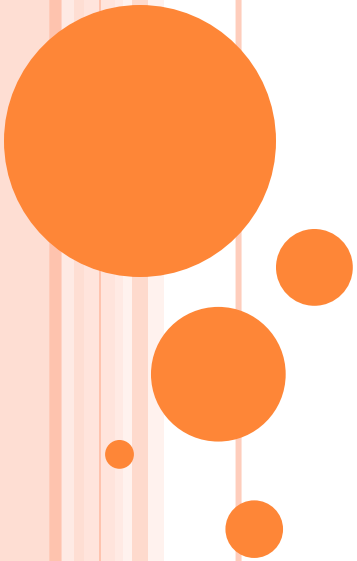


Programovacie jazyky pre vstavané systémy

Typové kvalifikátory, variadické funkcie, preprocesor

Otázky z minulej prednášky

- Môže v jazyku C vrátiť funkcia pole?
- Čo je to callback a ako ho môžeme implementovať v jazyku C?
- Na čo slúžia sockety?
- Sú sockety súčasťou štandardu jazyka C?
- Prečítajte nasledujúce zápisy:
  - int (*compar)(const void *, const void*);
  - void (*signal(int sig, void (*func)(int)))(int);
  - int* (*a)[10];
  - double* (*(*f)())[];
  - typedef int (*(*(*FUN_ARR)())[])(int, int);
- Napíšte definíciu typu P_FUN, ktorý predstavuje smerník na funkciu s parametrom typu double bez návratovej hodnoty.
- Napíšte definíciu typu P_ARR, ktorý predstavuje smerník na pole 10 prvkov typu P_FUN.
- Napíšte, ako by vyzerala definícia premennej pom, ktorá predstavuje:
  - pole smerníkov na typ int;
  - smerník na smerník na pole objektov typu int;
  - smerník na pole smerníkov na funkcie s jedným parametrom typu int, ktoré vracajú smerník na pole smerníkov na typ float.

Typové kvalifikátory

- Pre každý dátový typu v jazyku C existuje niekoľko tzv. kvalifikovaných verzií.
- Kvalifikovaný dátový typ má v porovnaní s nekvalifikovaným typom niekoľko dodatočných vlastností, ktoré závisia od použitého kvalifikátora.
- Jazyk C pozná 4 typové kvalifikátory:
  - const
  - volatile
  - restrict (C99)
  - _Atomic (C11) – modifikátor môže ovplyvniť veľkosť, objektovú reprezentáciu a zarovnanie dátového typu
- Dátový typ môže byť označený viacerými typovými kvalifikátormi, pričom kvalifikátor môže byť umiestnený pred alebo za názvom dátového typu.

Typový kvalifikátor const

- Objekty deklarované s použitím kvalifikátora const môže kompilátor umiestniť do „read-only memory“ a ak program nikdy nepotrebuje adresu takéhoto objektu tak nemusí byť ani špeciálne uložený.
- Objekty (l-hodnoty) s kvalifikátorom const musia byť inicializované počas definície, neskôr nemôžu byť modifikované:

```
const int n = 10;
~~n = 2;~~
```

- V prípade, že má štruktúra (alebo union) aspoň jedeného člena kvalifikovaného ako const, nemôžeme na takúto štruktúru aplikovať operátor priradenia (s výnimkou inicializácie).
- V prípade použitia kvalifikátora const pri formálnom parametri funkcie tento znamená, že skutočný parameter (argument) funkcie nemôže byť funkciou zmenený.
- Výrazy s kvalifikátorom const **nie sú konštantné výrazy** (na rozdiel od C++), t.j. nemôžu byť použité na inicializáciu statických objektov:

```
~~static int s = n;~~
int pole[n]; //jedná sa o VLA
```

Typový kvalifikátor const – príklady

- Štruktúra s konštantným členom:

```
struct A {
    const int clenA;
    int clenB;
};

struct A a = { 1, 1}, b = a, c;
~~e = b;~~
```

- Funkcie s konštantnými parametrami:

```
void fun1(const int n, const struct A pole[]);
void fun2(int n, const struct A * pole[]);
void fun3(int n, struct A const * pole[]);
void fun4(int n, struct A * const pole[]);
void fun5(int n, const struct A * const pole[]);
void fun6(int n, struct A * pole[const]); //C99
void fun7(int n, const struct A * const pole[const]); //C99
void fun8(int n, const struct A * const pole[const static 10]); //C99
```

Typový kvalifikátor volatile

- Kvalifikátor volatile indikuje, že hodnota príslušnej premennej sa môže zmeniť aj v prípade, že z kódu to nie je zrejmé.
- Tento kvalifikátor vynúti, že hodnota premennej (výrazu) bude pred každou operáciu načítaná priamo z operačnej pamäte.
- Tento modifikátor zabraňuje niektorým formám optimalizácie vykonávanej kompilátorom.
- Syntax – obdobná ako v prípade const.
- Príklad:

```
int koniec = 0;
while (!koniec)
    ;
```

- Typické využitie: viacvláknové aplikácie, Memory-mapped I/O

Typový kvalifikátor volatile – vplyv na optimalizáciu

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

int main(int argc, char** argv) {

    double d = 0;
    clock_t start = clock();
    for (int i = 0; i < 100000000; i++)
       d += d*i;
    clock_t stop = clock();
    printf("Klasicka premenna: %.2f s.\n", (double)(stop - start)/CLOCKS_PER_SEC);

    volatile double vd;
    start = clock();
    for (int i = 0; i < 100000000; i++)
        vd += vd*i;
    stop = clock();
    printf("Premenna volatile: %.2f s.\n", (double)(stop - start)/CLOCKS_PER_SEC);

    return 0;
}
```

- Bez optimalizácie

```
Klasicka premenna: 0.53 s.
Premenna volatile: 25.57 s.
```

- S optimalizáciou

```
Klasicka premenna: 0.00 s.
Premenna volatile: 25.15 s.
```

Typový kvalifikátor restrict (C99)

- Typový modifikátor restrict môže byť aplikovaný len na **ukazovatele** (**smerníky**) **na objektové typy** (nie funkcie):

```
int * restrict p1;
restrict int * p2;
int restrict * p3;
void (* restrict fun)();
```

- Kvalifikátor hovorí, že k objektu na ktorý ukazuje daný smerník sa bude pristupovať len cez **tento jediný smerník** (o dodržanie tohto pravidla sa musí postarať programátor). Nedodržanie tohto pravidla vedie k nedefinovanému správaniu.
- Syntax – ak je to možné, tak obdobná ako v prípade const.
- Účel – zlepšenie optimalizácie.

Typový kvalifikátor restrict (C99) – vplyv na optimalizáciu (1)

```
1  #include <stdio.h>
2  #include <time.h>
3
4  void fun1(int n, double * pA, double * pB) {
5      for (int i = 0; i < n; i++) {
6          *pA += *pB;
7          pA++;
8          pB++;
9      }
10 }
11
12 void fun2(int n, double * restrict pA, double * restrict pB) {
13     for (int i = 0; i < n; i++) {
14         *pA += *pB;
15         pA++;
16         pB++;
17     }
18 }
```

Typový kvalifikátor restrict (C99) – vplyv na optimalizáciu (2)

```
20  int main(int argc, char** argv) {
21  #define MAXSIZE 100000000
22
23      static double pole[MAXSIZE];
24
25      clock_t start = clock();
26      fun1(MAXSIZE / 2, pole, pole + MAXSIZE / 2);
27      clock_t stop = clock();
28      printf("Klasicke parametre: %.2f s.\n", (double)(stop - start)/CLOCKS_PER_SEC);
29
30      start = clock();
31      fun1(MAXSIZE / 2, pole, pole + MAXSIZE / 2);
32      stop = clock();
33      printf("Parametre restrict: %.2f s.\n", (double)(stop - start)/CLOCKS_PER_SEC);
34
35      return 0;
36  #undef MAXSIZE
37  }
```

```
Klasicke parametre: 0.37 s.
Parametre restrict: 0.16 s.
```

Práca na úrovni bitov – bitové operácie (1)

- Pomocou bitových operátorov môžeme manipulovať s bitmi v **celočíselných** dátových typoch.
- Bitové operácie sa používajú na tvorbu bitových masiek, pomocou ktorých je možné získať alebo nastaviť hodnotu konkrétneho bitu nejakej l-hodnoty.
- Pri použití bitových operátorov sa na operandy aplikujú implicitné konverzie typu „usual arithemtic conversions“.
- Určte typ nasledujúcich výrazov:
  - ~1
  - ~(char)1
  - (char)~1

Operátor	Význam	Príklad	Výsledok
~	bitová negácia	~0000 1111	1111 0000
<<	bitový posun doľava	1110 0011 << 2	1000 1100
>>	bitový posun doprava	1110 0011 >> 2	0011 1000
&	bitový súčin	1110 0011 & 0000 1111	0000 0011
^	bitový xor	1110 0011 ^ 0000 1111	1110 1100
|	bitový súčet	1110 0011 | 0000 1111	1110 1111

Práca na úrovni bitov – bitové operácie (2)

- Predpokladajme nasledujúcu definíciu:

  int dword = 123;

- Zistenie hodnoty n-tého bitu:

  dword & (1 << n)

  (dword >> n) & 1

- Nastavenie hodnoty n-tého bitu:
  - na hodnotu 1:

    dword | (1 << n)

  - na hodnotu 0:

    dword & ~(1 << n)

  - na opačnú hodnotu:

    dword ^ (1 << n)

Práca na úrovni bitov – bitové polia

- Pod bitovým poľom sa rozumie člen štruktúry alebo union-u s presne definovaným počtom bitov.
- Príklad štruktúry s bitovým poľom:

```
struct bits {
        unsigned int a:3;
        unsigned int :5;
        unsigned int b:7;
        unsigned int :0;
        unsigned int c:4;
};
```

- Bitové pole môže mať len nasledujúce typy:
  - unsigned int (unsigned int b:2, prípustné hodnoty: 0..3);
  - signed int (signed int b:2, prípustné hodnoty: -2..1);
  - int – správa sa buď ako unsigned int alebo signed int (je to implementačne závislé);
  - _Bool (C99) (**_Bool b:1**, prípustné hodnoty: 0..1);
  - podpora ostatných typov je implementačne závislá.
- Na bitové pole nie je možné aplikovať operátori &, sizeof a _Alignas (C11).

- Prečo nie je možné získať adresu bitového poľa?

Variadické funkcie

- Variadické funkcie sú funkcie, ktoré môžu mať ľubovoľný počet parametrov. Typickým príkladom sú rodiny funkcií typu printf() a scanf().
- Variadická funkcia musí mať aspoň jeden pomenovaný parameter, pričom variadické parametre sa uvádzajú symbolom „...“
- Variadické argumenty sa spracúvajú pomocou makier definovaných v hlavičkovom súbore <stdarg.h> (http://en.cppreference.com/w/c/variadic):
  - va_start
  - va_arg
  - va_copy (C99)
  - va_end
- Tieto makrá pracujú s parametrom typu:
  - va_list

VARIADICKÉ FUNKCIE – UKÁŽKA

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

int minInt(int count, int a,...) {
    int min = a;
    va_list args;
    va_start(args, a);
    for (int i = 1; i < count; i++) {
        int akt = va_arg(args, int);
        if (akt < min) {
            min = akt;
        }
    }
    va_end(args);
    return min;
}

int main(int argc, char** argv) {
    printf("Minimum z 2 cisel je %d.\n", minInt(2, 10, 20));
    printf("Minimum z 3 cisel je %d.\n", minInt(3, 10, 20, 5));
    printf("Minimum zo 4 cisel je %d.\n", minInt(4, 10, 20, 5, 7));
    return 0;
}
```

```
Minimum z 2 cisel je 10.
Minimum z 3 cisel je 5.
Minimum zo 4 cisel je 5.
```

Preprocesor

- Spracúva zdrojové súbory projektu predtým, ako sa dostane k slovu kompilátor.
- Ide o predspracovanie súborov na úrovni zdrojových textov.
- Činnosti preprocesora:
  - odstraňuje komentáre;
  - vkladá súbory;
  - nahradzuje symboly;
  - vykonáva podmienený preklad.

Direktívy preprocesora

- Činnosť preprocesora je možné riadiť pomocou direktív:
  - #define
  - #elif
  - #else
  - #endif
  - #error
  - #if
  - #ifdef
  - #ifndef
  - #include
  - #line
  - #pragma
  - #undef
- Direktíva musí byť prvým nebielym znakom na danom riadku

Direktívy #include a #error

- Direktíva #include
  - vloží určený súbor na dané miesto v zdrojovom kóde
  - #include <soubor.h>
  - #include "soubor.h"

- Direktíva #error [hlásenie]
  - generuje zadané hlásenie ako výstup prekladu

Preprocesor – makrá

- Definovaním makra môžeme do programu zaviesť pomenovanú konštantu.
- Každý výskyt postupnosti znakov, ktoré zodpovedajú názvu makra sa nahradí hodnotou makra.
- Definovanie makra – direktíva #define:
  - #define identifikator [hodnota]
- Zrušenie makra – direktíva #undef
  - #undef identifikator
- Príklady:
  - #define SIZE 1000 /*velkost pola*/
  - #define TWO_PI (2 * 3.141592f)

Makro ako jednoduchá „funkcia“

- Syntax:
  - #define identifikator([zoznam_parametrov]) [telo]
  - #define identifikator([zoznam_parametrov,]...) [telo] //C99
- Príklady:

  #define MAX(a, b) (((a) > (b)) ? (a):(b))
- Makro sa rozvinie v mieste použitia, na čo treba dávať pozor:

  #define SQR(x) (x * x)

  SQR(a + 1) = (a + 1 * a + 1) = (2 * a + 1)
  - správna verzia makra:

    #define SQR(x) ((x) * (x))
- V prípade viacriadkového makra spájame riadku prostredníctvom spätného lomítka “\”.

- Môžeme pomocou makra vytvoriť „rekurzívnu funkciu“?

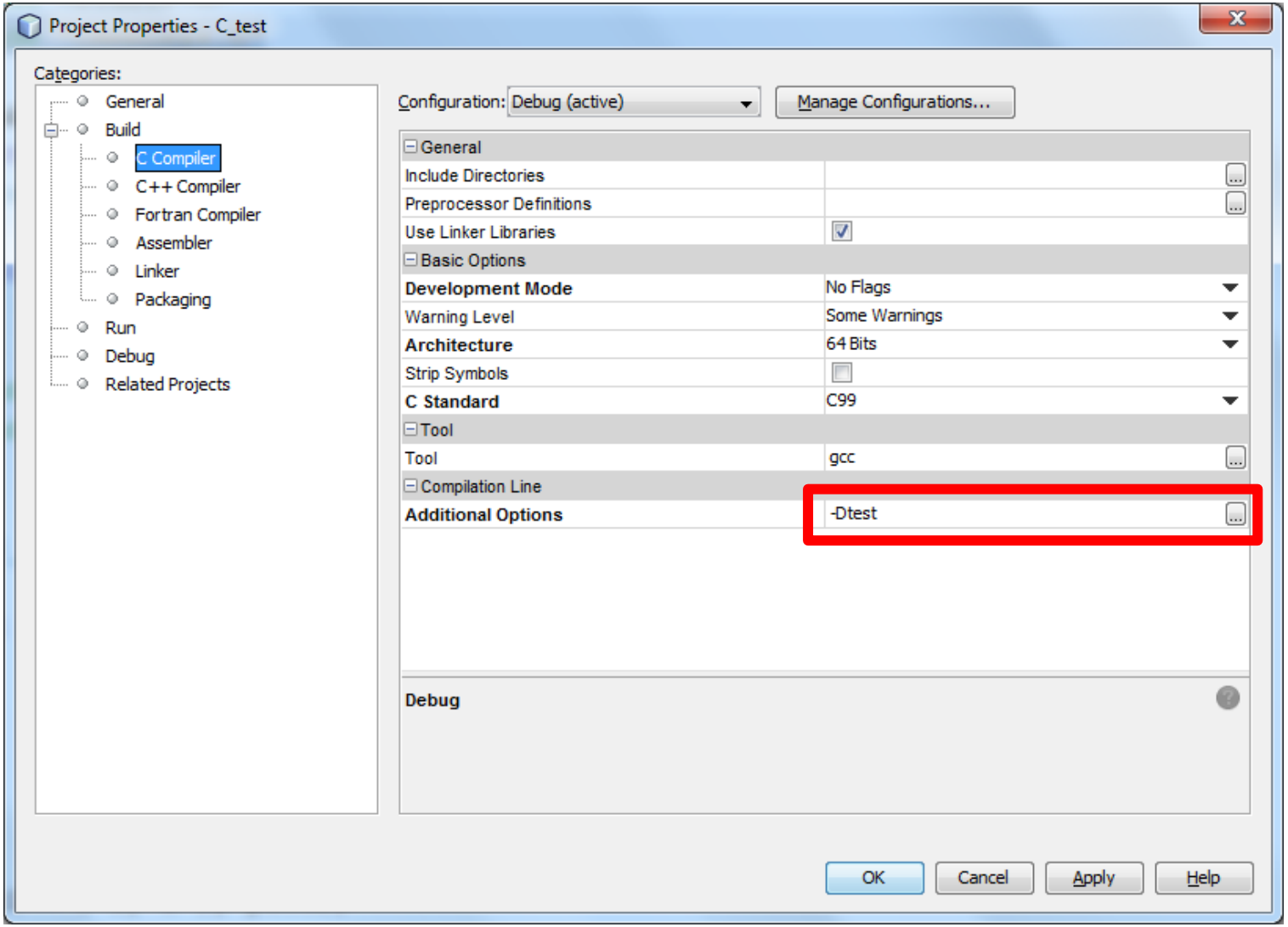
Preprocesor – podmienený preklad

- Pomocou nasledujúcich direktív je možné riadiť preklad zdrojových kódov (napr. použitie POSIX socketov pre Unix a Winsock pre Windows):
  - #ifdef
  - #ifndef
  - #if
  - #elif
  - #else
  - #endif
- Podmienený preklad sa obyčajne zabezpečuje definovaním makier bez hodnôt:

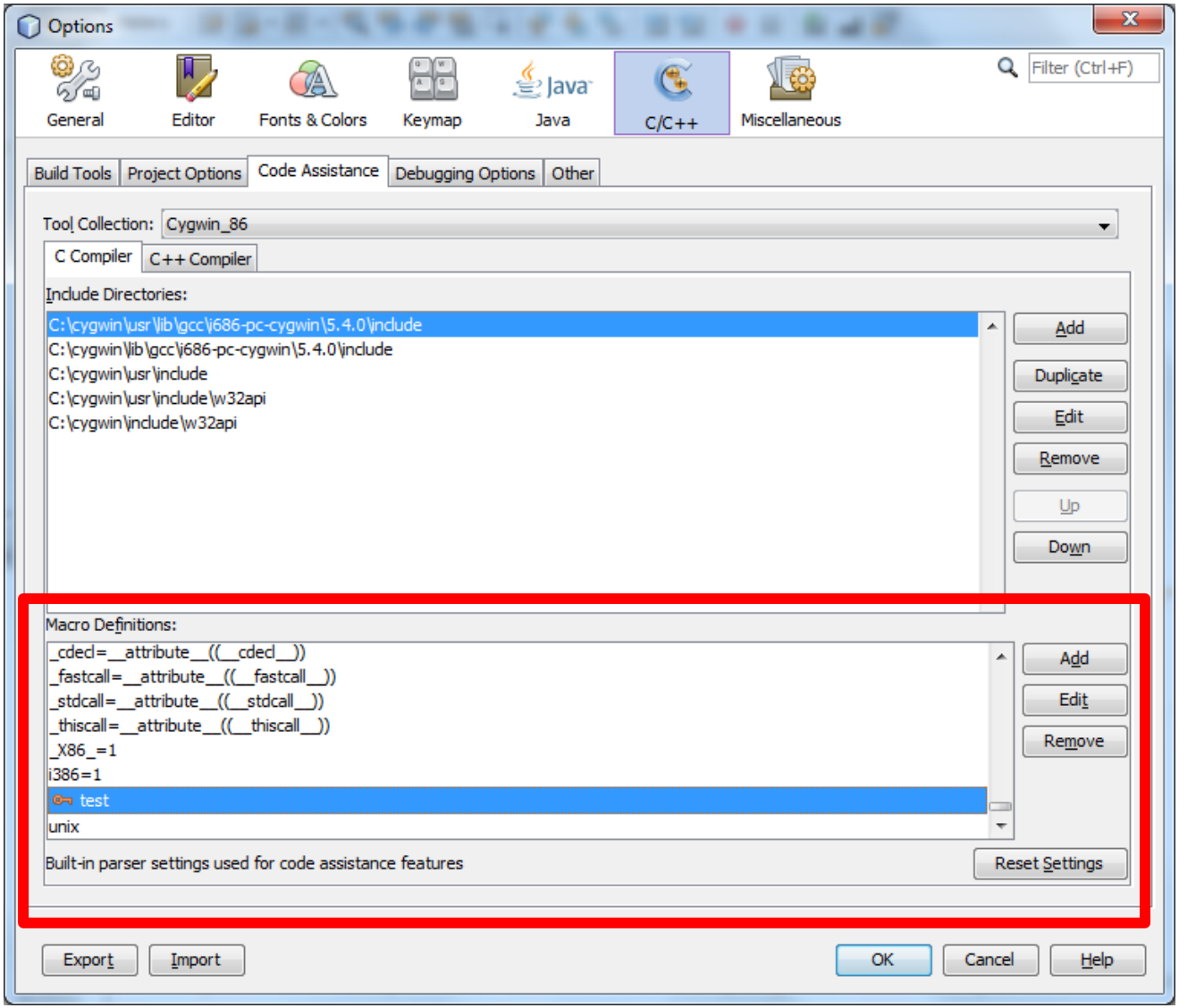
```
#define DEBUG

#ifdef DEBUG
    printf("debug message\n");
#endif
```

NETBEANS – DEFINOVANIE MAKRA NA ÚROVNI KOMPILÁTORA



NETBEANS – DEFINOVANIE MAKRA PRE ÚČELY ASISTENTA KÓDU



Podmienený preklad – príklad

```c
#include <stdio.h>
#ifndef __STDC_NO_COMPLEX__
#include <complex.h>
#include <math.h>

double complex nacitajKomplexneCislo() {
    double real, imag;
    scanf("%lf%lf", &real, &imag);
    return real + imag*I;
}
int main(int argc, char** argv) {
    double complex a;
    double complex b;

    printf("Zadaj realnu a imagiarnu cast komplexneho cisla\n");
    a = nacitajKomplexneCislo();
    b = 1 + 1*I;
    printf("Sucet komplexnych cisel %.2f%+.2fi a %.2f%+.2fi = %.2f%+.2fi.\n",
            creal(a), cimag(a), creal(b), cimag(b),
            creal(a + b), cimag(a + b));
   return 0;
}
#else
int main(int argc, char** argv) {
    printf("Komplexne cisla nie su podporovane kompilatorom.\n");
}
#endif
```

Direktívy #line a #pragma

- Direktíva #line:
  - umožňuje zmeniť číslo riadku (a všetkých riadkov za nim) a názov prekladaného súboru.

```
1   #define FILE_NAME "file.c"
2   int main(int argc, char** argv)
3   {
4   #line 100 FILE_NAME
5       printf("Nachadzam sa na riadku %d v subore %s.\n", __LINE__, __FILE__);
6       return 0;
7   }
```

```
Nachadzam sa na riadku 100 v subore file.c.
```

- Direktíva #pragma:
  - implementačne závislá direktíva;
  - ak jej preprocesor nerozumie, úplne ju ignoruje, bez akéhokoľvek hlásenia o chybe;
  - používa sa pre zadanie direktív pre špecifický preprocesor konkrétneho dodávateľa.

Preddefinované makrá

- Pre každý prekladaný súbor sú preddefinované nasledujúce makrá:
  - __LINE__ aktuálny riadok v zdrojovom súbore
  - __FILE__ meno zdrojového súboru
  - __TIME__ čas prekladu súboru
  - __DATE__ dátum prekladu súboru
  - __STDC__ má hodnotu 1, ak prekladáme ako ANSI C, inak nedefinované
  - __STDC_VERSION__ (C95) konštanta typu long, ktorej hodnota sa zvyšuje s každou verziou štandardu jazyka C:
    - 199409L (C95)
    - 199901L (C99)
    - 201112L (C11)
  - __STDC_HOSTED__ (C99) celočíselná konštanta 1 ak implementácia beží pod operačným systémom, inak celočíselná konštanta 0