A programozás alapjai 2.

Hivatalos segédlet a második laborhoz.

Tartalomjegyzék

C vs C++ 2
Szintaktika 2
Egyéb különbségek 3
Konstans típus 4
Konstans változót mindig *inicializálni kell*. 4
Konstans típusú változó *értéke közvetlenül nem változtatható*. 4
Konstansra nem mutathat olyan pointer, *amin keresztül megváltoztathatnánk a változót*. 4
Tömb is lehet konstans, de akkor nem változtatható! 4
Konstans int típussal lehet tömböt deklarálni, mivel fordítási időben kiértékelődik. 4
Konstans pointer 5
Konstans referencia = referencia egy konstansra 5
Konstans referencia használata paraméterátadásnál 5
Memóriatípusok 6
A statikus/globális memóriaterület 6
A verem/stack 6
A dinamikus memóriaterület: heap 6
Tömb átadása paraméterként 7
Miért nem szeretjük a rekurzív függvényeket? 7
Buffer overflow attack 8
Hogyan kell használni a scanf_s()-t? 9
Miért úgy használjuk a scanf() függvényt, ahogy? 10
Honnan tudja a scanf(), hogy honnan kell olvasnia? 10
Mi az a függvénypointer? 11
És ez miért jó? 11
Függvény túlterhelés 11
Parancssori paraméterek 12
Inline függvények 12
Alapértelmezett függvényparaméterek 12
Hogyan működik a tömbindexelés? 13
Miért nincs C/C++ nyelvben ellenőrzés tömbindexekre? 13
Hogyan lenne célszerűbb megoldani a problémát? ..... **Hiba! A könyvjelző nem létezik.**
Ajánlott irodalom 14

C vs C++

Szintaktika

C	C++

`int *x = malloc( sizeof(int) );`
`int *x_array = malloc( sizeof(int) * 10 );`
`// ...`
`free( x );`
`free( x_array );`
	`int *x = new int;`
	`int *x_array = new int[10];`
	`// ...`
	`delete x;`
	`delete[] x_array;`

`#include <stdio.h>`
`int main() {`
`  foo();`
`  return 0;`
`}`
 	
`int foo() {`
`  printf("Hello world");`
`}`
	`#include <stdio.h>`
	`int foo() {`
	`  printf("Hello world");`
	`}`
	
	`int main() {`
	`// kötelező deklarálni használat előtt`
	`  foo();`
	`  return 0;`
	`}`

`struct MyStruct`
`{`
`  int x;`
`};`
 	
`struct MyStruct MyStructInstance;`
 	
`// ...`
`typedef struct DummyStructName`
`{`
`  int y;`
`} MyAmazingStruct;`
 	
`MyAmazingStruct MyAmazingStructName;`
	`struct MyStruct {`
	`  int x;`
	`};`
	
	`MyStruct MyStructInstance;`
	
	`// ...`
	
	`typedef struct DummyStructName {`
	`  struct MyStruct MyStructBestInstance;`
	`  // struct_name_t instance2; // invalid!`
	`} MyAmazingStruct;`

`gcc foo.c -lm` `// -lm: Math library`	`g++ foo.cc` `// C++-ban már nem kell külön linkelni egyes libraryket.`

`typedef enum {FALSE, TRUE} bool;`	`// van beépítve bool típus`
	`bool b; // deklaráció`
	`bool b1 = true; // copy inicializáció`
	`bool b2(false); // direkt inicializáció`
	`bool b3 { true }; // uniform inicializáció (C++11)`
	`b1 = false; // értékadás`

	`bool b4 = !true; // false`
	`bool b5(!false); // true`

`int main() {`
`  printf("Hello, World");`
`  return 0;`
`}`
	`int main() {`
	`  printf("Hello, World");`
	`}`

`int i;`
`for(i=1;i<20;i++)`
`printf("i=%d\n",i);`
	`for(int i=1;i<20;i++)`
	`std::cout << "i=" << i << std::endl;`

Egyéb különbségek

	C	C++
Tervezte	Dennis Ritchie	Bjarne Stroustrup
Először megjelent	1972	1985
Platform	bármi, amire implementálták a compilert	bármi, amire implementálták a compilert
Szöveg típus (bővebben)	nincs, de reprezentálható char tömbként	van, std::string
OOP (Objektumorientált Programozás) támogatás (bővebben)	nincs, de imitálható struktúrákkal	van
Egységbezárás (Encapsulation) (bővebben)	nincs	van
Osztályok (class) (bővebben)	nincs	van
Polimorfizmus (bővebben)	nincs	van
Input/Output	input: scanf output: printf	input: std::cin output: std::cout, std::cerr, std::clog (bővebben)
Genericitás (bővebben)	nem	igen
Procedurális (bővebben)	igen	igen
Reflexió (bővebben)	nincs	van
Inline kommentezés	//	//
Blokk komentek	/* TODO: ... */	/* TODO: ... */

Operátor túlterhelés (bővebben)	nincs	van
Utasítás végjelzés	;	;
Névtér (bővebben)	nincs	van
Hibakezelés (bővebben)	nincs	van

(Az ismeretlen fogalmakkal a félév további részeiben fogtok megismerkedni, itt most csak ismeretterjesztésként szolgál).

Konstans típus

Konstans változót mindig *inicializálni kell*.

```
const int i;  // nem OK
const int i = 20; // OK
```

Konstans típusú változó *értéke közvetlenül nem változtatható*.

A következő példa eszerint **hibás**:

```
const int i = 20; // OK
i++; // nem OK
```

Konstansra nem mutathat olyan pointer, *amin keresztül megváltoztathatnánk a változót*.

A pointernél jelölni kell majd, hogy konstans értékre mutat. Egy példa a **helytelen** használatra:

```
const int i = 20;
int* p = &i;
(*p)++;
```

Tömb is lehet konstans, de akkor nem változtatható!

```
const int array[] = {10, 20, 30};
array[2]++;  // nem OK!
```

Konstans int típussal lehet tömböt deklarálni, mivel fordítási időben kiértékelődik.

Ezért ez a kódrészlet **jó**:

```
const int i = 20;
char array[i];
```

Konstans pointer

int* **const** egy konstans pointer egy nem konstans int-re → a mutatott változó módosítható, a mutató nem

int const* vagy **const int*** egy nem konstans pointer egy konstans int-re → a mutatott változó nem módosítható, de a mutató igen.

const int* **const** egy konstans pointer egy konstans intre

```
char s[]="Szia";
const char *pc=s;          // const char-ra mutató pointer
pc[0]='s';          // nem OK, konstansra mutat
pc++;               // OK, a pointer nem konstans, így változtatható

char s[]="Szia";
char* const cp=s;          // char-ra mutató konstanspointer
cp[0]='s';          // OK, nem konstansra mutat
cp++;               // nem OK, a pointer konstans

char s[]="Szia";
const char* const cpc=s;// const char-ra mutató konstanspointer
cpc[0]='s';         // nem OK, konstansra mutat
cpc++;              // nem OK, a pointer konstans…
```

Konstans referencia = referencia egy konstansra

Ez amiatt van, mert a referenciát alapból nem lehet megváltoztatni. Referencia esetén a const kulcsszó a hivatkozott értékre vonatkozik

```
int j = 0;
int const &i = j;
i = 1;              // nem OK, a hivatkozott érték konstans
const int &k = j;
k = 1;              // nem OK, a hivatkozott érték konstans
```

Konstans referencia használata paraméterátadásnál

Ha **nagy objektum**ot szeretnénk egy függvénynek paraméterül átadni 3 lehetőségünk van:

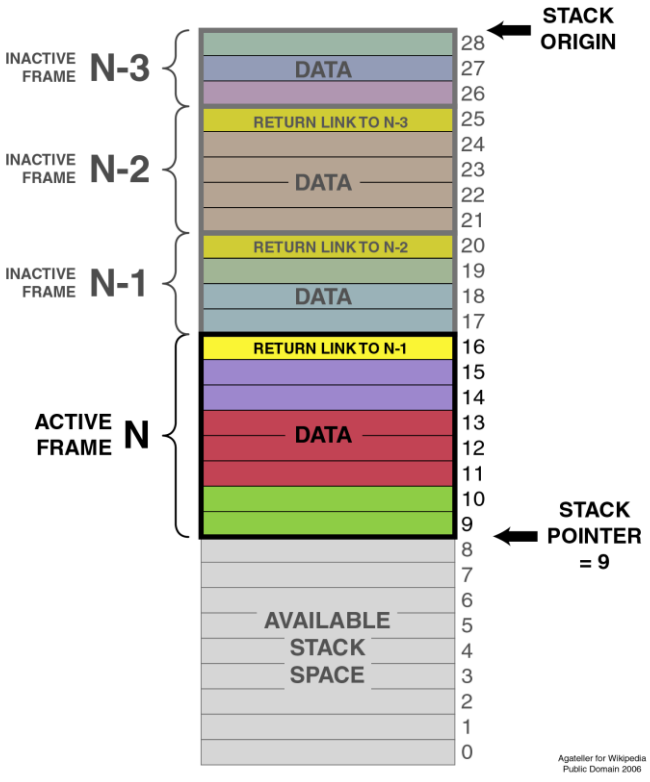
- érték szerinti átadás - lassú
- referencia szerinti átadás - gyors, de a függvény megváltoztathajta a hivatkozott objektumot
- konstans referencia szerinti átadás - gyors, a függvény nem módosíthatja az objektumot, ezt preferáljuk

Memóriatípusok

A statikus/globális memóriaterület

A statikus memóriaterületen helyezkednek el a globális változók, ezek a program egész futása során léteznek. Még a main() függvény végrehajtásának kezdete előtt létrejönnek.

```
int num; // Létrehoz egy integert a globális memóriaterületen
char str[10] = "door";   // Létrehoz egy 10 elemű karaktertömböt
char *p = "chair";
/* Létrehoz egy névtelen karaktertömböt a globális memóriaterületen ("chair") és deklarál egy char* típusú
pointert, ami erre a névtelen karaktertömbre mutat. A pointer csak kezdeti értékként mutat erre tömbre,
átállíthatjuk máshova, de onnantól  "chair" karaktertömböt nem érjük el*/
int main() {
   num = 10;
   printf("%d", num);
   /*a formátum string is a globális memóriaterületen jön létre,  a printf() pedig egy pointert kap erre*/
   return 0;
}
```



A verem/stack

A függvények **lokális változó**i kerülnek ide. Tartalma változik, függvényhíváskor a stack tetején létrejönnek annak lokális változói, amikor pedig **visszatérünk a függvényből, a lokális változói megszűnnek**. Minden függvény csak a **saját lokális változóit látja**, a többi függvényét nem. **Rekurzió esetén** (amikor a függvény saját magát hívja meg), egymástól **független lokális változók** keletkeznek a különböző függvénypéldányokhoz, egymását nem tudják elérni. Egy függvény hívásakor annak paraméterei is lemásolódnak a verembe.

A dinamikus memóriaterület: heap

Olyan terület, melyből a program **futása közben egy adott nagyságú részt kérhetünk**, és ha már nincs rá szükségünk, **visszaadhatjuk azt**. Ez teszi lehetővé, hogy akkora méretű memóriát foglaljunk le, melynek **nagyságát** a program **fordításakor még nem ismerjük**. Lefoglaláskor egy pointert kapunk a kért méretű memóriaterületre, ha talált ilyet. Ne felejtsük el, hogy nem biztos, hogy az operációs rendszer tud nekünk memóriát foglalni, ekkor a malloc() NULL-lal tér vissza, ezért **mindig ellenőrizni kell** a visszaadott pointer értékét.

Amikor **felszabadítunk** egy memóriaterületet, a pointerünk továbbra is arra a területre fog mutatni (amíg el nem állítjuk onnan), de már **tilos arra hivatkozni**. Még akkor sem, ha azóta nem foglaltunk új memóriát, mivel akár egy másik program is használhatja azt.

Mi történik ha elfelejtünk felszabadítani egy dinamikusan lefoglalt memóriaterületet? A program lefordul, ha nincsen más hiba, jó eredménnyel lefuthat. Viszont a felszabadítatlan memóriaterület a program futásának befejeződése után is **felszabadítatlan marad**. Ennek eredményeként lecsökken a felhasználható memória mérete, többszöri futtatás után egyre kevesebb és kevesebb szabad memóriája marad a gépnek.

Ezt általában csak akkor vesszük észre, ha lassulni kezd a gép. Ezért ez a hiba nehezen felderíthető, úgyhogy jegyezzük meg, hogy a programban pont annyi free() függvényhívásnak kell szerepelni, ahány malloc()-nak. ***(“Aki malloc()-ot mond, mondjon free()-t is” /Dr. Czirkos Zoltán/)***.

Vigyázat! Ez alatt nem azt értjük, hogy megszámoljuk, hányszor lett leírva a programban a malloc() és free() utasítás, hanem hogy a futás során hányszor hívódik meg. Pl. ha egy ciklus törzsében foglalunk memóriát malloc()-kal 50-szer, akkor ehhez összesen 50 darab free() hívás fog tartozni, valószínűleg szintén ciklusban. Elágazásnál szintén figyelni kell, hogy a különböző ágakban hány malloc() hívás történik, és a későbbiekben ugyanennyi free() hívás történjen.

Tömb átadása paraméterként

Tömböket paraméterként függvényeknek átadni a **kezdőcímükkel** lehet. Mivel a függvény a tömbből csak a kezdőcímét látja, nem tudja annak méretét. Próbáljuk meg a sizeof() függvénnyel megtudni a függvényben? Nem fog sikerülni, mivel csak egy pointert kap a függvény, így a sizeof()-fal ennek a pointernek a méretét kapjuk meg, nem a tömbbét. A megoldás, hogy **át kell adnunk a tömb méretét is** paraméterként.

Miért nem szeretjük a rekurzív függvényeket?

A függvényeket meghívásukkor úgynevezett stack frame-ekbe (verem keret) helyezzük. Az adott függvény befejezésekor a stack frame megsemmisül. Ez három részből áll:

- a függvény argumentumai
- minden nem-statikus lokális változó a függvényben
- a visszatérési cím, ahova vissza kell ugorni, miután lefutott a függvény

Ez az elv lehetővé teszi a rekurziót. Például, ha van egy ilyenünk:

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return factorial(n - 1) * n;
}
```

Aztán meghívjuk így: factorial(3); ilyen szerkezetet kapunk:

---------- ESP

n = 1

RA = <in factorial()>

---

n = 2

RA = <in factorial()>

---

n = 3

RA = <in main()>

A függvény definíciójából látszódik, hogy 3 stack frame-nek kell létrejönnie. Ez ugyan ebben az esetben nem visz el számottevő memóriát, de ha nagyobb számokat adunk meg paraméternek, már jól kivehető, hogy egyáltalán nem helytakarékos, ráadásul még lassú is. Azonban tudjuk, hogy minden rekurzív függvény megírható procedurálisan, ciklusokkal. Lehetőség szerint inkább írjuk meg így a programunkat.

Buffer overflow attack

Ha scanf() függvényt akartok használni a programotokban, a következő warningot fogjátok kapni Visual Studioban:

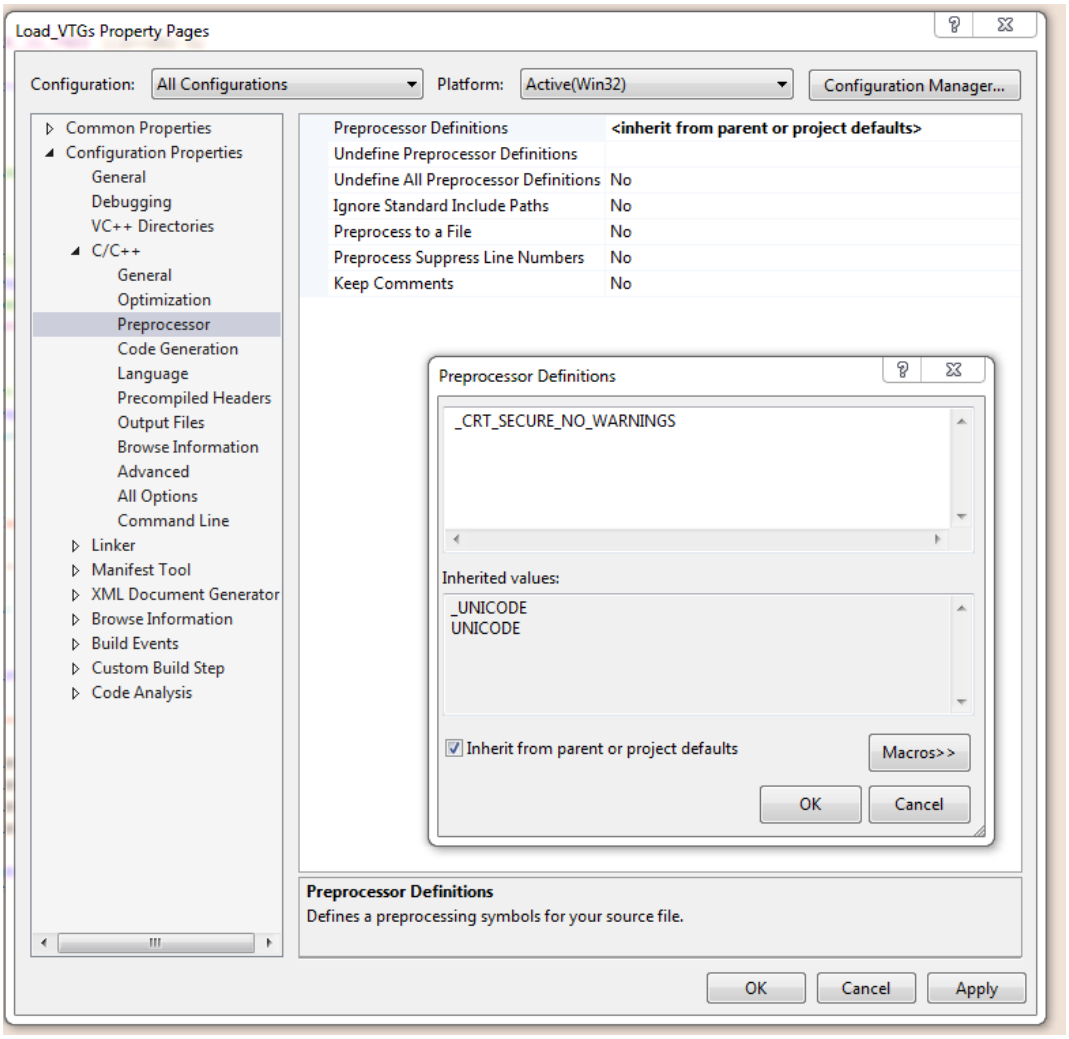
Error C4996 'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.

Ezzel az üzenettel jelzi nekünk a fordító, hogy a scanf() **nem biztonságos**. Na de mit is jelent ez? A scanf()-ben található egy olyan bug amit ha kihasználnak (exploit), akkor a programunk azonnyomban leáll. Ez az úgynevezett **buffer overflow attack.** A probléma abból fakad, hogy a scanf()-nek senki nem mondja meg, hogy hány bájtot olvasson. Ebből fakadóan addig olvas, amíg azt “hiszi”, hogy olvasnia kell. Egy karakter tömb esetében ez egy null terminátor ‘\0’ vagy ‘\n’.

Azonban ha valaki több karaktert akar eltárolni, mint amennyit lefoglalt, a függvény olyan helyre fog írni, ami nem az övé. Ekkor a program **segmentation fault** hibával leáll.

Viszont van egy modernebb - C++11 szabványban létező - függvényünk, a **scanf_s** (_s = secure), ami lehetővé teszi, hogy megadjuk hogy hány bájtot akarunk olvasni. Ez a funkció garantálja, hogy több bájtot olvassunk, mint amennyire szükségünk van.

Ha viszont valamiért mégis használni szeretnénk a scanf()-et, a következő beállítást kell végrehajtanunk a compileren:



Vagy másik megoldásként kikapcsolhatjuk a warning-ot: #pragma warning(disable:4996) (és ez egyéb, nem biztonságosnak minősített műveletek esetén is megtehető, különböző hibakódokkal)

Hogyan kell használni a scanf_s()-t?

```
printf("Enter your name: ");
char name[5]; // OK: "Alex\0", NOT OK: "Alexander\0"
if (scanf_s("%s", name, sizeof(name)) == 1)
  printf("Your name: %s\n", name);
else { // kezeljük a keletkezett szituációt (kivételkezelés alapja...)
  printf("Invalid length of data!\n");
  strcpy(name, "????");
  int c;
  while ((c = getchar()) != '\n' && c != EOF); // stdin buffer ürítése (flush)
}
```

Miért úgy használjuk a scanf() függvényt, ahogy?

Honnan tudja a scanf(), hogy honnan kell olvasnia?

Ha megnézzük a scanf() definícióját, látni fogjuk, hogy teljesen megegyezik a fscanf() függvénnyel. Az fscanf()-ről azt kell tudni, hogy bármilyen streamről (folyam) tud olvasni. Minden operációs rendszer alapértelmezetten hármat szolgáltat:

- **stdin:** standard input; Pl. a leütött billentyűk betűi ennek a bufferén jelennek meg. Innen lehet őket kiolvasni.
- **stdout:** standard output; ha a konzolra írunk ki, akkor valójában erre a streamre írunk (#protipp: printf() = fprintf())
- **stderr:** standard error; kimenete a stdout-ra irányul át alapértelmezetten, de ha szeretnénk, akár fájl streambe is átirányíthatjuk.

Fájl stream? Igen, ilyen is van, ugyanis az operációs rendszer ugyanolyan adatfolyamként kezeli mint a stdin, stdout, stderr mester hármast. Ez azért jó, mert egyszerűen tudunk fájlt kezelni (az igazi erejével Java-ban találkozhatunk, mert ott akár át is irányíthatunk egymásba streameket: ami az egyiknek kimenet, az a másiknak a bemenete lesz és a kettő közé beékelhetünk valamilyen “filtert” ami mondjuk vagy átenged adatot, vagy nem valamilyen szempont szerint, vagy akár módosíthat rajtuk. Alkalmazása pl.: real-time titkosítás).

Szóval a lényeg:

- `scanf("%d", &valtozo) = fprintf(stdin, "%d", &valtozo)`
- `printf("Num: %d", n) = fprintf(stdout, "Num: %d", n)`

Mi az a függvénypointer?

Pointerekkel nem csak változókra illetve tömbökre tudunk mutatni, hanem akár függvényekre is. Gondoljunk bele: a függvényeket is a memóriában tároljuk, tehát kell lennie címüknek is. Na de milyen típussal hivatkozzunk rájuk? Esetleg void*? Nos, saját szintaktikájuk van. Tekintsük az alábbi C kódot:

```
#include <stdio.h>

int add(int a, int b) {
	return a + b;
}

main() {
	int (*plus1)(int, int) = add;  // jó
	int (*plus2)(int, int) = &add;  // jó
	printf("%d, %d", plus1(1, 2), plus2(1, 2)); // 3, 3 a kimenet
	getchar();
}
```

Azért lehetséges függvénypointert létrehozni, mert a függvények is a memóriában tárolódnak, így van memóriacímük is, amire tudunk hivatkozni.

Szintaktika: **visszatérési_típus(*fv_pointer)(paraméterek típusa);**

És ez miért jó?

TFH írunk egy rendező függvényt, ami alapértelmezetten tud növekvő sorrendbe rendezni, de azt szeretnénk hogy ha úgy kívánja a helyzet, akkor tudjon csökkenőbe is, szóval a függvényünknek adunk egy bool (0/1) paramétert hogy ha az igaz (1), akkor növekvőbe, különben meg csökkenőbe rendezzen.

Aztán ahogy telik az idő, igényt formálunk arra hogy ne csak ebben a kétféle módban rendezzen, hanem bonyolultabb algoritmus szerint. Szóval azt csináljuk, hogy 0/1 érték helyett egy saját kiértékelő függvényt (**predikátumot**) adunk át paraméternek.

Függvénynév túlterhelés

Lehetővé teszi, hogy ugyanahhoz a függvénynévhez több definíciót hozzunk létre. Az ugyanolyan nevű függvényeknek deklarációban különbözniük kell, tehát a paramétereik száma vagy azok típusa más. Viszont nem számít túlterhelésnek, és a fordító el sem fogadja, ha két függvény csak és kizárólag a visszatérési értékük típusában tér el egymástól.

Mire jó ez? Tegyük fel, hogy írtunk egy olyan függvényt, mely összead két double típusú számot. Legyen ez a double add(double a, double b) nevű és paraméterezésű függvényünk. Aztán később szükségünk lenne egy olyan függvényre, ami int típusú számokat ad össze. Adhatnánk a függvényünknek más nevet is, de az “add” elnevezés egyszerű és elegendően kifejező, ezért egyszerűen írhatunk egy int add(int a, int b) függvényt.

Ebben az esetben, amikor meghívjuk a programunkban az “add” függvényt, a fordító kikeresi, hogy melyik függvényt is kell most meghívni a megadott paraméterektől függően. Ha két double értékkel hívjuk meg, akkor

az első függvény hívódik meg, ha két int értékkel hívjuk meg, akkor pedig a második. Tegyük fel, hogy több "add" nevű függvényt nem írtunk. Ekkor, ha egy double és egy int paraméterrel hívjuk meg, hibát kapunk, mivel nem létezik "add" függvény ilyen típusú paraméterekkel.

Parancssori paraméterek

C++-ban lehetőség van a main függvényben parancssori argumentumok fogadására a következő paraméterezéssel:

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

- **argc**: A kapott paraméterek száma.
- **argv[]**: A kapott paraméterek karaktertömbként

A legelső paraméter mindig a futtatható fájl elérési útvonala és neve.

A main() függvényt megírhatjuk paraméterek átvétele nélkül is, amit a függvénytúlterhelés (ld. fentebb) tesz lehetővé.

Inline függvények

Az inline függvények abban különböznek a hagyományos függvényektől, hogy a fordító minden hívás helyére bemásolja a függvény törzsét, ha a függvény kicsi. Az "inline" megjelölés valójában egy ajánlás a fordítónak, hogy tegye meg ezeket a lépéseket, azonban dönthet úgy, hogy ezt figyelmen kívül hagyja. Hogy megértsük, mikor hogy dönt, nézzük meg a hátterét:

Az egész elképzelés mögött az áll, hogy ha van egy rövid függvény (pl. Két szám összeadása vagy kivonása), és azt nagyon sok helyen hívjuk, akkor ez sok felesleges idővel jár. Minden függvényhíváskor például kezelni kell a stack-et. Ekkor sok időt spórolunk, ha függvényhívás helyett a kódba másoljuk a függvény törzsét. Pont ez történik inline függvények esetén.

A sok bemásolgatással viszont nő a kód mérete. Amikor a compiler úgy véli, hogy túl hosszú az inline-nak megjelölt függvény törzse és így nagyon megnőne a kód mérete, egyszerűen nem végzi el a másolásokat, hanem marad hagyományos függvény.

Alapértelmezett függvényparaméterek

- Egyszer lehet definiálni. Még akkor sem lehet többször (pl. külön a header és *.c vagy *.cpp fájlban), ha ugyanazt az értéket adnánk meg.
  - `int magicFunction(int a, int b); // header-ben`
  - `int magicFunction(int a, int b = 0) {...} // *.c vagy *.cpp fájlban`
- Lehetővé teszik, hogy egy függvényt minden paraméterének megadása nélkül meghívjuk. Ilyen esetben a nem megadott paraméterek értékét helyettesítik a definiált értékkel.
- A paraméterlista vége felől folytonosnak kell lenniük.

```
void add(int a, int b = 3, int c, int d = 4); // nem OK
void add(int a, int b = 3, int c, int d); // nem OK
void add(int a, int b = 3, int c = 2, int d = 4); // OK
```

Hogyan működik a tömbindexelés?

```
int tomb[10];
int *p = tomb; // a p pointert a tömb elejére állítjuk (0. elemre)
int  i = 2;
*(tomb+i) = 3;
tomb[i] = 3;
*(p+i) = 3; // p+i az i. elemre mutató pointer, *(p+i) pedig maga az i. elem
p[i] = 3;
```

Tömbök elemének elérésekor két művelet hajtódik végre (mindig): az elem címének kiszámítása, majd az elem elérése (indirekcióval). Ezt a kétlépcsős elérést jobban szemlélteti a **(tomb+i) = 3;* és **(p+i) = 3;* sor a fenti kódban. A tömb nevére hivatkozva a fordító megadja, hogy hol található a tömb, egy pointert kapunk rá, melyen az indexelő [] operátort alkalmazva címszámítást és dereferálást (*) is végzünk. A **(tomb+i)* ugyanazt jelenti, mint a *t[i].* Az indexelő operátor [] csak egy rövidítés, az emberi gondolkodást könnyíti: könnyebben olvasható forma, ezért általában ezt használjuk.

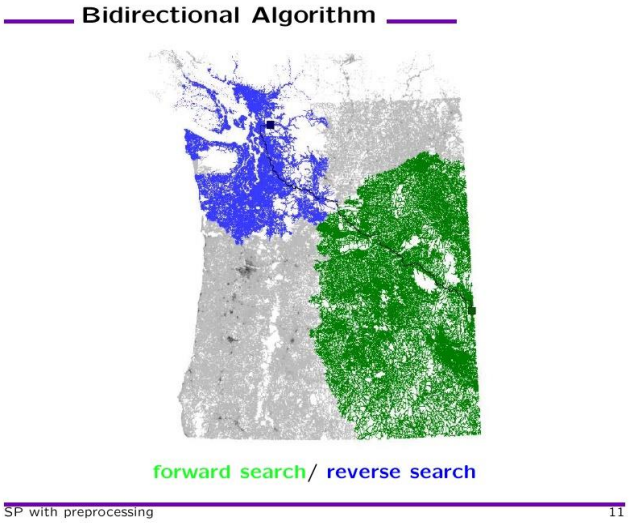
Miért nincs C/C++ nyelvben ellenőrzés tömbindexekre?

- http://stackoverflow.com/questions/7410296/why-is-bounds-checking-not-implemented-in-some-of-the-languages (ajánlott irodalomban is megtalálod)

Vermes labor feladat problémái

Tegyük fel, hogy írtunk N db C függvényt, ami mind-mind ahhoz kell, hogy stack (*verem, LIFO*) struktúrákat tudjunk kezelni (push, pop, …). Írni szeretnénk M db queue-t (*sor, FIFO*) kezelő függvényt is (push, pop).
Első probléma: milyen névvel illessük az új függvényeket? Push, pop már van, szóval prefixet illesztünk a nevük elé, hogy ne legyen névütközés: stack_push, stack_pop, queue_push, queue_pop. Az analógiát követve nyilvánvalóan belátható, hogy egy komolyabb méreteket öltő szoftver alkalmazásnál *ez a módszer nem alkalmazható.*

Második probléma: összefüggő, komplex adatstruktúrát nem tudunk modellezni. Például: ismert probléma, hogy Dijkstra útkereső algoritmusának időbeli komplexitása már elég alacsony ahhoz, hogy implementálásával egy "jó" rendszert lehessen építeni, de nem elég jó például egy Google Mapshez ahol rengeteg csúcs és él van, így még mindig lassúnak számít. Ezért mérnökök (a matematikusok általában az előző lépésnél már megálltak), el kezdtek kísérletezni. Az egyik próbálkozás az volt, hogy ha ismert a start és a cél, akkor indítsák mindkét irányból a Dijkstrat és ahol legelőször találkoznak, összekötik a két szakaszt és megkapják a legkisebb költségű utat. Remélem ezután mindenki kapisgálja, hogy itt bizony *két azonos adatstruktúra* van a háttérben, azonban *különböző értékekkel*. Ilyenkor természetes késztetést érzünk arra, hogy az ugyanazon célt szolgáló változókat, tulajdonságokat, metódusokat *egységbezárjuk (encapsulation)*. Ezt az új adatmodellt (adatok és kapcsolataik + a rajtuk végezhető műveletek definiálását) osztályoknak hívják. Ha konkretizálni szeretnénk az osztályt és az egyes változóknak értéket szeretnénk adni, *példányosítjuk* egy (vagy több) *objektumba.*



Harmadik probléma: nem tudunk adatot elrejteni. Tegyük fel, hogy többen dolgoznak egy szoftveren (értsd: mindig), nem teljesen egyértelmű, hogy melyik adatot kell és hogyan módosítani. Például ha lemodelleztünk egy személyt (*Person* osztály), akkor nem szeretnénk ha valaki más módosítaná a *quentin* nevű Person típusú objektumunk email címét *asdqwerew*-re, hiszen ez nem egy értelmes adat. Előtte validálnunk kell, hogy tényleg megfelel-e az elvárt email formátumnak. A megoldás, hogy a *Person* osztályon kívüli világ elől elrejtünk minden változót és tagfüggvényeken keresztül módosítjuk (*setEmail*).

Ajánlott irodalom

- https://en.wikipedia.org/wiki/Standard_streams
- http://stackoverflow.com/questions/23378636/string-input-using-c-scanf-s
- http://stackoverflow.com/questions/21434735/difference-between-scanf-and-scanf-s
- http://www.eet.bme.hu/~czirkos/cpp.php
- http://www.eet.bme.hu/~czirkos/cpp_iras/memoria.pdf
- http://stackoverflow.com/questions/840501/how-do-function-pointers-in-c-work
- http://stackoverflow.com/questions/7410296/why-is-bounds-checking-not-implemented-in-some-of-the-languages
- https://en.wikipedia.org/wiki/Function_pointer
- http://c.learncodethehardway.org/book/ex18.htm
- http://stackoverflow.com/questions/654754/what-really-happens-when-you-dont-free-after-malloc
- http://stackoverflow.com/questions/31372892/running-malloc-in-an-infinite-loop
- https://www.programiz.com/cpp-programming/default-argument