ( ) const ;

iterator end ( ) ;
const_iterator end ( ) const ;
iterator erase ( iterator i ) ;

iterator erase ( iterator start , iterator end ) ;

```

низа . Това е броя символи които може да съдържа преди да трябва да заделва още памет

Сравнява *str* със извиквания низ

Връща едно от следните :

по – малко от **0** ако **this < str*

0 ако **this == str*

по – голямо от **0** ако **this > str*

Сравнява *str* със подниз в извиквания низ . Подниза започва от *indx* и е *len* символа дълъг . Връща едно от следните :

по – малко от **0** ако **this < str*

0 ако **this == str*

по – голямо от **0** ако **this > str*

Сравнява подниз от *str* със подниз във извиквания низ .

Подниза във извиквания низ започва от *indx* и е *len* символа дълъг . Подниза във *str* започва от *indx2* и е *len2* символа дълъг . Връща едно от следните :

по – малко от **0** ако **this < str*

0 ако **this == str*

по – голямо от **0** ако **this > str*

Сравнява *str* със извиквания низ

Връща едно от следните :

по – малко от **0** ако **this < str*

0 ако **this == str*

по – голямо от **0** ако **this > str*

Сравнява подниз от *str* със подниз във извиквания низ .

Подниза във извиквания низ започва от *indx* и е *len* символа дълъг . Подниза във *str* започва от **0** и е *len2* символа дълъг . Връща едно от следните :

по – малко от **0** ако **this < str*

0 ако **this == str*

по – голямо от **0** ако **this > str*

Започвайки от *indx* , копира *len* символа от извиквания низ в символен масив указван от *str* ;

Връща броя на копираните символи

Връща указател към първия символ във извиквания низ

Връща *true* ако извиквания низ е празен и *false* в противен случай

Връща итератор към края на низа

Премахва символа указван от *i*

върща итератор за символа след първия премахнат

Премахва символите в

```
basic_string &erase ( size_type indx = 0 ,
                     size_type len = npos ) ;
```

```
size_type find ( const basic_string &str ,
                 size_type indx = 0 ) const ;
```

```
size_type find ( const charT *str ,
                 size_type indx = 0 ) const ;
```

```
size_type find ( const charT *str ,
                 size_type indx , size_type len )
const ;
```

```
size_type find ( charT ch ,
                 size_type indx = 0 ) const ;
```

```
size_type
find_first_of ( const basic_string &str ,
                size_type indx = 0 ) const ;
```

```
size_type find_first_of ( const charT *str ,
                          size_type indx = 0 ) const ;
```

```
size_type find_first_of ( const charT *str ,
                          size_type indx , size_type len )
const ;
```

```
size_type find_first_of ( charT ch ,
                          size_type indx = 0 ) const ;
```

```
size_type
find_first_not_of ( const basic_string &str ,
                   size_type indx = 0 ) const ;
```

интервала *start* – *end* , връща итератор за символа след последния премахнат символ

Започвайки от *indx* премахва *len* символа от извиквания низ ;
връща **this*

Връща индекса за първото срещане на *str* в извиквания низ
Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първото срещане на *str* в извиквания низ
Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първото срещане на първите *len* символа от *str* във извиквания низ .

Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първото срещане на *ch* в извиквания низ
Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първия символ във извиквания низ който съпада със някой символ във *str* . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първия символ във извиквания низ който съпада със някой символ във *str* . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първия символ във извиквания низ който съпада със някой символ от първите *len* символа във *str* . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първото срещане на *ch* във извиквания низ
Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение

Връща индекса за първия символ във извиквания низ който не съпада със никой

```
size_type
find_first_not_of( const charT *str,
                  size_type indx = 0 ) const ;
```

```
size_type
find_first_not_of( const charT *str,
                  size_type indx, size_type len )
const ;
```

```
size_type find_first_not_of( charT ch,
                            size_type indx = 0 ) const ;
```

```
size_type
find_last_of( const basic_string &str,
              size_type indx = npos ) const ;
```

```
size_type find_last_of( const charT *str,
                       size_type indx = npos ) const ;
```

```
size_type find_last_of( const charT *str,
                       size_type indx, size_type len )
const ;
```

```
size_type find_last_of( charT ch,
                       size_type indx = npos ) const ;
```

```
size_type
find_last_not_of( const basic_string &str,
                 size_type indx = npos ) const ;
```

```
size_type
find_last_not_of( const charT *str,
                 size_type indx = npos ) const ;
```

символ във *str*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито несъвпадение

Връща индекса за първия символ във извиквания низ който не съвпада със никой символ във *str*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито несъвпадение

Връща индекса за първия символ във извиквания низ на който не съвпада никой символ във първите *len* символа от *str*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито несъвпадение

Връща индекса за първия символ във извиквания низ на който не съвпада *ch*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито несъвпадение

Връща индекса на последния символ във извиквания низ който съвпада със някой символ във *str*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито съвпадение

Връща индекса на последния символ във извиквания низ който съвпада със някой символ във *str*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито съвпадение

Връща индекса на последния символ във извиквания низ който съвпада със някой символ от първите *len* символа във *str*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито съвпадение

Връща индекса на последното срещане на *ch* във извиквания низ. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито съвпадение

Връща индекса на последния символ във извиквания низ който не съвпада със никой символ във *str*. Търсенето започва от индекс *indx*. *npos* се връща ако няма открито несъвпадение

Връща индекса на последния символ във извиквания низ който не съвпада със никой

```

size_type
    find_last_not_of ( const charT *str ,
                      size_type indx , size_type len )
    const ;

size_type find_last_not_of ( charT ch ,
                          size_type indx = npos ) const ;

allocator_type get_allocator ( ) const ;
iterator insert ( iterator i ,
                const charT &ch = charT ( ) ) ;

basic_string &insert ( size_type indx ,
                    const basic_string &str ) ;

basic_string &insert ( size_type indx1 ,
                    const basic_string &str ,
                    size_type indx2 , size_type len ) ;

basic_string &insert ( size_type indx ,
                    const charT *str ) ;

basic_string &insert ( size_type indx ,
                    const charT *str , size_type len ) ;

basic_string &insert ( size_type indx ,
                    size_type len , charT ch ) ;

void insert ( iterator i , size_type len ,
            const charT &ch ) ;

template < class InIter >
    void insert ( iterator i , InIter start ,
                InIter end ) ;

size_type length ( ) const ;
size_type max_size ( ) const ;

reference operator [ ] ( size_type indx )
    const ;
const_reference operator [ ] ( size_type indx )
    const ;
basic_string
    &operator = ( const basic_string &str ) ;
basic_string

```

символ във *str* . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито несъвпадение

Връща индекса на последния символ във извиквания низ на който не съвпада никой символ във първите *len* символа от *str* . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито несъвпадение

Връща индекса на последния символ във извиквания низ на който не съвпада *ch* . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито несъвпадение

Връща алокатор за низ

Вмъква *ch* непосредствено преди символа определен от *i*

Връща се итератор към символа

Вмъква *str* във извиквания низ на индекса определен от *indx* .

Връща **this*

Вмъква подниз от *str* във извиквания низ на индекс определен от *indx1* . Подниза започва от *indx2* и е *len* символа дълъг . Връща **this*

Вмъква *str* във извиквания низ на индекса определен от *indx* .

Връща **this*

Вмъква първите *len* символа от *str* в извиквания низ на индекса определен от *indx* .

Връща **this*

Вмъква *len* символа със стойност *ch* в извиквания низ на индекса определен от *indx* .

Връща **this*

Вмъква *len* копия от *ch* непосредствено преди елемента определен от *i*

Вмъква поредицата определена от *start – end* непосредствено преди елемента определен от *i*

Връща броя на символите в низа

Връща максималния брой символи които може да съдържа низа

Връща псевдоним за символа определен от *indx*

Присвоява определения низ или символ на извиквания низ ;

връща **this*

```

&operator = ( const charT *str );
basic_string &operator = ( charT ch );
basic_string
&operator += ( const basic_string &str );
&operator += ( const charT *str );
basic_string &operator += ( charT ch ); низ ;

```

Добавя определения низ или символ към края на извиквания *basic_string* връща **this*

```

reverse_iterator rbegin ( ) ;
const_reverse_iterator rbegin ( ) const ;
reverse_iterator rend ( ) ;
const_reverse_iterator rend ( ) const ;
basic_string &replace ( size_type indx ,
size_type len ,
const basic_string &str );
basic_string &replace ( size_type indx1 ,
size_type len1 , const basic_string &str ,
size_type indx2 , size_type len2 );

```

```

basic_string &replace ( size_type indx ,
size_type len , const charT *str );

```

```

basic_string &replace ( size_type indx1 ,
size_type len1 , const charT *str ,
size_type len2 );

```

```

basic_string &replace ( size_type indx ,
size_type len1 , size_type len2 ,
charT ch );

```

```

basic_string &replace ( iterator start ,
iterator end , const basic_string &str );

```

```

basic_string &replace ( iterator start ,
iterator end , const charT *str );

```

```

basic_string &replace ( iterator start ,
iterator end , const charT *str ,
size_type len );
basic_string &replace ( iterator start ,
iterator end , size_type len , charT ch );

```

```

template < class InIter >
basic_string &replace ( iterator start1 ,
iterator end1 , InIter start2 ,
InIter end2 );
void reserve ( size_type num = 0 );

```

```

void resize ( size_type num );
void resize ( size_type num , charT ch );

```

```

size_type rfind ( const basic_string &str ,

```

Връща обратен итератор към края на низа
Връща обратен итератор към началото на низа
Замества *len* символа във извиквания низ започвайки от *indx* със низа в *str*; връща **this*
Замества *len1* символа във извиквания низ започвайки от *indx1* със *len2* символа от низа в *str* започващи от *indx2*; връща **this*
Замества *len* символа във извиквания низ започвайки от *indx* със низа в *str*; връща **this*
Замества *len1* символа във извиквания низ започвайки от *indx1* със *len2* символа от низа в *str* започващи от *indx2*; връща **this*
Замества *len1* символа във извиквания низ започвайки от *indx1* със *len2* символа от низа в *str* започващи от *indx2*; връща **this*
Замества *len1* символа във извиквания низ започвайки от *indx1* със *len2* символа определени от *ch*; връща **this*
Замества интервала определен от *start* и *end* със *str*; връща **this*
Замества интервала определен от *start* и *end* със *str*; връща **this*

Замества интервала определен от *start* и *end* със първите *len* символа от *str*; връща **this*
Замества интервала определен от *start* и *end* със *len* символа от *str*; връща **this*
Замества интервала *start1 – end1* със символите определени чрез *start2 – end2*; връща **this*

Установява капацитета на низа така че да е равен най – малко на *num*

Променя размера на низа към този определен от *num*. Ако низа трябва да бъде удължен, тогава елементите със стойността определена от *ch* се добавят към края

Връща индекса на последното

```

size_type indx = npos ) const ;

size_type rfind ( const charT *str ,
size_type indx = npos ) const ;

size_type rfind ( const charT *str ,
size_type indx , size_type len ) const ;

size_type rfind ( charT ch ,
size_type indx = npos ) const ;

size_type size ( ) const ;

basic_string substr ( size_type indx = 0 ,
size_type len = npos ) const ;

void swap ( basic_string &str ) ;

```

срещане на *str* във извиквания низ .
Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение
Връща индекса на последното срещане на *str* във извиквания низ . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение
Връща индекса на последното срещане на първите *len* символа от *str* във извиквания низ . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение
Връща индекса на последното срещане на *ch* във извиквания низ . Търсенето започва от индекс *indx* . *npos* се връща ако няма открито съвпадение
Връща текущия брой символи в низа
Връща подниз от *len* символа започвайки от *indx* във извиквания низ
Разменя символите съхранявани в извиквания низ със тези в *str*

Програмен съвет

Докато традиционните низове във C – стил са били винаги прости за употреба , C++ - низовите класове правят низовото манипулиране извънредно лесно .
Например , използвайки *string* обекти може да добавите оператор за присвояване на ограден с кавички низ към *string* , релационни оператори за сравняване на низове и голямо разнообразие от низоманипулиращи функции които правят поднизовите операции удобни . Например , разгледайте следната програма :

```

// Демонстрира string
#include <iostream>
#include <string>
using namespace std ;

int main ( )
{
    string str1 = " abcdefghijklmnopqrstuvwxyz " ;
    string str2 ;
    string str3 ( str1 ) ;
    str2 = str1.substr ( 10 , 5 ) ;
    cout << "str1 : " << str1 << endl ;
    cout << "str2 : " << str2 << endl ;
    cout << "str3 : " << str3 << endl ;
    str1.replace ( 5 , 10 , " " ) ;
}

```

```

cout << "str1.replace ( 5 , 10 , \" \" ) : \"
    << str1 << endl ;
str1 = " one " ;
str2 = " two " ;
str3 = " three " ;
cout << "str1.compare ( str2 ) : \" ;
cout << str1.compare ( str2 ) << endl ;
if ( str1 < str2 ) cout << "str1 is less than str2 \n \" ;
string str4 = str1 + " \" + str2 + " \" + str3 ;
cout << "str4 : \" << str4 << endl ;
int i = str4.find ( " wo \" ) ;
cout << "str4.substr ( i ) : \" << str4.substr ( i ) ;
return 0 ;
}

```

Изхода на програмата е :

```

str1 : abcdefghijklmnopqrstuvwxyz
str2 : klmno
str3 : abcdefghijklmnopqrstuvwxyz
str1.replace ( 5 , 10 , \" \" ) : abcdeqrstuvwxyz
str1.compare ( str2 ) : -1
str1 is less than str2
str4 : one two three
str4.substr ( i ) : wo three

```

Забележете леснотата с която низовото манипулиране е изпълнено . Например , + се използва да свърже низове и < се използва да сравни два низа . Изпълнявайки тези операции със използване на C – стиловете , нулево – терминирани низове , е по – удобно от извикването на *strcat ()* и *strcmp ()* където са изисквани . Понеже C++ *string* обектите могат да бъдат свободно смесвани със C – стиловете , нулево – терминирани низове , няма неудобство да ги използвате в програми – а има значителна ползи .

Исключения

Стандартната C++ библиотека дефинира два хедъра които се отнасят за изключенията : < *exception* > и < *stdexcept* > . Изключенията се използват да докладват за грешни условия . Всеки от тези хедъри е разгледан тук .

< *exception* >

Хедъра < *exception* > дефинира класове , типове и функции които са свързани със манипулирането на изключения . Класовете дефинирани от < *exception* > са показани тук :

Клас
exception

bad_exception

Предназначение

Базов клас за всички изключения дефинирани в стандартната C++ библиотека
Типа изключение върнато от функцията *unexpected ()*

Типовете дефинирани от < *exception* > са :

Тип	Значение
<i>terminate_handler</i>	<i>typedef void (*terminate_handler)();</i>
<i>unexpected_handler</i>	<i>typedef void (*unexpected_handler)();</i>

Функциите са показани тук :

Функция	Описание
<i>terminate_handler</i> <i>set_terminate (terminate_handler fn)</i> <i>throw ();</i>	Установява функцията определена от <i>fn</i> като прекъсващ манипулатор . Връща се указател към стария прекъсващ манипулатор
<i>unexpected_handler</i> <i>set_unexpected (unexpected_handler fn)</i> <i>throw ();</i>	Установява функцията определена от <i>fn</i> като unexpected манипулатор . Връща се указател към стария unexpected манипулатор
<i>void terminate ();</i>	Извиква прекъсващия манипулатор когато не е обработено фатално изключение ; извиква <i>abort ()</i> по подразбиране
<i>bool uncaught_exception ();</i>	Връща <i>true</i> ако е непрехванато изключение
<i>void unexpected ();</i>	Извиква манипулатора за unexpected изключения когато функция прехвърли неразрешено изключение . По подразбиране се извиква <i>terminate ()</i>

< stdexcept >

Хедъра **< stdexcept >** дефинира няколко стандартни изключения които могат да бъдат прехвърлени от C++ - библиотечни функции и/или системата . Има два общи типа изключения дефинирани от **< stdexcept >** : логически грешки и грешки по време на изпълнение . Логически грешки стават при грешка на програмиста . Грешки по време на изпълнение се причиняват от грешки във библиотечните функции или **runtime** системата и са извън контрола на програмиста . Стандартните изключения дефинирани от C++ причинени от логически грешки са производни на базовия клас **logic_error** . Тези изключения са изброени тук :

Изключение	Значение
<i>domain_error</i>	Станала е грешка в областта
<i>invalid_argument</i>	Използва се невалиден аргумент във извикването на функция
<i>length_error</i>	Опит за създаване на обект който е твърде голям
<i>out_of_range</i>	Аргумента на функция не е във изисквания интервал

Следните грешки по време на изпълнение са производни от базовия клас **runtime_error** :

Изключение	Значение
<i>overflow_error</i>	Станало е аритметично препълване
<i>range_error</i>	Станала е грешка във вътрешния интервал
<i>underflow_error</i>	Станало е препълване след десетичната точка .