

Консултация по УП за контролно 2

Изготвена и представена от Мартин Илиев

Какво покрива тази презентация

- Референции
- Пойнтъри
- Масиви
- Символни низове
- Примерни задачи
- Време за въпроси (надявам се да остане такава)

Функция разменяща стойностите на 2 променливи

```
void swap(double a, double b)
{
    double c = a;
    a = b;
    b = c;
}
```

- Нищо няма да се случи, защото a и b са нови временни обекти.
- Тези променливи не са свързани с променливите, които сме подали като параметри!!!

Какво е паметта

- Както казахме преди, паметта представлява много последователни битове, групирани в байтове, които може да са групирани в нещо по-голямо.
- Обикновено се представя като една огромна редица от клетки, като всяка клетка представлява 1 байт.

Създаване на променлива на по-ниско НИВО

- Като създадем променлива(равна на константа)
 - Запазваме дадено количество памет, в която да съхраняваме данните на променливата, като нямаме контрол коя памет да заделим
 - Запазваме името на променливата
 - (Задаваме и стойност)
- Като създадем променлива, равна на друга променлива
 - Запазваме дадено количество памет, в която да съхраняваме данните на променливата, като нямаме контрол коя памет да заделим
 - Запазваме името на променливата
 - Задаваме и стойност, равна на другата променлива

Пример

- `int a = 5;`
- `int b = a;`
- `a = 2;`
- `std::cout<<a<<', '<<b; //2,5`

Абстрактен пример

- Нека паметта представлява клетки с тигри
- Да приемем, че всеки път като създадем променлива, ние сливаме няколко клетки в една, като вътре слагаме и един гледач
- Когато искаме с променливата да се случи нещо, ние все едно казваме на гледача да го направи
- Например:
 - При смяна на стойността казваме на пазача да смени тигрите
 - При увеличаване на стойността казваме на пазача да нахрани тигрите и т.н

Относно абстрактния пример

- Защо примерът е с надзирател? Как това ще ни помогне да решим задачата?
- При извикване на функция винаги се създава нов обект.
- Не можем да променим това.
- Винаги ще се създаде нов гледач, който трябва да бъде сложен в клетка със същия размер като на параметъра.
- Но тогава не можем ли да сложим новия пазач в същата клетка?

Референции

- Синтаксис:

<тип> & <име> = <име на друга променлива от същия тип>

Предназначение:

- Променлива, която притежава данните на вече съществуваща променлива

Пояснение:

- Използвайки предишния пример, все едно в дадена клетка вкарваме още един гледач

Пояснение на пояснението

- Все едно дадената променлива вече има две имена

Пример

```
int a = 5;  
int b = a;  
a = 2;  
std::cout<<a<<', '<<b; //2,5
```

```
int a = 5;  
int& b = a;  
a = 2;  
std::cout<<a<<', '<<b; //2,2
```

Важно

- Също както константите, референцията трябва да се инициализира още при дефиницията
- След като веднъж е била декларирана, инициализирането е необратимо (обвързване за цял живот/гледачът не може да излезе от клетката)
- Типът на референцията и на променливата трябва да съвпадат
- При инициализация не се копират данните на оригиналната променлива ([Demo1](#)) => пести се време и памет при огромни структури от данни

Обратно в началото

Как да накараме функцията да работи?

```
void swap(double a, double b)
{
    double c = a;
    a = b;
    b = c;
}
```

Обратно в началото

Как да накараме функцията да работи?

```
void swap(double & a, double & b)
{
    double c = a;
    a = b;
    b = c;
}
```

- Demo2

Задача от миналия път

- Какво ще изведе следната програма?

```
void Abs(int a)
{
    if(a<0)
        a*=-1;
}
int numb = -5;
Abs(numb);
std::cout<<numb;
```

- Отговор: -5, преговорете си частта със създаването на нови обекти

Задача от миналия път Remastered

- Какво ще изведе следната програма?

```
void Abs(int & a)
{
    if(a<0)
        a*=-1;
}
int numb = -5;
Abs(numb);
std::cout<<numb;
```

- Отговор: 5, вече работи така както искаме

Защо сега работи

- Както казахме, винаги при извикването на функция се създава нов обект, на който присвояваме стойността на оригиналния
- Ако обаче обектът е от тип референция, то ние ще създадем променлива, която използва същите данни (ще вкараме нов гледач в клетката)
- Така вече каквото правим с променливата във функцията, то ние го правим и с оригиналната променлива (всеки от двамата гледачи може да се грижи за тигъра)

Задачи за вас #1

- Отидете на www.menti.com

Задача

Какво ще се изведе на конзолата?

```
double b = 5;
```

```
double & a = b;
```

```
double & c = a;
```

```
--b;
```

```
a += 3;
```

```
std::cout << c;
```

Отговор: 7

Задача

Ще работи ли следният код?

```
void cout(char a){std::cout<<a;}  
void cout(char& a){std::cout<<a;}
```

Отговор: Не, защото ще се получи двусмислие (ambiguity).
Компилаторът няма как да знае към коя от двете функции да се обърне, освен ако не му се подаде литерал.

Задача

Какво ще се изведе на конзолата?

```
int a = 5;
```

```
int & b = a;
```

```
int c = 2;
```

```
b = c;
```

```
b *= 2;
```

```
std::cout << a << '-' << b << '-' << c;
```

Отговор: 4-4-2

Още към абстрактния пример

- След като вече видяхме, че има как да вкараме втори гледач в клетката на тигъра, изникнаха следните проблеми
 1. Трябва да го вкараме в клетката в момента, в който го създадем
 2. След като веднъж сме вкарали гледач в клетка, повече не можем да го местим
 3. Всичко досега беше лесно :D

Решение на проблемите от предния слайд

- Решението на проблемите от предния слайд е някой свръх-квалифициран гледач, който знае как да влезе и излезе от клетката без тигрите да избягат с него
- Такъв гледач трябва да стои пред дадена клетка и да влиза в нея само когато му кажем
- Такъв гледач трябва да може да сменя клетката, пред която стои
- Такъв гледач е звезда и затова го бележим с * и ще го наричаме маниак

Пойнтър

- Тип данна, която като стойност притежава адрес(клетка)
- Съществуват различни пойнтери, които сочат към адресите на различни по тип данни
- Може да съдържа както адреса на някоя lvalue, така и празното пространство (nullptr) или някоя непозволена памет (което е източник на грешки)
- Адресът, който съдържа пойнтерът, може да се променя
- Може да се извършват промени по данните в съответния адрес

Пойнтьър - пояснения

- Нарича се пойнтьър (pointer), защото все едно сочи към мястото в паметта, чийто адрес съхранява.

- Синтаксис:

<тип> *<име> [= <израз>];

- Пример:

```
int * pointer; //маниак, който не стои пред никаква клетка, но е  
              //способен да стои само пред клетки на int
```


Пойнтър – пояснения относно адресите

- Съществува така нареченият нулев адрес (`nullptr`), наричан още пойнтеров литерал
- `nullptr` не сочи към нищо и има специални свойства, с които ще се сблъскате в курсовете по ООП и СДА
- Оператор `&` има и още 1 приложение:
 - Когато се използва като префикс, връща адреса на дадената променлива
 - Пример:
`int a = 5;`
`int * b = &a;`

Пойнтьър – пояснения относно адресите

- Един пойнтьър може да сочи към друг пойнтьър (двоен пойнтьър)

```
int a = 5;
```

```
int * b = &a; //сочи към адреса на a
```

```
int **c = &b; //сочи към адреса на b
```

- Тъй като пойнтьърът е обект, той също притежава адрес!
- Тази концепция отнема време да се разбере, но точно тя е разликата между занаятчията и програмиста

Пойнтьър – пояснения относно адресите

- При двойния пойнтьър абстрактният пример с маниака, който стои пред клетката ,малко увисва, но идеята е същата

```
int a = 5;    //клетка с тигър от тип int
```

```
int * b = &a; //маниак пред клетката на тигъра a
```

```
int **c = &b; //някакъв надзирател, който отговаря за маниака b, но  
             //не знае за коя клетка отговаря b, но може да провери
```

- Важно е да се отбележи, че c няма пряка връзка с a
- c единствено съдържа адреса на b и не знае какво се съдържа в него

Не се предавайте!

- Само още малко суха теория и ще има много примери 😊

Рефериране и дереференциране

- `&<име>` — взимане на адреса на променливата `<име>` (рефериране)
- `*<указател>` — влизане в паметта, към която сочи `<указател>` (дереференциране)
- Да се върнем на примера с маниака и клетките
- Както казахме още в самото начало маниакът може да влиза и да излиза от клетките с лекота
- Да си представим, че `&` е операцията за излизане от клетката, а `*` е операцията за влизане в клетката, а в нормално състояние маниакът е извън клетката

Рефериране и дереференциране - примери

```
int cage1 = 5;           //клетка с тигри от тип int, която се казва cage1
int *tarzan = &cage1;    //маниака tarzan, който стои пред клетка cage1
int *maugli = tarzan;    //маниака maugli, който стои пред
                        //клетката, пред която стои tarzan => cage1

(*maugli)++;            // казваме на maugli да влезе в клетката и да
                        // увеличи стойността на cage1 с 1
```

!Важно рефериране и дереференциране са 3ти по приоритет в таблицата с приоритети, затова трябва да се укаже със скоби, че първо искаме да влезем в клетката и чак след това да действваме!

Precedence	Operator	Description
1	::	Scope resolution
2	++ -- type() type{} () [] . ->	Suffix/postfix increment and decrement Function-style type cast Function call Array subscripting Element selection by reference Element selection through pointer
3	++ -- + - ! ~ (type)	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style type cast
	* &	Indirection (dereference) Address-of
	sizeof new, new[] delete, delete[]	Size-of Dynamic memory allocation Dynamic memory deallocation
4	.* ->*	Pointer to member
5	* / %	Multiplication, division, and remainder
6	+ -	Addition and subtraction
7	<< >>	Bitwise left shift and right shift
8	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively
9	== !=	For relational = and ≠ respectively
10	&	Bitwise AND
11	^	Bitwise XOR (exclusive or)
12		Bitwise OR (inclusive or)
13	&&	Logical AND
14		Logical OR
15	?: = += -= *= /= %= <<= >>= &= ^= =	Ternary conditional Direct assignment (provided by default for C++ classes) Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR
16	throw	Throw operator (for exceptions)
17	,	Comma

Рефериране и дереференциране - примери

```
int cage2 = *tarzan; //нова клетка, която има стойността тази клетка,  
                    //пред която стои tarzan, tarzan трябва да влезе в  
                    //клетката, за да каже какво се съдържа в нея  
maugli = &cage2;    //maugli вече стои пред клетка cage2  
*tarzan = 1;         //tarzan влиза в клетката си (cage1) и прави така,  
                    //че стойността и да стане 1  
*maugli = *tarzan;  //maugli и tarzan си влизат в клетките и тарзан  
                    //казва каква е стойността в неговата, а maugli  
                    //прави така, че и в неговата стойността да е  
                    //такава като в на tarzan
```


Рефериране и дереференциране

- При рефериране (&) се взима адреса на съответния елемент, затова е задължително да се работи с lvalue (не можем да вземем адреса на константа)
- Адресът съответно е константа и не можем да го променим
- При дереференциране приемаме като параметър някакъв адрес, който е константа
- След като влезем в съответния адрес, имаме достъп до променливата, която е lvalue

Рефериране и дереференциране

- $\&\langle lvalue \rangle$ връща като резултат $\langle rvalue \rangle$!
 - Следните операции са невалидни:
 - $\&3$ – рефериране на константа
 - $\&x = 1$ - присвояване на константа за стойност на референция
- $*\langle rvalue \rangle$ връща като резултат $\langle lvalue \rangle$!
- операциите са дуални една на друга и се унищожават взаимно
 - $\&(*p) \iff p$
 - $*(\&x) \iff x$

Пойнтъри към константи и константни пойнтьри

- `const int * == int const *` - пойнтьр към константа
- Пойнтьрите към константа са същите като обикновените пойнтьри, с тази разлика, че не може да се променя стойността на променливата, към която сочат (дори маниакът да влезе в клетката той не може да променя нищо, а само да казва какво става вътре)
- `int * const` – константен пойнтьр
- Константните пойнтьри са като референциите, но са пойнтьри (маниакът си стои пред клетката, но не може да я сменя с друга)

Други практики, заслужаващи преглед

- `const int * const == int const * const` – константен поинтър към константа
- `int ** const` – константен поинтър към поинтър от тип `int`
- `int * const *` - поинтър към константен поинтър към `int`
- `int const **` - двоен поинтър към константа от тип `int`
- `int * const * const` – константен поинтър към константен поинтър към `int`
- `const int * const * const` – константен поинтър към константен поинтър към константа от тип `int`

Задачи за вас #2

- Отидете на www.menti.com

Задача

- Кой от дадените изрази е верен?

A) `const int a = 5;`
`int * b = &a;`

B) `int c = 5;`
`const int * d = &c;`

Отговор: B, защото поинтърът може да третира неконстантна променлива като константна, докато обратното е невъзможно

Задача

- Кой от дадените изрази е неверен?

A) `int a = 5;`
 `int const* b = &a;`
 `int c = 3;`
 `b = &c;`

B) `int d = 5;`
 `int const* e = &d;`
 `int f = 3;`
 `*e = f;`

- Отговор: B, защото пойнтьърът третира неконстантна променлива като константна и макар d да не е константна, не можем да я модифицираме през e

Задача

- Кой от дадените изрази е верен?

A) `const int a = 5;`
`int *const b = &a;`

B) `int d = 5;`
`int *const e = &d;`
`*e = 5;`

- Отговор: B, защото пойнтърът е константен и не може да се променя накъде да сочи, но може да се променя това, към което сочи

Почивка 15 минути

- След като взехме основите, вече можем да преминем към истинските предизвикателства 😊

Пойнтъри, референции и функции 3 в 1

- Да повторим какво става ако имаме референция като параметър
- Какво става, ако имаме пойнтър като параметър
- Какво става, ако имаме референция към пойнтър като параметър
- Какво става, ако функцията връща референция
- Какво става, ако функцията връща пойнтър
- Някои добри практики, с които ще е по-трудно да се гръмнете в крака по невнимание

Референция като параметър

- Създава се нов обект, чиито данни са на адреса на формален параметър
- Няма копиране на данни
- Тъй като новият обект е пряко свързан с оригиналния, то каквито и промени да му направим, ние променяме и оригиналния

Пойнтър като параметър

- Създава се нов обект, който копира информацията на формалния параметър
- Този нов обект сочи към същия адрес като оригиналния
- Ако извършим промени в адреса, към който сочи новият обект, то ние ще променим информацията там (същата информация, към която сочи и оригиналният пойнтер)
- Ако сменим адреса, към който сочи новият пойнтер, няма да сменим адреса, към който сочи оригиналният пойнтер, защото двата обекта са различни и не са пряко свързани

Референция към поинтър като параметър

- Поинтърът е обект, който си има адрес и стойност => няма причина да няма референция към поинтър
- Синтаксис <тип> * & <име> = <lvalue>
- Аналогично както при всички референции, щом се създаде референция към поинтър, то вече има 2 начина да се обърнем към един и същи обект
- Ако променим към какво сочи поинтърът през което и да е име, променяме към какво сочи самият обект
- Ако имаме референция към поинтър като формален параметър, то важат абсолютно същите правила, за които говорихме досега

Функция, връщаща референция

- Доста tricky елемент за УП
- Много е лесно да се простреляте в крака на този етап, но ще ви го покажа, защото в бъдеще ще ви е полезно
- Когато връщате референция, вие не връщате стойността на променливата, а цялата променлива
- Трябва да сте сигурни, че променливата, чиято референция връщате, съществува и след приключването на функцията, тоест не връщате локално създаден обект
- Demo3

Функция, връщаща референция

- Пример за грешна функция връщаща референция е

```
int & errorProne()  
{  
    int a = 5;  
    return a;  
}
```

Недефинирано поведение, което компилаторът на Visual Studio, любезно заличава, но реално това е проблем и не всички компилатори го позволяват

Функция, връщаща поинтър

- Поинтърът е най-обикновен обект => когато една функция връща поинтър, тя връща нов обект, който има за стойност адреса, към който сочи оригиналният поинтър
- Щом се връща нов обект, то той не е свързан с оригиналния, но въпреки това сочат към един и същи адрес и ако се извърши дереференциране (влизане в клетката), то ще бъде променена информацията в адреса, който съдържат и двата поинтъра

Някои добри практики, с които ще е по-трудно да се гръмнете в крака по невнимание

- Можете да връщате референции и пойтъри към константи, както и да подавате такива формални параметри във функция
- Винаги създавайте константи, когато нямате намерение да промените обекта, липсата на 1 константа може да ви коства много време и главоболия!
- Референциите към примитивни типове данни и пойнтъри не спестяват толкова много време и памет колкото си мислите, затова не е добра практика да ги използвате, освен ако не искате да промените оригиналните обекти

Задачи за вас #3

- Отидете на www.menti.com

Задача

Посочете невалидното(невалидните):

```
int tmp = 5;
```

A) `int * a = &tmp;`

B) `int *& b = a;`

C) `int **&c = &a;`

D) `int *&d = b;`

E) `int &*e= tmp;`

F) `int *f = b;`

Отговор: C) и E)

Задача - пояснение

Кой/Кои от следните примери са невалидни?

```
int tmp = 5;
```

- A) `int * a = &tmp;` //поинтър към tmp
- B) `int *& b = a;` //референция към поинтър равна на поинтъра a
- C) `int **&c = &a;` //&a е rvalue, а не е обект => не може да се реферира
//не можем да създадем референция към rvalue
- D) `int *&d = b;` //(int *)& референция към поинтър b, виж F)
- E) `int &*e= tmp;` //(int&)* pointer to reference is not allowed
- F) `int *f = b;` //макар и референция b си е стандартен поинтър

Отговор: C) и E)

Задача

Ще се компилира ли следният код и ако да, то какво ще се изведе?

```
void shaker(const int & aRef, int const * bPtr)
{
    int a = aRef;
    a+=50;
    bPtr = &a;
}
int a =5;
int * b = &a;
shaker(a, b);
std::cout<<a;
```

Отговор: Да, ще се изведе 5

Задача - пояснение

Ще се компилира ли следният код и ако да, то какво ще се изведе?

```
void shaker(const int & aRef, int const * bPtr)
{ //създаваме нови обекти от тип int и int *
    int a = aRef; //нов обект, на който присвояваме стойността на истинското a
    a+=50; //обработка над временния обект
    bPtr = &a; //новият обект поинтър сочи временния обект int, не променяме оригиналния
}

int a =5;
int * b = &a;
shaker(a, b);
std::cout<<*b; //оригиналните обекти са непроменени
```

Отговор: Да, ще се изведе 5

Задача

Ще се компилира ли следният код и ако да, то какво ще се изведе?

```
void shaker(const int & aRef, int const *& bPtr)
{
    int a = aRef;
    bPtr = &a;
    *bPtr += 50;
}

int a =5;
const int * b = &a;
shaker(a, b);
std::cout<<*b;
```

Отговор: Не, защото bPtr е поинтър към константа

Задача - пояснение

Ще се компилира ли следният код и ако да, то какво ще се изведе?

```
void shaker(const int & aRef, int const *& bPtr) //bPtr, също като b е поинтър към константа
{
    int a = aRef;
    bPtr = &a;
    *bPtr += 50; //за bPtr a е константа => не може да променя стойността и
}

int a =5;
const int * b = &a;
shaker(a, b);
std::cout<<*b;
```

Отговор: Не, защото bPtr е поинтър към константа

Pointer arithmetic

- За разлика от референциите, както вече казахме пойнтьрите могат да променят адреса, към който сочат
- На създателите на езика им е хрумнала идеята, че може да има няколко съседни клетки от един и същи тип една до друга
- Тъй като пойнтьрът знае към какъв тип променлива сочи, той знае точно колко байта да се измести в паметта (наляво или надясно), за да стигне до следващата или по-следващата клетка
- Така се появяват и тъй наречените Pointer arithmetic, които позволяват извършване на аритметични операции с пойнтьри

Размер на примитивните типове данни

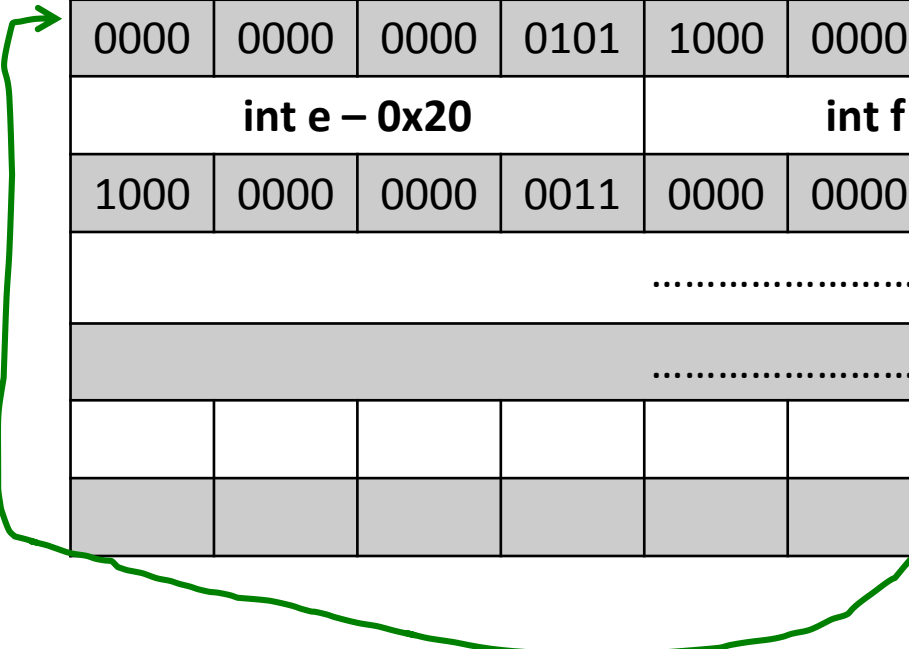
*Информацията е валидна за x86 VC компилатор

- 1 byte – bool, char
- 2 bytes – short (int)
- 4 bytes – int, long, float, enum
- 8 bytes – long long, double, long double
- Размерът на пойнтъра съвпада с този на типа на променливата
(при x64 VC компилатор пойнтьрите са двойно по-големи)

Pointer arithmetic

- Нека да създадем поинтър `int* ptr`, който сочи към `a`.

int a – 0x10				int b – 0x14				int c – 0x18				int d – 0x1C			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				int f – 0x24				int g – 0x28				int h – 0x2C			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
.....															
.....															
					int * ptr – 0xA16										
					0x10										



Pointer arithmetic

- Тъй като ptr знае, че сочи към int, то за да го накараме да сочи към съседния(тоест следващите 4 клетки), трябва да му кажем премести се с 1 клетка: ptr+1

int a – 0x10				int b – 0x14				int c – 0x18				int d – 0x1C			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				int f – 0x24				int g – 0x28				int h – 0x2C			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
.....															
.....															
					int * ptr – 0xA16										
					0x10										

Pointer arithmetic

- Можем и тотално да сменим адреса, към който сочи ptr: `ptr+=1`

int a – 0x10				int b – 0x14				int c – 0x18				int d – 0x1C			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				int f – 0x24				int g – 0x28				int h – 0x2C			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
.....															
.....															
					int * ptr – 0xA16										
					0x14										

Pointer arithmetic

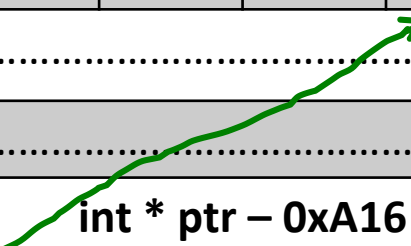
- Можем да действаме и по-смело $\text{ptr} += 6$

int a – 0x10				int b – 0x14				int c – 0x18				int d – 0x1C			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				int f – 0x24				int g – 0x28				int h – 0x2C			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
.....															
.....															
					int * ptr – 0xA16										
					0x2C										

Pointer arithmetic

- Разбира се, можем и да се връщаме назад: --ptr

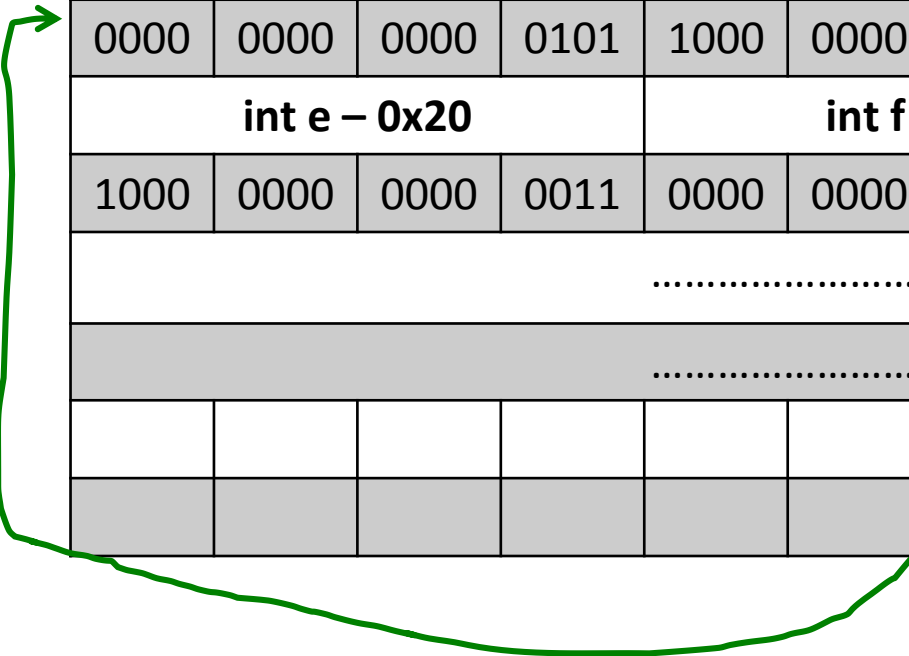
int a – 0x10				int b – 0x14				int c – 0x18				int d – 0x1C			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				int f - 0x24				int g – 0x28				int h – 0x2C			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
.....															
.....															
					int * ptr – 0xA16										
					0x28										



Pointer arithmetic

- Да се върнем в началото: `ptr -= 6`

int a – 0x10				int b – 0x14				int c – 0x18				int d – 0x1C			
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000	0000	0000	0000	1111
int e – 0x20				int f – 0x24				int g – 0x28				int h – 0x2C			
1000	0000	0000	0011	0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
.....															
.....															
					int * ptr – 0xA16										
					0x10										



Pointer arithmetic

- За да обходим всичките клетки и да изпишем какво съдържат, можем да напишем следния код:

```
for(unsigned i = 0; i<8;++i)
{
    std::cout<<*(ptr + i)<<' ';
}
```

- Изглежда ли ви познато?

Почивка 10 минути

- Hint: починете си добре 😊

Масиви

- Масивът е съставен тип данни
- Представя крайни редици от елементи
- Всички елементи са от един и същи тип
- Позволява произволен достъп до всеки негов елемент по номер (индекс)

Синтаксис

- `<тип> <идентификатор> [[<константа>] [= { <константа> [, <константа>] }]] ;`
- Примери:
 - `bool b[10];`
 - `double x[3] = { 0.5, 1.5, 2.5 }, y = 3.8;`
 - `int a[] = { 3 + 2, 2 * 4 }; \Leftrightarrow int a[2] = { 5, 8 };`
- За всички фенове на Java и C#
`bool[10] b;` е невалиден израз

Масиви и пойнтери

- Може да се каже, че масивът е по-специален константен пойнтер
- Реално масивът е пойнтер към 0вия елемент в поредицата, но има и по-специални свойства като:
 - запазване на последователни блокове памет при инициализация
 - знание колко му е размера
- Операцията [`<число>`] работи и за пойнтери и е равносилна на `*(<пойнтер/име_на_масив> + число)`
- Demo4 и Demo5

Операции за работа с масив

- Достъп до елемент по индекс: `<масив>[<цяло_число>]`
- Примери: `x = a[2];` (rvalue) `a[i] = 7;` (lvalue!)
- Броенето на индексите започва от 0
- Оператор `sizeof()` връща разликата на първата и последната клетка

Операции за работа с масив

- За масив от тип `int`, който съдържа 5 елемента, `sizeof()` ще ви върне 20, защото в масива има 20 клетки ($5 * 4$)
- За да вземете големината на целия масив, трябва да използвате следната хитринка: `sizeof(<име>)/sizeof(<тип>)`
- Внимание: няма проверка за коректност на индекса!

Операции за работа с масив

- Няма присвояване `a = b`
- Няма поелементно сравнение `a == b` винаги връща `false` ако `a` и `b` са различни масиви, дори и да имат еднакви елементи
- Няма операции за вход и изход `std::cin >> a; std::cout << a;`
- `std::cout << a;` извежда адреса на `a` (не важи за символен низ)

Операции за работа с масив

- Не можете да подадете масив като параметър на функция
- Можете да имате като параметър:
 - <име>[]
 - <име>[<число>]
 - пойнтьр
- Който и от трите метода да изберете, ще получите едно и също

Операции за работа с масив

- Не можете да направите функция, която връща масив
- Можете да направите функция, която връща поинтър към първия елемент на масив
- Работата с масиви в C++ понякога е сложна и ограничаваща, затова в стандартната библиотека има много различни имплементации, които много улесняват програмиста
- За да ги оцените подобаващо и да ги разберете, трябва първо да се поизцапате в калта

Двумерни масиви

- Така както поинтър може да сочи към поинтър, то може да има и масив от масиви
- Аналогията не е случайна, защото `int **` има същата логика като `int [][]`
- `int a [5][4]` - масив, който съдържа 5 масива, които съдържат 4 елемента от тип `int` (все едно имаме масив от поинтъри)
- `int **b = a` - поинтър, който сочи към поинтър сочещ към `int`

Двумерни масиви

- Също както при обикновените масиви, щом имаме масив от пойнтьори, то те са един до друг в паметта, но не е нужно това, към което сочат, също да е последователно в паметта
- Demo6

Двумерен масив

- Нека видим примерно представяне на двумерен масив `int arr[2][3]` (два масива с по 3 елемента/два поинтъра към `int`)

Int a – 0x10				int b – 0x14				int c – 0x18							
0000	0000	0000	0101	1000	0000	0000	0001	0000	0000	0000	1000				
				int f - 0x24				int g – 0x28				int h – 0x2C			
				0000	0000	0001	0000	0000	0000	0000	0000	1000	0000	0000	0010
.....															
.....															
					int arr[2][3] – 0xA16				int * - 0xA1A						
					0x10				0x24						

Многомерни масиви

- Защо да спираме само с двумерни масиви?
- Реално можем да имаме n -мерни масиви
 - Двумерните масиви представляват някаква таблица - [Demo7](#)
 - Тримерните масиви са като някакъв паралелепипед
 - N -мерните – илюминати
- Подаването на многомерни масиви като формални параметри може да е tricky, затова избягвайте да го правите

Задачи за вас #4

- Отидете на www.menti.com

Задача

- Можем ли да кажем към елемент на кой от двата масива сочи p?

```
int a[7] = {1,1,1,1,1,1,1};
```

```
int b[7] = {1,1,1,1,1,1,1};
```

```
int * p;
```

```
.....
```

```
std::cout<<*p;      //извежда 1
```

Отговор: Да. За да го направим можем да проверим дали адреса на някоя от клетките съвпада с този, към който сочи p

```
for(unsigned i=0; i<7; ++i)
```

```
    &(a[i])==p;
```

Задача

- Какво ще се изведе на конзолата?

```
int a[7] = { 1,1,2,1,1,1,1 };
```

```
std::cout<< (*(&a[0] + 2) == 2);
```

Отговор: 1, защото true и false се извеждат чрез числените им стойности

$\&a[0]$ е равно на $a+0 \Rightarrow a+0+2 == a[2]$, а $a[2] == 2$

Задача

Какво ще се изведе на конзолата?

```
int a[7] = { 1,2,3,4,5,6,7 };
```

```
int b[7] = { 7,6,5,4,3,2,1 };
```

```
int sum1 = 0;
```

```
int sum2 = 0;
```

```
for (int i = 1; i < 7; i++){
```

```
    sum1 += *(a + i);
```

```
    sum2 += b[i];
```

```
}
```

```
std::cout << (sum1 == sum2);
```

Отговор: 0, защото броенето започва от 1ви индекс => 2рия елемент

Символен низ

- Описание: Символен низ наричаме последователност от символи (последователност от 0 символи наричаме празен низ)
- Представяне в C++: Масив от символи (char), в който след последния символ в низа е записан терминиращият символ '\0'

Относно '\0'

- Първият символ в ASCII таблицата, с код 0
- Използва се като прекъсвач(терминатор) от много функции за символни низове, за да се определя края на низа

- Може да се сложи в средата на масив от символи

`char a = {'H', 'e', 'l', 'l', '\0', 'o'}; //символният низ е "Hell"`

Символен низ

- Примери:
- `char word[] = { 'H', 'e', 'l', 'l', 'o', '\0' };`
- `char word[6] = { 'H', 'e', 'l', 'l', 'o' };`
- `char word[100] = "Hello";`
- `char word[5] = "Hello";` //валиден масив е, но не е символен низ
- `char word[6] = "Hello";`
- `char word[5] = { 'H', 'e', 'l', 'l', 'o' };`

Готини неща относно символните низове

- Вход (>>, `cin.getline(<низ>)`) и изход (<<) вече работят както се очаква
- Библиотеката `<cstring>` съдържа готови функции, които много улесняват работата с низове:
 - `strlen(<низ>)` връща колко символа има от началото до `'\0'`:
 - `char word[100] = "Hello";`
 - `std::cout<<strlen(word)` ще изведе 5

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Важни неща относно ASCII на този етап

- Може да се извършват математически операции със символи (търпение, скоро ще дефинираме и какво са мат. операции)
- За да преобразувате символ число в число, от символа трябва да извадите 48 или символа '0'
 - $'9' - 7 = 50$, защото '9' има числена стойност 57
 - $'9' - '0' - 7 = 57 - 48 - 7 = 2$
- Главните букви са преди малките
- Разстоянието между малка и главна буква е $2^5 = 32$

ИЗТОЧНИЦИ

- Голяма част от информацията е сверена с <https://en.cppreference.com>
- Използвани са дефиниции и описания от материали на доц. Трифон Трифонов
- Авторският код е проверяван на [VisualStudio2017](#)