3

Práce s polem a pamětí

Inicializace jednorozměrného pole


array1d

Jednorozměrné pole lze inicializovat přímo v deklaraci.

```
int array[LENGTH] = {1, 5, 8, 9};
```

Prvky pole umístíte do složených závorek a oddělíte je čárkou. V příkladě je `LENGTH` makro definované hodnotou 4. Inicializujete-li všechny prvky pole (což je v uvedeném příkladě), nemusíte udávat v hranatých závorkách velikost pole. Bude doplněná automaticky podle počtu inicializačních prvků. Je tedy možné napsat i zápis:

```
int array[] = {1, 5, 8, 9};
```

Inicializace vybraných prvků jednorozměrného pole


array1d

Jednorozměrné pole lze inicializovat přímo v deklaraci, nemusíte ale inicializovat všechny prvky pole. Lze určit, které prvky inicializovat jakou hodnotou. Ostatní prvky budou mít hodnotu 0.

```
int array2[LENGTH] = {[0] = 1, [3] = 5};
```

První prvek s indexem 0 bude inicializován hodnotou 1. Poslední prvek s indexem 3 bude inicializován hodnotou 5. Ostatní prvky budou mít hodnotu 0.

Poznámka: Tuto konstrukci podporují pouze překladače jazyka C implementující normu jazyka C roku 1999 označovanou jako C99. Uvedená konstrukce není součástí normy jazyka C++ a nezvládne ji například současná verze překladače Microsoft Visual C++.

Inicializace dvourozměrného pole


matrixC

Dvourozměrné pole lze stejně jako jednorozměrné pole inicializovat přímo v deklaraci.

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

Prvky pole umístíte do složených závorek a oddělíte je čárkou. Takto deklarované dvourozměrné pole je v paměti uloženo jako souvislý blok. Prvky dvourozměrného pole jsou v paměti ukládány postupně za sebou, jak je vidět na obrázku 3.1. Platí pravidlo, že rychleji se mění index více napravo.

Výše uvedená inicializace není chyba, ale dvourozměrné pole lze inicializovat i jiným způsobem. Dvourozměrné pole je pole jednorozměrných polí. Při inicializaci můžete jednotlivá jednorozměrná pole uzavřít do závorek.

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

1	2
3	4
5	6

1	2	3	4	5	6

Obrázek 3.1 Uložení dvourozměrného pole v paměti

matrixC

Součet matic

Součet dvou matic ukazuje přístup k prvkům pole pomocí indexů. Součet matic (dvourozměrných polí) je součet prvků na stejných pozicích. Označují-li identifikátory A, B a C matice a A_{ij} je prvek v i-tém řádku a j-tém sloupci matice A, potom matice C = A + B je matice, pro jejíž prvky platí $C_{ij} = A_{ij} + B_{ij}$.

```
for(i = 0; i < LENGTHX; i++){
  for(j = 0; j < LENGTHY; j++){
    c[i][j] = a[i][j] + b[i][j];
  }
}
```

Upozornění: Sčítat lze jen matice stejného typu (musí mít stejný počet řádků i sloupců).

matrixC

Součin matic

Součinem matic je matice, jejíž počet sloupců je roven počtu sloupců první matice a počet řádků je roven počtu řádků druhé matice. Pro každý prvek výsledné matice platí $C_{ij} = A_{i1} * B_{1j} + A_{i2} * B_{2j} + \ldots + A_{in} * B_{nj}$, kde n je počet sloupců první matice.

```
for (i = 0; i < LENGTHX; i++){/*Počet sloupců první*/
  for(j = 0; j < LENGTHX; j++){/*Počet řádků druhé*/
    for(k = 0; k < LENGTHY; k++){
      e[i][j] += b[i][k] * d[k][j];
    }
  }
}
```

Upozornění: Násobit matice lze jen v případě, že první matice má stejný počet řádků, jako má druhá matice počet sloupců.

Práce s dynamickou pamětí v C

dynamicArrayC1d

Jazyk C umožňuje alokovat paměť (požádat operační systém o přidělení paměti) pomocí funkce `malloc` a také dealokovat (uvolnit, vrátit paměť operačnímu systému) pomocí funkce `free`.

```
if ((block = (int *)malloc(LENGTH * sizeof(int))) == NULL){
...
}
...
free(block);
```

Dříve alokovanou paměť operační systém znovu nepřidělí, dokud nebude dealokována. Alokací paměti získá váš program vyhrazené místo v paměti, které nebude přiděleno nikomu jinému, dokud jej neuvolníte. Funkce `malloc` alokuje vždy souvislý blok v paměti. K alokované paměti lze přistupovat pomocí ukazatelů.

Funkce `malloc` má jako parametr počet bajtů, které chcete alokovat. Funkce `malloc` vrací ukazatel na `void`. Vrácený ukazatel odkazuje na začátek alokovaného bloku paměti a můžete jej samozřejmě přetypovat. Jestliže se alokace paměti nezdaří, je návratová hodnota NULL.

Funkce `free` má jako svůj parametr ukazatel na začátek alokovaného bloku, který chcete uvolnit. Každou paměť, kterou alokujete, musíte také dealokovat (uvolnit).

Funkce `malloc` i `free` jsou deklarovány v hlavičkovém souboru `stdlib.h`.

Změna velikosti alokovaného bloku paměti

reallocC

Máte-li potřebu změnit velikost alokovaného bloku paměti, můžete využít funkci `realloc`, která je deklarována v hlavičkovém souboru `stdlib.h`. Tímto způsobem lze zvětšit nebo zmenšit dynamicky alokované pole.

```
if ((block = (int *)malloc(LENGTH * sizeof(int))) == NULL){
...
}
if ((block = (int *)realloc(block, LENGTH * 2 * sizeof(int))) == NULL){
...
}
```

První parametr funkce `realoc` je ukazatel na začátek alokovaného bloku paměti, jehož velikost chcete změnit. Druhým parametrem je nová velikost alokovaného bloku paměti v bajtech. Funkce vrací ukazatel na počátek bloku paměti o nové velikosti.

Návratová hodnota může, ale nemusí být stejná jako první parametr. Je-li to možné, bude původní blok paměti zvětšen nebo zmenšen a vrácený ukazatel bude stejný jako první parametr. Jestliže nebude možné alokovaný blok paměti rozšířit (například za alokovaným blokem je alokován jiný blok), funkce `realloc` alokuje nový blok, překopíruje data ze starého bloku a starý blok uvolní (dealokuje). Poté bude vrácený ukazatel jiný než první parametr.

dynamicArrayC1d

Přístup k paměti pomocí ukazatelové aritmetiky

Máte-li ukazatel odkazující se na alokovaný blok paměti (nemusí být nutně alokován dynamicky pomocí `malloc`, `realloc`, `new` a podobně), lze k datům v paměti přistupovat pomocí takzvané *ukazatelové aritmetiky*.

```
int *block = NULL;
if ((block = (int *)malloc(LENGTH * sizeof(int))) == NULL){
...
}
for(temp = block; temp - block != LENGTH; temp++, p *= 5){
  *temp = p;
}
```

Počítání s ukazateli má následující pravidla:

1. Součet ukazatele (u) a celého čísla (c) je ukazatel ukazující na místo v paměti, které leží za místem, na které ukazoval ukazatel u, o počet bajtů, který je roven součinu čísla c a velikostí datového typu, na který se ukazatel odkazuje. Lze použít zkrácené zápisy, kde `temp += 10` je obdoba `temp = temp + 10` nebo `temp++` je obdoba `temp = temp + 1`. V příkladu je použit výraz `temp++`, kde temp je ukazatel na `int`, což znamená, že ukazatel `temp` se odkazuje o jedno číslo typu integer dále než před vykonáním výrazu. Představte si, že ukazatel ukazuje na nějaký prvek v poli. Potom přičtení čísla k ukazateli je posunutí ukazatele o číslem daný počet prvků „doprava“, takže součet ukazatele a čísla bude ukazatel ukazující na prvek, jehož index je oproti indexu původního prvku vyšší o dané číslo.
2. Rozdíl ukazatele (u) a čísla (c) je ukazatel odkazující před ukazatel u o počet bajtů, který je roven součinu čísla c a velikostí datového typu, na který se ukazatel odkazuje. Lze použít zkrácené zápisy, kde `temp -= 10` je obdoba `temp = temp - 10`, nebo `temp--` je obdoba `temp = temp - 1`. Představte si, že ukazatel ukazuje na nějaký prvek v poli. Potom odečtení čísla od ukazatele je posunutí ukazatele o číslem daný počet prvků „doleva“. To znamená, že rozdíl ukazatele a čísla bude ukazatel ukazující na prvek, jehož index je oproti indexu původního prvku nižší o dané číslo.
3. Rozdíl dvou ukazatelů je celé číslo udávající počet prvků, které se vejdou do paměti mezi místa, na která ukazují tyto dva ukazatele. Prvky jsou typu, na které se ukazatel odkazuje. V příkladu je použit výraz `temp - block`; vzhledem k tomu, že se jedná o ukazatele na `int`, je výsledkem číslo udávající, o kolik celých čísel je ukazatel `temp` za ukazatelem

`block`. Představte si, že oba ukazatele ukazují na nějaké prvky v poli. Potom rozdíl ukazatelů je rozdíl indexů obou prvků.

Ostatní aritmetické operace (prováděné aritmetickými operátory) s ukazateli nedávají smysl. Ukazatele můžete ještě porovnávat pomocí relačních operátorů (==, !=, <, <=, >, >=).

Přístup k paměti pomocí operátoru [] – dynamicky alokované jednorozměrné pole v C

dynamicArrayC1d

V jazycích C a C++ jsou pole a ukazatel velmi blízké pojmy. Pole se vždy (s výjimkou tří situací) konvertuje ve výrazech na ukazatel na první prvek v poli. Výjimky, při nichž nedochází ke konverzi pole na ukazatel na první prvek v poli:

1. použití pole v operátorech `sizeof`,
2. při použití operátoru & na identifikátor pole (operátor & vrátí typ „adresa pole“, nikoli typ „adresa prvního prvku v poli“),
3. použití pole k inicializaci reference.

```
int *block = NULL;
if ((block = (int *)malloc(LENGTH * sizeof(int))) == NULL){
...
}
for(p = 0; p < LENGTH; p++){
  printf("%d\n", block[p]);
}
free(block);
```

Identifikátor `blok` je sice definován jako ukazatel, ale klidně s ním můžete pracovat jako s polem.

Poznámka: Protože lze na ukazatel použít operátor [], můžete uvedený příklad chápat jako ukázku dynamicky alokovaného pole.

Upozornění: Zapisovat nebo číst nealokovanou paměť (stejně jako zápis nebo čtení prvků pole s indexem vyšším, než je maximální index) vede k nekorektnímu chování programu.

Dynamická alokace dvourozměrného pole v C

dynamicArrayC2d

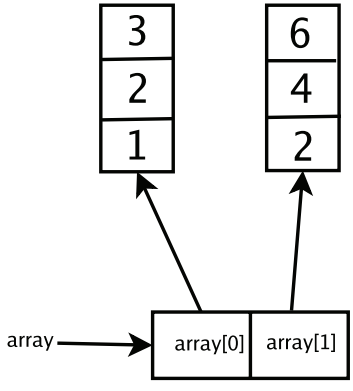
Dynamicky alokované (jeho rozměry budou známé až při běhu programu, nikoliv při psaní zdrojového textu) dvourozměrné pole lze v jazycích C a C++ vytvořit jako jednorozměrné pole, jehož prvky budou opět pole.

```
int **array;
if ((array = (int **)malloc(LENGTHX * sizeof(int *))) == NULL){
...
}
```

```
for (i = 0; i < LENGTHX; i++){
  if ((array[i] = (int*)malloc(LENGTHY * sizeof(int)))
       == NULL){
    printf("Problém s alokací paměti\n");
    while(--i > 0){
      free(array[i]);
    }
    free(array);
    return -1;
  }
}
/* Zde je nějaká práce s polem */
for(i = 0; i < LENGTHX; i++){
  free(array[i]);
}
free(array);
```

Identifikátor array označuje ukazatel na ukazatel na int. Práce s dynamickým dvourozměrným polem by se dala popsat v pěti bodech:

1. Nejprve musíte alokovat jeden rozměr dvourozměrného pole. Bude se jednat o pole ukazatelů na typ `int`. Všimněte si, že výsledek, který vrací funkce `malloc`, musíte v C++ přetypovat na ukazatel na ukazatel na `int (int**)`. Také si všimněte, že velikost alokovaného bloku v paměti je součin velikosti jednoho rozměru dvourozměrného pole a **velikosti ukazatele** na typ (nikoliv velikosti samotného typu).
2. Dále musíte alokovat pro každý prvek pole ukazatelů paměťový prostor reprezentující druhý rozměr dvourozměrného pole. Nyní již výsledek, který vrací funkce `malloc`, přetypujete na ukazatel na daný typ. Velikost alokovaného bloku paměti bude rovna součinu druhého rozměru dvourozměrného pole a velikosti datového typu.
3. Nyní můžete pracovat s dvourozměrným polem. Můžete využívat operátory `[]` pro indexaci pole. Například: `array[i][j] = (i + 1) * (j + 1);`
4. Postupně v cyklu uvolníte všechny alokované bloky paměti, které jste alokovali v bodě 2.



Obrázek 3.2 Uložení dvourozměrného pole v paměti

5. Uvolníte ukazatel odkazující se na ukazatel odkazující se na typ `int`. Paměť jste alokovali v prvním kroku.

Způsob uložení dynamického dvourozměrného pole (pole polí) v paměti je zobrazen na obrázku 3.2.

Upozornění: Takto alokované dynamické dvourozměrné pole není souvislý blok paměti tak jako statické dvourozměrné pole.

Alokace a inicializace dvourozměrného pole s různě dlouhými řádky

triangleArrayC

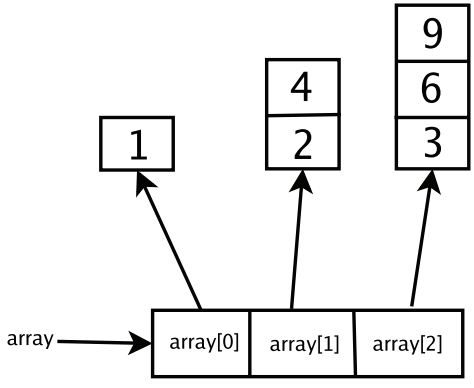
Ukazatel na ukazatel lze použít nejen pro práci s dvourozměrným dynamickým polem, ale také pro práci s dvourozměrným polem, ve kterém každý řádek bude mít různě dlouhou délku. Oproti klasické alokaci dvourozměrného pole budete pro každý řádek alokovat blok paměti jiné velikosti.

```
int **array;
if ((array = (int **)malloc(LENGTH * sizeof(int *))) == NULL){
...
}
for (i = 0; i < LENGTH; i++){
  if ((array[i] = (int*)malloc((i + 1) * sizeof(int))) == NULL){
    printf("Problém s alokací paměti\n");
    while(--i > 0){
      free(array[i]);
    }
    free(array);
    return -1;
  }
}
for(i = 0; i < LENGTH; i++){
  for(j = 0; j < i + 1; j++){
    array[i][j] = (i + 1) * (j + 1);
    printf("[%d][%d] = %d\n", i, j, array[i][j]);
  }
}
for(i = 0; i < LENGTH; i++){
  free(array[i]);
}
free(array);
```

Identifikátor `array` je ukazatel na ukazatel odkazující se na číslo `int`. Práce s dynamickým dvourozměrným polem, které má různě dlouhé řádky (nebo sloupce, záleží jak si pole otočíte ve svých představách), by se dala popsat v pěti bodech:

1. Nejprve musíte alokovat jeden rozměr dvourozměrného pole. Bude se jednat o pole ukazatelů na typ `int`. Všimněte si, že výsledek, který vrací funkce `malloc`, musíte přetypovat na ukazatel na ukazatel na `int (int**)`. Také si všimněte, že velikost alokovaného bloku v paměti je součin velikosti jednoho rozměru dvourozměrného pole a **velikosti ukazatele** na typ (nikoliv velikosti samotného typu).
2. Dále musíte alokovat pro každý prvek pole ukazatelů paměťový prostor reprezentující druhý rozměr dvourozměrného pole. Nyní již výsledek, který vrací funkce `malloc` přetypujete na ukazatel na daný typ. Velikost alokovaného bloku paměti bude rovna součinu velikosti konkrétního řádku (nebo sloupce) dvourozměrného pole a velikosti datového typu. V příkladu je dvourozměrné pole trojúhelníkového tvaru.
3. Nyní můžete pracovat s dvourozměrným polem. Můžete využívat operátory `[]` pro indexaci pole. Například: `array[i][j] = (i + 1) * (j + 1);`
4. Postupně v cyklu uvolníte všechny alokované bloky paměti, které jste alokovali v bodě 2.
5. Uvolníte ukazatel odkazující se na ukazatel odkazující se na typ `int`. Paměť jste alokovali v prvním kroku.

Způsob uložení dvourozměrného pole s různě dlouhými řádky je zobrazen na obrázku 3.3.



Obrázek 3.3 Uložení dvourozměrného pole s různě dlouhými řádky v paměti

Poznámka: Práce s dvourozměrným polem mající různě dlouhé řádky je velmi podobná práci s dvourozměrným polem majícím stejně dlouhé řádky. Rozdíl je jen v alokaci a v tom, že si musíte dávat lepší pozor, abyste nepřekročili meze jednotlivých řádků.

dynamic
CoherentArrayC2D

Dynamicky alokované souvislé dvourozměrné pole v C

Souvislé dvourozměrné pole je pole polí, tedy pole, jehož prvky jsou opět pole. V příkladu je ukázka alokace pole polí prvků typu `int` pomocí funkce `malloc`.

```
int (*array)[LENGTHY];
int i, j;

if ((array = (int (*)[LENGTHY])
malloc(LENGTHX * sizeof(int[LENGTHY]))) == NULL){
...
}
// Nějaká práce s polem
// Uvolnění pole:
free(array);
```

Identifikátor `array` je ukazatel na pole typu `int` o délce `LENGHTY`. V jazyce C se ale ukazatel vždy chápe jako ukazatel na první prvek pole daného typu. To znamená, že `array` se chápe jako ukazatel na pole složené z polí typu `int` o délce `LENGTHY`. Po alokaci pomocí funkce `malloc` můžete s identifikátorem `array` zacházet jako s identifikátorem pole, jehož počet prvků odpovídá hodnotě konstanty `LENGTHX` a každý prvek v tomto poli je pole typu `int[LENGTHY]`.

Kopírování bloků paměti

Funkce pro nízkoúrovňové kopírování částí paměti jsou v hlavičkovém souboru string.h. Pomocí funkcí `memcpy` a `memove` lze rychle kopírovat nejen obyčejné proměnné, ale i velká pole, struktury, případně pole struktur.

```
int array[] = {1, 2, 3, 4, 5, 6}, *block1, i;
if ((block1 = (int*)malloc(LENGTH * sizeof(int))) == NULL){
...
}
if (memcpy(block1, array, LENGTH * sizeof(int)) == NULL){
...
}
for(i = 0; i < LENGTH; i++){
  printf("block1[%d] = %d\n", i, block1[i]);
}
if (memmove(block1 + 1, block1, (LENGTH - 1) * sizeof(int)) == NULL){
...
}
for(i = 0; i < LENGTH; i++){
  printf("block1[%d] = %d\n", i, block1[i]);
}
free(block1);
```

Funkce `memcpy` zkopíruje blok paměti. Blok paměti začíná místem, na které se odkazuje druhý parametr. Velikost bloku paměti v bajtech udává třetí parametr. Místo, kam budete data kopírovat, je dáno prvním parametrem. Jestliže se původní blok paměti (originál) překrývá s cílovým blokem paměti (kopie), je chování funkce `memcpy` nedefinováno a bude nekorektní.

Při použití funkce `memcpy` se nesmějí originál a kopie překrývat. Funkce vrací ukazatel na začátek cílového bloku nebo `NULL` v případě chyby.

Funkce `memove` oproti `memcpy` nejprve obsah originálu zkopíruje na dočasné místo v paměti a poté obsah z dočasného místa zkopíruje na místo určené pro kopii. Celý proces je sice pomalejší, ale zato můžete funkci `memmove` použít i v situacích, kdy se originál a kopie překrývají. Význam parametrů i návratové hodnoty funkce `memmove` je stejný jako u funkce `memcpy`.

Upozornění: Při zavolání funkce memcpy nebo memmove musí první parametr odkazovat na alokovanou paměť. První parametr se musí odkazovat na alokovaný paměťový prostor minimálně o velikosti dané třetím parametrem funkce memcpy v bajtech.

memcmp

Porovnání bloků paměti

Porovnání dvou bloků paměti (porovnání jednotlivých bajtů v bloku paměti) lze využít například k porovnání dvou polí, struktur nebo polí struktur. K porovnání dvou bloků paměti slouží funkce `memcmp`.

```
unsigned char ch1[] = {'a', 'b', 'c', 'd'};
unsigned char ch2[] = {'a', 'b', 'c', 'd'};
int i1[] = {3, 4, 7, 1};
int i2[] = {3, 4, 7, 2};
int cmp;
if ((cmp = memcmp(ch1, ch2, LENGTH)) == 0){
  printf("Stejné\n");
}
else {
  if (cmp < 0){
    printf("1. je menší\n");
  }
  else{
    printf("1. je větší\n");
  }
}
if ((cmp = memcmp(i1, i2, LENGTH * sizeof(int))) == 0){
  printf("Stejné\n");
}
else {
  if (cmp < 0){
    printf("1. je menší\n");
  }
  else{
    printf("1. je větší\n");
  }
}
```

Funkce `memcmp` porovná prvních několik bajtů (počet je dán jejím třetím parametrem) dvou paměťových míst. První blok v paměti začíná místem, na které se odkazuje první parametr funkce, začátek druhého bloku paměti je dán druhým parametrem funkce. Je-li prvních N bajtů (velikost N je dána třetím parametrem) shodných, funkce vrací 0. V opačném případě vrací nenulovou hodnotu. Z návratové hodnoty můžeme i zjistit, který z paměťových bloků je „menší" a který „větší". Funkce `memcmp` chápe jednotlivé bajty jako `unsigned char` a narazí-li na první dvojici bajtů, která není shodná, potom:

- v případě, že bajt chápaný jako `unsigned char` v prvním bloku paměti je menší než bajt na stejné pozici v druhém bloku paměti, vrací funkce `memcmp` záporné číslo,
- v případě, že bajt chápaný jako `unsigned char` v prvním bloku paměti je větší než bajt na stejné pozici v druhém bloku paměti, vrací funkce `memcmp` kladné číslo.

Funkce `memcmp` je deklarována v hlavičkovém souboru `string.h`.

Inicializace (nastavení) bloku paměti


memset

Alokovaný blok paměti lze inicializovat nízkoúrovňovou funkcí `memset`. Pomocí funkce `memset` lze například inicializovat prvky pole na předem určenou hodnotu.

```
unsigned char *ch;
unsigned int *i, p;
if ((ch = (unsigned char *) malloc(LENGTH * sizeof(unsigned char))) == NULL){
...
}
if ((i = (unsigned int *)malloc(LENGTH * sizeof(int))) == NULL){
...
}
if (memset(ch, 'A', LENGTH * sizeof(unsigned char)) == NULL){
...
}
if (memset(i, 255, LENGTH * sizeof(unsigned int)) == NULL){
...
}
for(p = 0; p < LENGTH; p++){
  printf("%c\t%u\n", ch[p], i[p]);
}
free(ch);
free(i);
```

Funkce `memset` má jako svůj první parametr ukazatel na začátek alokovaného bloku paměti. Druhý parametr je hodnota, na kterou budou nastaveny všechny bajty v bloku paměti. Druhý parametr je sice typu `int`, bude ale zkonvertován na `unsigned char`. Třetím parametrem je velikost bloku paměti v bajtech.

Práce s dynamickou pamětí v C++ – operátor new

V C++ je situace poněkud komplikovanější. Funkce z jazyka C (`malloc, realloc, free`) lze použít jen za určitých okolností. Vhodnější je používat nové operátory `new` a `delete`.

Problémové je alokovat paměť funkcemi `malloc` nebo `realloc` pro instance tříd. Funkce `malloc` nebo `realloc` nezavolají konstruktor, a proto (což je asi největší problém) má-li třída, pro jejíž instance alokujete paměť, virtuální metody, nezajistí funkce `malloc` nebo `realloc` vytvoření a inicializování *tabulky virtuálních metod*.

Jazyk C++ má k dispozici operátor `new`, který krom alokace paměti také korektně vytvoří instanci, pro niž jste chtěli alokovat paměť. Operátor `new` je možné použít i při práci se základními datovými typy.

```
try{
  int *number = new int;
  *number = 5;
  std::cout << *number << std::endl;
  delete number;
}
catch (std::bad_alloc){
...
}
SimpleClass *sc = new(std::nothrow) SimpleClass;
if (sc == NULL){
...
}
delete sc;
try{
  sc = new SimpleClass(1);
}
catch (std::bad_alloc){
...
}
delete sc;
```

V případě selhání alokace paměti (například není dostatek místa v paměti) nastane jedna ze dvou možností:

1. Operátor `new` nemá parametry a v případě neúspěchu vyvolá výjimku `std::bad_alloc`.
2. Operátor `new` má parametr `nothrow` a v případě neúspěchu vrátí `NULL`.

V příkladu jsou tři situace:

1. Operátor `new` vytváří dynamicky proměnnou základního datového typu. V případě selhání alokace vyvolá výjimku typu `std::bad_alloc`.
2. Operátor `new` vytváří dynamicky instanci třídy. Instance je vytvořená bezparametrickým konstruktorem. V případě selhání alokace operátor `new` vrací `NULL`.

3. Operátor `new` vytváří dynamicky instanci třídy. Instance je vytvořená konstruktorem s parametrem. V případě selhání alokace je vyvolaná výjimka `std::bad_alloc`.

Poznámka: Operátory new a delete lze stejně jako i jiné operátory v C++ přetížit.

Dynamicky alokované jednorozměrné pole v C++

dynamicArrayCpp1d

Pro alokaci pole (souvislého alokovaného bloku paměti) v C++ slouží operátor `new`. V případě, že chcete alokovat paměť pro instanci nějaké třídy, není vhodné použít pro dynamickou alokaci paměti funkci `malloc`, která nevytvoří regulérně instanci (volání konstruktoru, vytvoření tabulky virtuálních metod). Operátor `new` dokáže vytvořit nejen jednu instanci, ale i celé pole instancí.

```
try{
  int *array = new int[LENGTH];
  for(int i = 0; i < LENGTH; i++){
    array[i] = i;
    std::cout << array[i] << std::endl;
  }
  delete[] array;
}
catch (std::bad_alloc){
...
}
SimpleClass *sc = new(std::nothrow) SimpleClass[LENGTH];
if (sc == NULL){
...
}
delete[] sc;
```

Má-li operátor new alokovat pole, je za názvem typu v hranatých závorkách velikost pole. Každý prvek pole bude inicializován bezparametrickým konstruktorem (v případě základních typů nebude inicializován). V příkladu nastávají dvě situace:

1. Operátor `new` vytváří pole základních datových typů. V případě selhání vyvolá výjimku `std::bad_alloc`.
2. Operátor `new` vytváří pole instancí tříd `SimpleClass`. Operátor `new` má jako svůj parametr `std::nothrow`, a proto v případě selhání místo vyvolání výjimky `std::bad_alloc` vrátí `NULL`.

Neexistuje možnost, jak alokovat pole instancí nějaké třídy a instance vytvořit (inicializovat) jiným než bezparametrickým konstruktorem.

Upozornění: Všimněte si, že dynamicky alokované pole se uvolňuje operátorem delete[], nikoliv delete.

Změna velikosti pole (alokovaného bloku paměti) v C++

reallocCpp

Může nastat situace, kdy v C++ budete potřebovat změnit velikost pole vytvořeného operátorem `new[]`. V C++ bohužel neexistuje obdoba funkce `realloc`, kterou znáte z jazyka C. Funkci `realloc` není vhodné používat pro úpravu velikosti bloku paměti alokovaného pomocí operátoru `new`. Funkce `realloc` (stejně jako `malloc`) nevytvoří korektně instance (nezavolá konstruktor, nevytvoří a neinicializuje tabulku virtuálních metod).

Následující příklad obsahuje šablonu funkce, která zajistí obdobnou funkčnost jako funkce `realloc` (změní velikost pole).

```
#include <algorithm>

template<class Type>
Type *reallocCpp(Type *pointer, size_t originalSize,
size_t newSize){
  Type *ret = new Type[newSize];
  size_t n = std::min(originalSize, newSize);
  std::copy(pointer, &pointer[n], ret);
  delete[] pointer;
  return ret;
}
```

Parametr šablony je typ pole, jehož velikost chcete měnit. Instancí šablony je funkce. Prvním parametrem funkce je ukazatel na začátek dynamicky alokovaného pole pomocí operátoru `new`. Datový typ prvků v poli určuje parametr šablony. Druhým parametrem je původní počet prvků a třetím parametrem je nový počet prvků.

V samotné funkci dojde k alokaci nového pole, překopírování prvků (pomocí standardního algoritmu `std::copy`) do nového pole a uvolnění starého pole. Počet prvků, které budete kopírovat, je menší číslo z originální a nové velikosti.

Dynamická alokace dvourozměrného pole v C++

dynamicArrayCpp2d

Dvourozměrné pole v C++ je stejně jako v C pole polí. Na rozdíl od jazyka C nebudete ale alokovat pole pomocí funkce `malloc`, ale operátorem `new`.

```
int **array;
int i, j;
try {
  array = new int* [LENGTHX];
}
catch(std::bad_alloc e){
  std::cerr << "Problém s alokací paměti" << std::endl;
  return -1;
}
for (i = 0; i < LENGTHX; i++){
```

```
  try{
    array[i] = new int[LENGTHY];
  }
  catch (std::bad_alloc e){
    std::cerr << "Problém s alokací paměti" << std::endl;
    while(--i > 0){
      delete[] array[i];
    }
    delete[] array;
    return -1;
  }
}
// Nějaká práce s polem
// Uvolnění pole:
for(i = 0; i < LENGTHX; i++){
  delete[] array[i];
}
delete[] array;
```

Identifikátor array je ukazatel na ukazatel odkazující se na typ int. Práce s dynamickým dvourozměrným polem by se dala popsat v pěti bodech:

1. Nejprve musíte alokovat jeden rozměr dvourozměrného pole. Bude se jednat o pole ukazatelů na typ `int`. Alokace pole ukazatelů v C++ vypadá takto: `new int* [LENGTHX]`.
2. Dále musíte alokovat pro každý prvek pole ukazatelů paměťový prostor reprezentující druhý rozměr dvourozměrného pole.
3. Nyní můžete s dvourozměrným polem pracovat. Pro indexaci pole můžete využívat operátory `[]`. Například: `array[i][j] = (i + 1) * (j + 1)`.
4. Postupně v cyklu uvolníte všechny alokované bloky paměti, které jste alokovali v bodě 2. Pole v C++ musíte uvolnit operátorem `delete[]`, nikoliv `delete`.
5. Uvolníte ukazatel odkazující se na ukazatel odkazující se na typ `int`. Paměť jste alokovali v prvním kroku. Pole v C++ musíte uvolnit operátorem `delete[]`, nikoliv `delete`.

Způsob uložení dynamického dvourozměrného pole (pole polí) v paměti je shodný jako způsob uložení dynamicky alokovaného dvourozměrného pole v C (alokovaného funkcí `malloc`) a je zobrazen na obrázku 3.2.

Upozornění: Dynamické dvourozměrné pole není souvislý blok paměti tak jako statické dvourozměrné pole.

Alokace a naplnění dvourozměrného pole s různě dlouhými řádky v C++

triangleArrayCpp

Dvourozměrné pole s různě dlouhými řádky je vlastně jednorozměrné pole ukazatelů ukazujících na jednotlivá jednorozměrná pole prvků. Nic vám nebrání v tom, aby jednotlivá pole byla různě dlouhá. Ukazatel na ukazatel lze použít k práci s dvourozměrným polem, ve kterém

každý řádek bude mít různě dlouhou délku. Oproti klasické alokaci dvourozměrného pole budete pro každý řádek alokovat blok paměti jiné velikosti.

```
int **array;
int i, j;
try {
  array = new int* [LENGTH];
}
catch(std::bad_alloc e){
  std::cerr << "Problém s alokací paměti" << std::endl;
  return -1;
}
for (i = 0; i < LENGTH; i++){
  try{
    array[i] = new int[LENGTH];
  }
  catch (std::bad_alloc e){
    std::cerr << "Problém s alokací paměti" << std::endl;
    while(--i > 0){
      delete[] array[i];
    }
    delete[] array;
    return -1;
  }
}
// Nějaká práce s polem
// Uvolnění pole:
for(i = 0; i < LENGTH; i++){
  delete[] array[i];
}
delete[] array;
```

Identifikátor `array` je ukazatel na ukazatel odkazující se na číslo typu `int`. Práce s dynamickým dvourozměrným polem, které má různě dlouhé řádky (nebo sloupce, záleží jak si pole otočíte ve svých představách), by se dala popsat v pěti bodech:

1. Nejprve musíte alokovat jeden rozměr dvourozměrného pole. Bude se jednat o pole ukazatelů na typ int. Alokace pole ukazatelů v C++ vypadá takto: `new int* [LENGTHX]`.
2. Dále musíte alokovat pro každý prvek pole ukazatelů paměťový prostor reprezentující druhý rozměr dvourozměrného pole. V příkladu je dvourozměrné pole trojúhelníkového tvaru, proto délka řádků závisí na pořadí řádku.
3. Nyní můžete s dvourozměrným polem pracovat. Můžete využívat operátory `[]` pro indexaci pole. Například: `array[i][j] = (i + 1) * (j + 1)`
4. Postupně v cyklu uvolníte všechny alokované bloky paměti, které jste alokovali v bodě 2.
5. Uvolníte ukazatel odkazující na ukazatel odkazující se na typ `int`. Paměť jste alokovali v prvním kroku.

Způsob uložení dvourozměrného pole s různě dlouhými řádky v C++ je shodný se způsobem uložení dvourozměrného pole s různě dlouhými řádky v C (alokovaného pomocí funkce `malloc`) a je zobrazen na obrázku 3.3.

Poznámka: Práce s dvourozměrným polem majícím různě dlouhé řádky je velmi podobná práci s dvourozměrným polem majícím stejně dlouhé řádky. Rozdíl je jen v alokaci a v tom, že musíte pečlivěji hlídat meze polí, abyste je nepřekročili.

Dynamicky alokované souvislé dvourozměrné pole v C++

dynamicCoherent ArrayCpp2D

Souvislé dvourozměrné pole je pole polí, tedy pole, jehož prvky jsou opět pole. V příkladu je ukázka alokace pole polí prvků typu `int` pomocí operátoru `new`. Obdobným způsobem lze v C++ alokovat dvourozměrné pole prvků jakéhokoliv typu.

```
int (*array)[LENGTHY];
int i, j;
try {
  array = new int [LENGTHX][LENGTHY];
}
catch(std::bad_alloc e){
...
}
// Nějaká práce s polem
// Uvolnění pole:
delete[] array;
```

Identifikátor `array` je ukazatel na pole, jehož prvky jsou jednorozměrná pole obsahující jako prvky čísla typu `int`. Po alokaci pomocí operátoru `new` můžete s identifikátorem `array` zacházet jako s identifikátorem pole, jehož počet prvků odpovídá hodnotě konstanty `LENGTHX` a každý prvek v tomto poli je pole typu `int[LENGTHY]`.