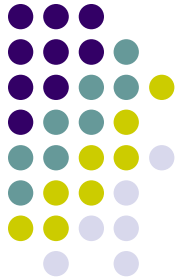
NIT

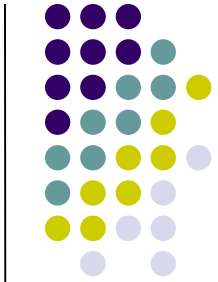
Програмски језик Це - неформално



- Императивни језик (насупрот декларативном)
  Наредбе се извршавају редом и у складу са значењем мењају **стање** програма.
  - Стање
    - Променљиве и парчад меморије - објекти
  - Наредбе
    - Искази и изрази
    - Функције
    - Блокови

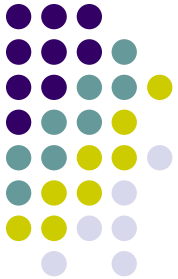
Програмски језик Це - неформално

- — 127 нивоа угнеждења блокова
- — 63 нивоа угнеждења условних укључивања датотека
- — 63 угнеждених декларатора у заградама унутар пуног декларатора
- — 63 угнеждених израза у заградама унутар пуног израза
- — 63 значајних почетних знакова код интерних идентификатора и назива макроа
- — 31 значајних почетних знакова код екстерних идентификатора
- — 4095 екстерних идентификатора у једној јединици превођења
- — 4095 макро идентификатора у једној јединици превођења
- — 511 идентификатора са блоковским досегом унутар једног блока
- — 127 параметера у једној дефиницији функције
- — 127 параметара (аргумената) у једном позиву функције
- — 127 параметара у једној макро дефиницији
- — 127 параметара у једном макро позиву
- — 4095 знакова у једној (логичкој) линији изворног кода
- — 4095 знакова у стринг константи (након спајања)
- — 65535 бајтова у једном објекту (in a hosted environment only)
- — 15 нивоа угнеждења укључених датотека (путем #include директиве)
- — 1023 кејс (case) лабела у једној свич (switch) наредби
- — 1023 чланова структуре или уније
- — 1023 елемената у једном набројивом типу
- — 63 нивоа угнеждења у декларацији једне структуре или уније



Основни типови у Цеу

NIT



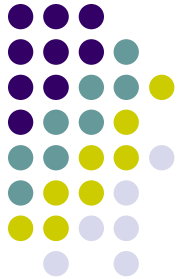
Типови?

- Тип је особина променљиве која одређује које вредности она може да има и шта се са њом може радити.
- Це је експлицитно типизиран језик -> свакој променљивој тип мора бити задат од стране програмера -> приликом њене декларације
- Це је статички типизиран језик -> једном постављен тип се не може мењати
- Свака променљива има тип, сваки израз има тип, (скоро) све има тип
- Основни типови - изведени типови
- Дефинисан скупом вредности и операцијама
- Тип не мора одређивати физичку репрезентацију вредности

Целобројни (интеџер) типови

- **char**
  - миниму 8 бита, најчешће баш 8 бита
    - уобичајен опсег [-128, 127] ако је означен или [0, 255], ако је неозначен
  - захтева се да буде „најмања адресива јединица” циљне платформе - **бајт**, тј. да његова величина буде једнака ширини меморије.
- **short (int)**
  - минимум 16 бита, најчешће баш 16 бита.
    - уобичајен опсег [-32768, 32767] ако је означен или [0, 65535], ако је неозначен
- **int**
  - минимум 16 бита, најчешће 32 бита.
    - уобичајен опсег -2147483648, 2147483647] ако је означен или [0, 4294967295], ако је неозначен
  - уобичајено је да представља „најприроднију” величину за циљну платформу.
- **long (int)**
  - бар 32 бита, најчешће баш 32 бита.
    - уобичајен опсег -2147483648, 2147483647] ако је означен или [0, 4294967295], ако је неозначен
- **long long (int)**
  - бар 64 бита, најчешће баш 64 бита.
  - део C99 стандарда

NIT



Модификатори

- Могући модификатори основних типова:
  - **char**
    - **signed**
    - **unsigned**
  - **int**
    - **signed**
    - **unsigned**
    - **short**
    - **long**

На шта се ослонити када величине интеџера варирају од платформе до платформе?

Одговор је у `stdint.h` заглављу. Ако желимо да интеџер тип буде:

- тачно одређене величине

`int`**`N`**`_t и uint`**`N`**`_t` где N може бити било који природан број

Постојање ових типова није обавезно, али ако је неки од интеџер типова широк 8, 16, 32 или 64 бита онда одговарајући тип мора бити дефинисан. Шта су последице овога?

- бар одређене величине
  - најмањи бар толике величине

    `int_least`**`N`**`_t и uint_least`**`N`**`_t,` за **N** 8, 16, 32 и 64 постоје увек.
  - најбржи бар толике величине

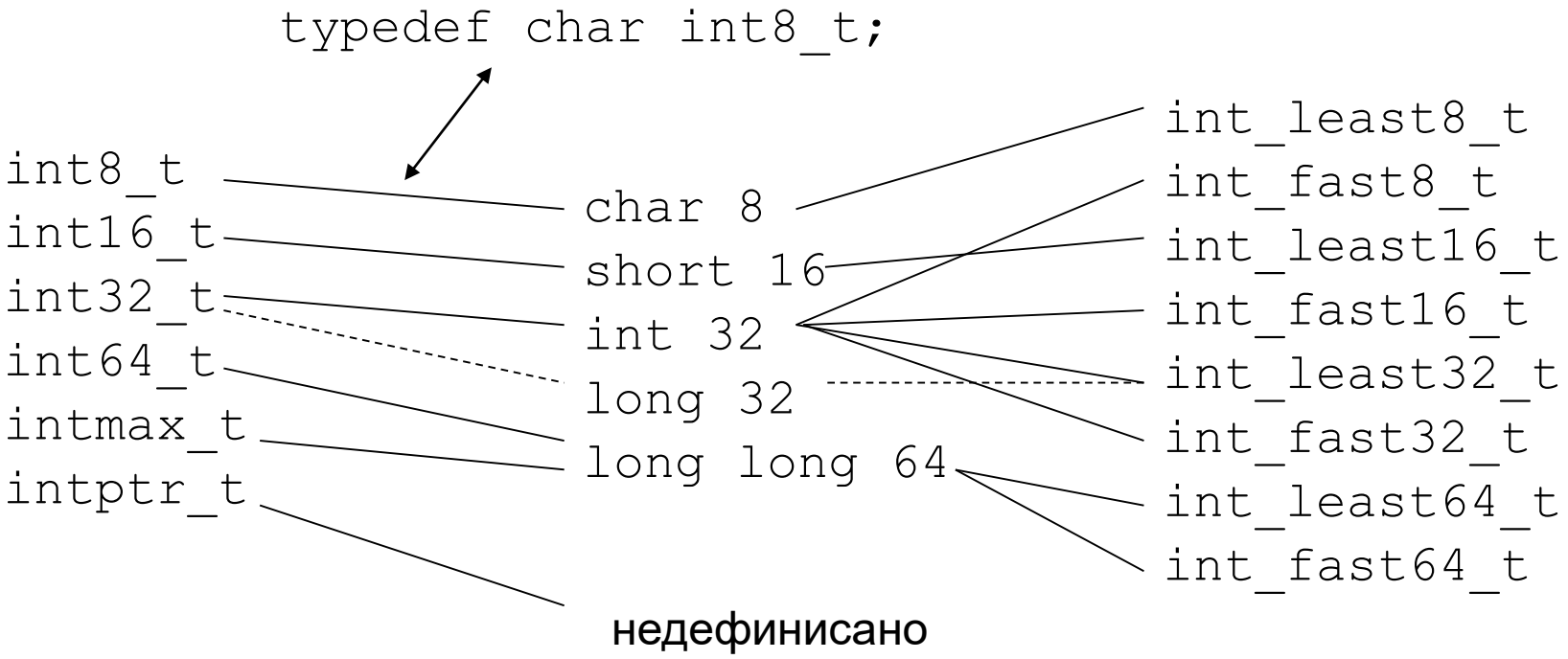
    `int_fast`**`N`**`_t и uint_fast`**`N`**`_t,` за **N** 8, 16, 32 и 64 постоје увек.
- довољно велик да у себи држи показивач, тј. адресу

`intptr_t и uintptr_t,` али нису обавезни. (Зашто?)

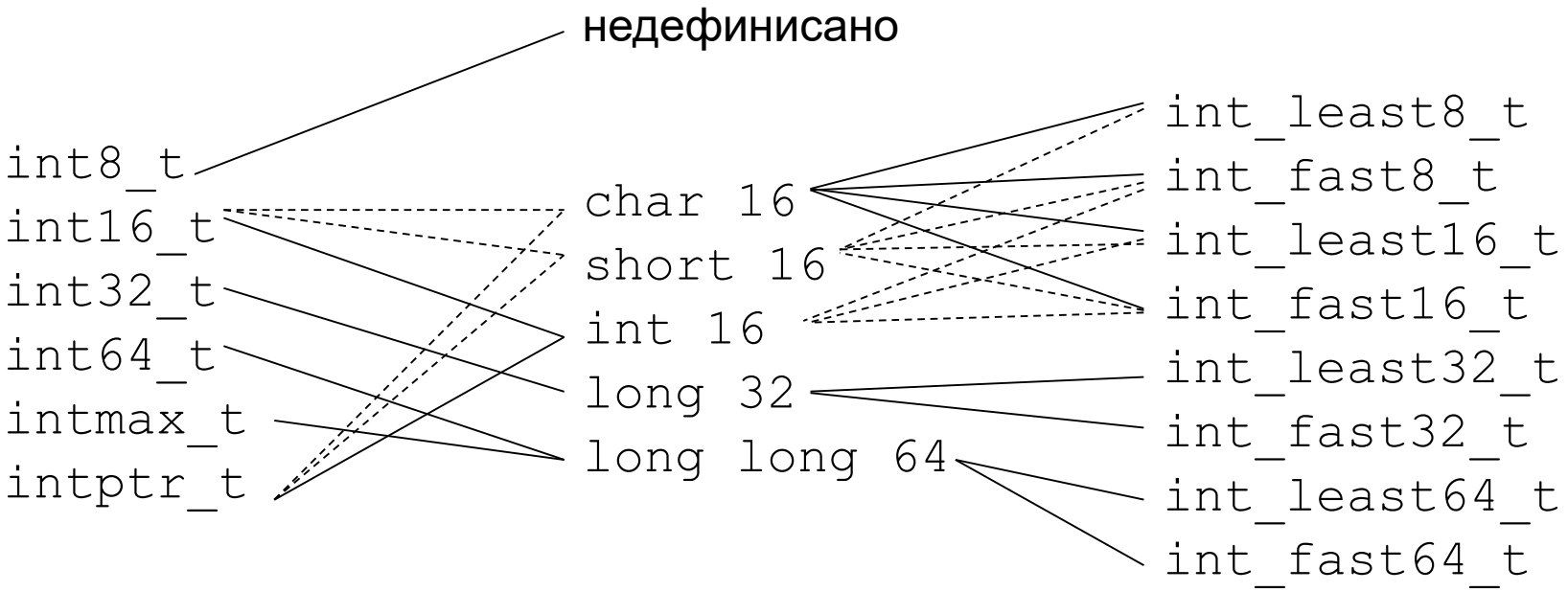
- највећи интеџер тип

`intmax_t и uintmax_t`

Ширина 8 бита
Регистар 32 бита
Адресни простор 70 бита



Ширина 16 бита
Регистар 16 бита
Адресни простор 16 бита



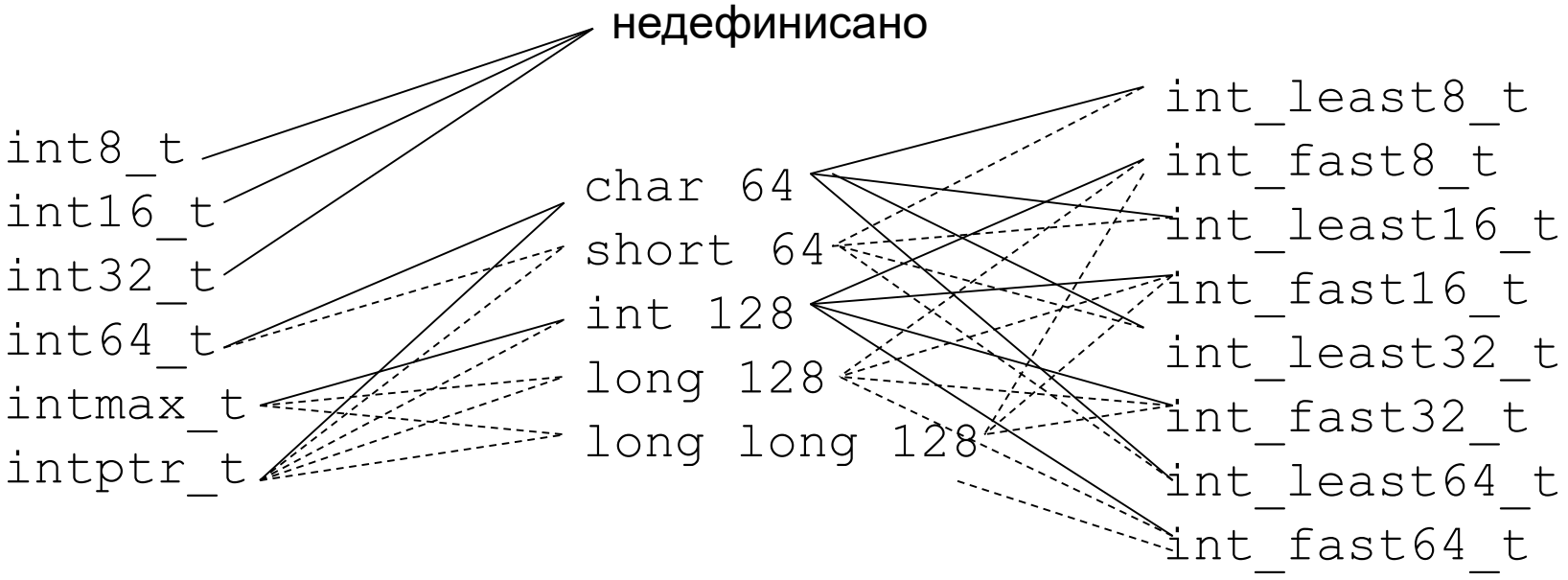
Ширина 64 бита
Регистар 128 бита
Адресни простор 16 бита

```
int8_t
int16_t
int32_t
int64_t
intmax_t
intptr_t
```

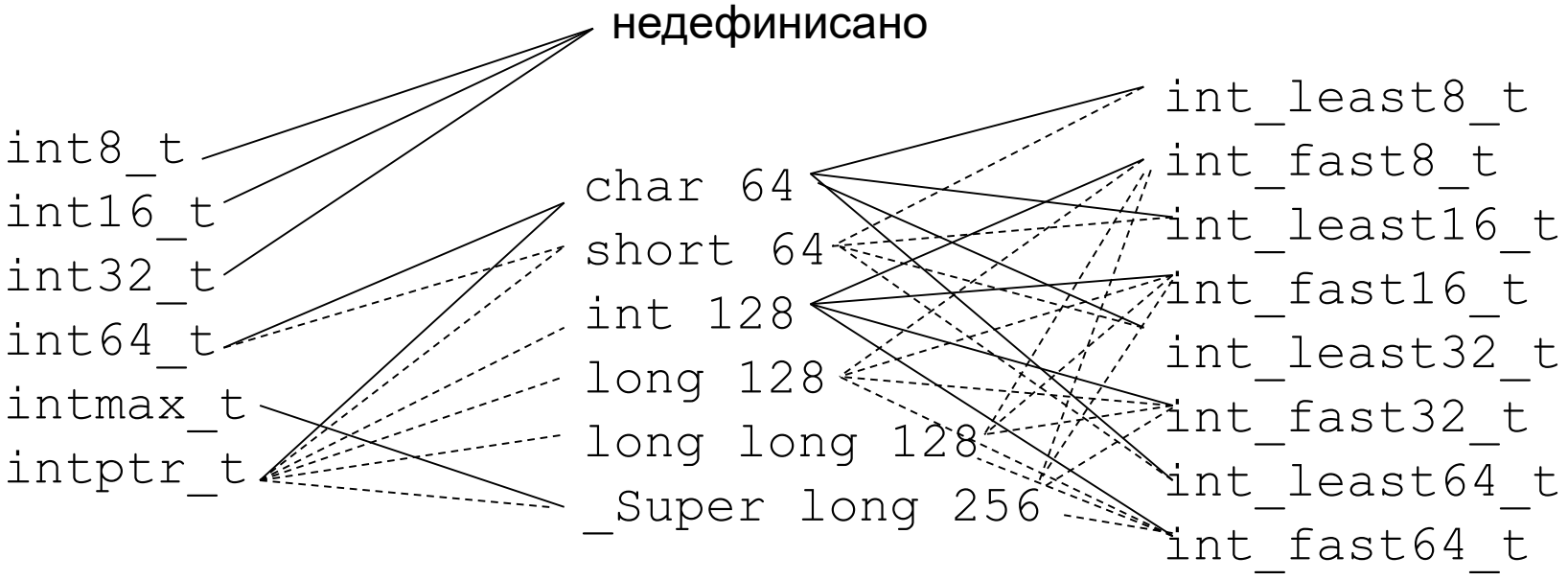
```
char 64
short 64
int 128
long 128
long long 128
```

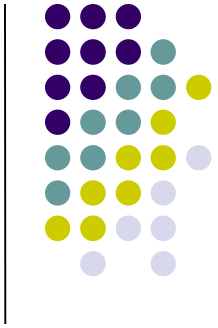
```
int_least8_t
int_fast8_t
int_least16_t
int_fast16_t
int_least32_t
int_fast32_t
int_least64_t
int_fast64_t
```

Ширина 64 бита
Регистар 128 бита
Адресни простор 16 бита



Ширина 64 бита
Регистар 128 бита
Адресни простор 16 бита



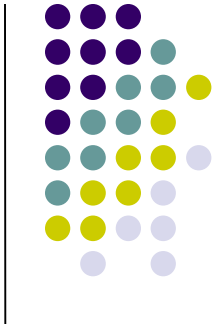


```
int16_t niz[30000];
```

16b
```
short niz[30000];
```

16b
```
int niz[30000];
```

Undefined int16_t

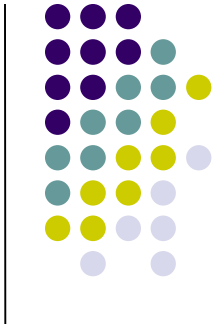


```
int_least16_t niz[30000];
```

16b
```
short niz[30000];
```

16b
```
short niz[30000];
```

64b
```
short niz[30000];
```



```
int_least16_t niz[30000];
int_fast16_t i;
for (i=0; i<30000; i++)
```

16b
```
short niz[30000];
int i; // 32b
```

16b
```
short niz[30000];
short i; // 16b
```

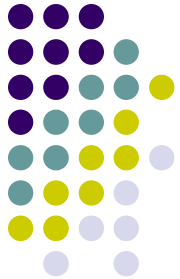
64b
```
short niz[30000];
int i; // 128b
```

NIT

Да ли је char тип означен или неозначен?

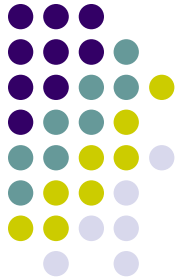
- Це стандард оставља конкретном компајлеру да специфицира да ли је char тип подразумевано означен или неозначен. Рецимо:
  - Код компајлера за x86 (GNU/Linux и Microsoft Windows) char тип је обично означен
  - Код компајлера за PowerPC и ARM процесоре char тип је обично неозначен
- Преносивос кода може бити нарушена ако се ослања на подразумевану означеност!
- По C99 стандарду, у заглављу limits.h мора бити дефинисан симбол CHAR_MIN. Ако је нула, char је подразумевано неозначен, и обрнуто.

signed или unsigned char?



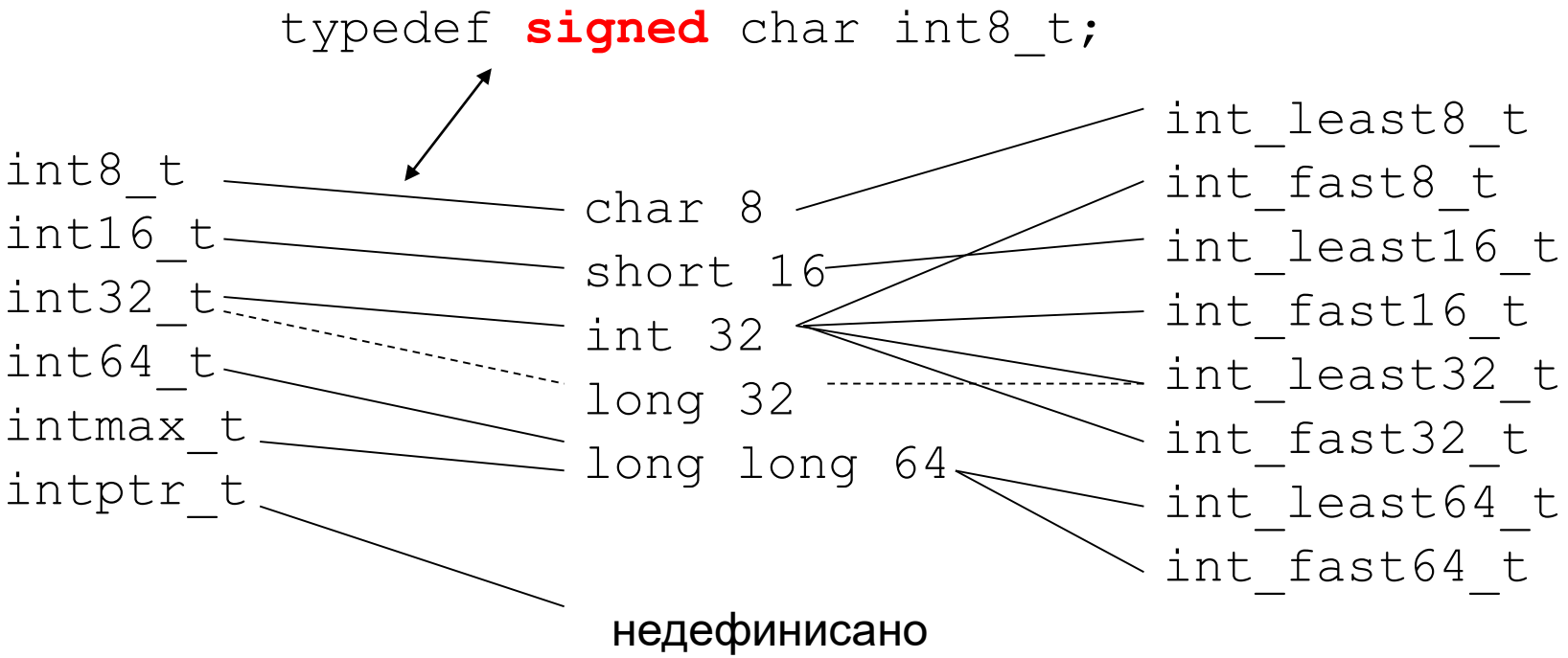
```
#include <stdio.h>
#include <limits.h>

int main (void)
{
  if (CHAR_MIN == 0)
  {
    printf("char is unsigned.”);
  }
  else
  {
    printf("char is signed.”);
  }
  return 0;
}
```

Решење за већ постојећи код

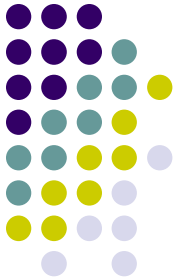


- Основно решење је прерадити код
- Неки компајлери, као што је GCC нуде опцију којом се одређује која ће бити подразумевана означеност char типа:
  `-fsigned-char` и `-funsigned-char`.

Типови у stdint.h воде рачуна о означености char типа



NIT



Показивачи

Најбоље је на показиваче гледати као на посебан тип.

У том смислу, звездицу залепити за тип на који се показује:

✓ **`int* p;`** уместо: ✗ **`int *p;`**

Показивачи на објекте:

```
float* p;
p = &x;
*p = y;
z = *p;
```

Показивачи на функције:

```
int* (*p)(int a, float b);
int* foo(int x, float y);
p = foo;
ip = p(n, m);
```

Нул (NULL) показивач:

```
#include <stddef.h>
p = NULL;
```

✗
```
*p = y;
```

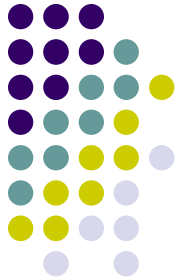
Показивач на воид (void):

```
void* p;
p = &x;
```

✗
```
*p = y;
```

✓
```
*(int*)p = y;
```

Кориснички типови typedef наспрам #define



За дефинисање нових типова може се користити #define и typedef

```
#define INT_PTR int*
typedef int* int_ptr;
```

Пример употребе:

```
INT_PTR ptr1, ptr2;
int_ptr ptr3, ptr4;
```

Зачкољица: после претпроцесора прва линија изгледа овако

```
int*  ptr1, ptr2;
```

Шта ту није у реду?

Савет 1: за нове типове користите typedef

Савет 2: свака декларација променљиве у засебном реду

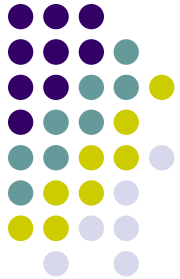
Типови са покретним зарезом (float типови)

- Типови са покретним зарезом служе за изражавање реалних бројева.
- Постоје три типа са покретним зарезом који се разликују по величини и прецизности:
  - **float** – заузима 32 бита
  - **double** – „дупла прецизност” заузима 64 бита
  - **long double** (увео C99)
- У стандардном заглављу float.h дефинисане су минималне и максималне вредности сваког типа са покретним зарезом, заједно са још неким корисним вредностима, као што су прецизност и слично.

Тип	Знак	Експонент	Мантиса	Укупно бита
float	1	8	23	32
double	1	11	52	64

NIT

Набројиви типови - Енумерације



- Уведени у C89/90 стандарду
- Представљају једну вредост из скупа наведених вредности. Вредности су Це симболи.
- Како то ради:
  - свакој симболичкој вредности из енума (енум типа) додељује се јединствени цео број
  - свако место где се спомиње симбол из енума замењује се са одговарајућим int литералом од стране компајлера

Енум пример

- Првом наведеном симболу придружује се 0, осим ако експлицитно није придружен неки други број.

```
enum Dani
{
  PONEDELJAK = 1,
  UTORAK,
  SREDA,
  CETVRTAK,
  PETAK,
  SUBOTA,
  NEDELJA
};
```

- Сваком наредном симболу придружује се број придружен претходном симболу увећан за 1

- Неки пут (обично у старијим кодовима) користи се #defines да би се направио скуп симбола

```
#define PONEDELJAK 1
#define UTORAK     2
#define SREDA      3
#define CETVRTAK   4
#define PETAK      5
#define SUBOTA     6
#define NEDELJA    7
```

Досег симбола из енум типа

- Дефинисани симболи упадају у тренутан досег

```
enum PeriniDrugari { SIMA, DJURA, STEVA };

SIMA // овако се обраћамо том симболу
```

- Што понекад може правити проблем:

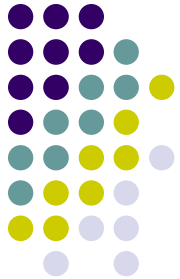
```
enum PeriniDrugari { SIMA, DJURA, STEVA };
enum MikiniDrugari { DJOLE, SIMA, MILE };
SIMA // на шта се мисли овде?
```

- Па је препорука да се уради ово:

```
enum PeriniDrugari { PD_SIMA, PD_DJURA, PD_STEVA };
enum MikiniDrugari { MD_DJOLE, MD_SIMA, MD_MILE };
PD_SIMA // Сада је јасно.
```

NIT

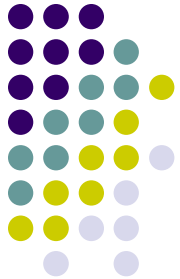
Величина и означеност енум типа



- Енум тип се своди на целобројан тип.
- Остављено је компајлеру да одлучи на који конретан тип ће свести одређени енум тип. За сваки енум може бити другачије.
- Величина тог целобројног типа стандардом је ограничена на два начина:
  - Мора бити довољно велик (широк) да би сместио све вредности које су у енуму дефинисане
  - али не већи од величине int типа (јер су симболи int типа).
- Неки компајлери ипак подржавају енуме који су већи од int типа (тј. могу свести енуме и на типове веће од int).

NIT

Величина и означеност енум типа



- Као величина целобројног типа тако и његова означеност је остављена компајлеру на избор.
- Такође не мора бити иста за све енуме.
- Обично ако постоји симбол са негативном вредношћу енум ће бити представљен означеним типом, у супротном – неозначеним. **Међутим**, то може да бидне ал` не мора да значи!!!
- Зато се не ослањати на ово!

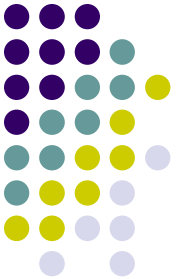
Величина и означеност енум типа

- Ово постаје проблем само ако са енум вредностима радимо аритметику, тј. ако их мешамо са интеџерима.
- Зато то у Цеу треба да избегавамо. Једино је сигурно да користимо енум променљиве за складиштење и поређење вредности (односно симбола), и то само оних које су у том енуму дефинисане.
- Мада, рађење аритметике са енумима је некада врло згодно.
- Сам Це нам не олакшава да избегнемо мешање, јер дозвољава ово:

```
Dani x = 75;
Mesec y = PONEDELJAK;
```

NIT

Це++ излет



- enum class помаже у неким случајевима:
- 1) досег

```
enum class PeriniDrugari { SIMA, DJURA, STEVA };
enum struct MikiniDrugari { DJOLE, SIMA, MILE };
PeriniDrugari::SIMA
```

- 2) целобројни тип у основи енума

```
enum class Primer1 { A, B, V }; // своди увек на int
enum class Primer2 : long { X, Y, Z }; // сада на long
enum class Primer3;
...
enum class Primer3 { P, Q, R }
```

- 3) Конверзија

```
// Ово сада не може (без експлицитне конверзије)
Dani x = 75;
Mesec y = PONEDELJAK;
```

NIT

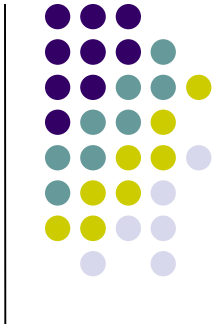
Конверзије типова

- Када се два различита типа комбинују у одређеним операцијама настаје потреба за конверзијом типова
- Конверзија може бити:
  - **Имплицитна**
    - **Додела**
    - **Позив функције**
      - **Прослеђивање параметара**
      - **Повратна вредност**
    - **Аритметичке конверзије** – Најчешћи извор проблема ако се добро не познаје.
    - **Промоција типова**
  - **Експлицитна**
    - **Такозвано „кастовање”** – `(int)x`

Типови константи (литерала)

- **Сваки операнд (израз) има тип!**
- Код литерала тип је одређен суфиксом и постојањем децималне тачке
- Примери:

```
2U                  -37                          'c' // није исто што и "c"
3L 31 31            051          // 41
5UL                 0x2b         // 43
7LL                 0xFFFFFFD1   // -47
11ULL
13.0                3.14159      // 3.14159
17.                 6.02e23      // 6.02 x 10^23
19.0F               1.6e-19      // 1.6 x 10^-19
23.F
29.0L
31.L
```



```
int_least16_t niz[30000];
int_fast16_t i;
for (i=0; i<INT16_C(30000); i++)
```

16b

```
short niz[30000];
int i; // 32b
```

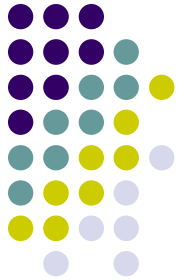
16b

```
short niz[30000];
short i; // 16b
```

64b

```
short niz[30000];
int i; // 128b
```

NIT



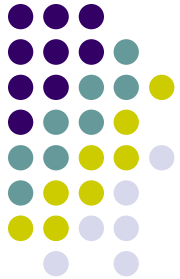
Промоције типова

Промоција типова је специјалан случај имплицитне конверзије. Промоције:

- Промоције интеџера
  - У операцијама где учествују целобројни типови мањи од int (дакле: char или short) операнди се прво промовишу у одговарајући int (означени или неозначени), обавља се операција, а резултат се своди на почетни тип одсецањем вишка битова.

```
char a;
char b;
char c = a + b;
```

Аритметичке конверзије: пример



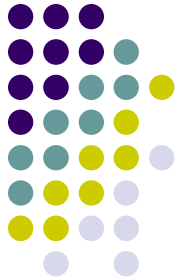
```
double polovina = 1/2;
```

Која ће бити вредност променљиве **`polovina`**?
Зашто?

Ког типа су дељеник и делилац?
Ког типа је количник?

Сваки операнд, тј. израз, има тип, па тако и израз „1/2” има тип и тај тип нема везе са типом операнда са леве стране једнакости!

Аритметичке конверзије: пример - наставак



Константе 1 и 2 су интеџер типа.

Ако су оба операнда **int** и тип резултата је **int**. Дакле, целобројни резултат овог израза је 0.

```
double polovina = 1.0/2.0;
```

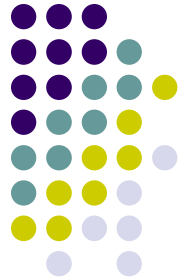
али довољно је и овако:

```
double polovina = 1.0/2;
```

```
У случају да су x и y интеџер типа:
double kolicnik = ((double)x)/y;
```

NIT

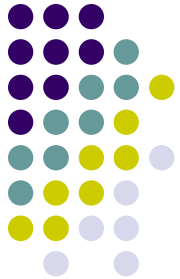
Аритметичке конверзије



- Бројчани типови су поређани по ранговима од најмањег до највећег овако:
  `char, short, int, long, long long, float, double, long double`
- Уколико у бинарној операцији учествују два операнда различитог типа, онда се операнд чији је тип мањег ранга конвертује у тип другог операнда
- Уколико у бинарној операцији типови операнада нису исте означености, онда се операнд чији је тип означен конвертује у неозначени тип.
- Поука: бити јако пажљив са овим!
- Најбоље је да осим конверзије из неког целобројног типа у тип са покретним зарезом увек користимо експлицитне конверзије.
- Компајлер ће обично генерисати упозорења за део проблематичних случајева.

- За детаље погледати поглавље 6.3 стандарда, конкретније: поглавље 6.3.1.8.

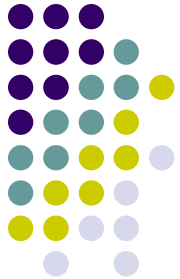
Потенцијални проблеми приликом конверзија



- Опасне конверзије
  - **Губитак вредности**: Конверзија већег (ширег) типа у мањи (ужи)
  - **Губитак знака**: Конверзија између означених и неозначених типова може довести до промене знака
  - **Губитак прецизности**: Конверзија прецизнијег у непрецизнији тип
- Безбедне конверзије
  - Конверзија целобројног типа у већи (шири) целобројни тип
  - Конверзија типа са покретним зарезом у већи (шири и прецизнији) тип са покретним зарезом

NIT

Конверзије показивача

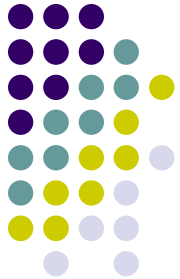


- Конверзије које не ваља правити су:
  - између показивача и интеџер типа.
    - Ова конверзија није дефинисана стандардом, тј. сваки компајлер може изабрати своју дефиницију.
    - Неки пут је неизбежно, рецимо код приступању меморијски мапираним регистрима или другим аспектима специфичним за циљну платформу.
  - између показивача на два различита типа.
    - Проблем може настати услед неодговоарајућег поравнања (поравнање ће бити детаљније обрађено касније)

```
uint8_t p1[4];
uint32_t* p2;
p2 = (uint32_t*)p1;
```

NIT

Конверзије показивача

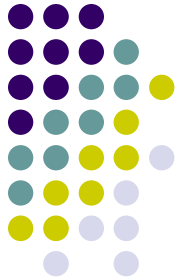


- Валидне конверзије:
  - Из показивача на воид (void*) у показивач на неки други тип, и обрнуто.
  - Конверзија литерала 0 у показивач. То представља једину универзално валидну конверзију из интеџера у показивач. Литерал 0 у контексту показивача представља специјалну вредност која значи „не показујем ни на шта“. Та вредност се зове још и нул вредност, или нул показивач.
  - Нул показивач у показивач на било који други тип.

- Конверзије које морају бити експлицитне:
  - Показивач на објекат било ког типа у показивач на објекат другог типа.
  - Показивач на функцију једног типа у показивач на функцију другог типа.
- Невалидне конверзије:
  - Показивач на функцију у показивач на објекат (или показивач на воид) и обрнуто.

NIT

Нул показивач, плус излет у Це++



- Савет: Не користити литерал 0 у контексту показивача, већ симбол NULL, који је дефинисан у заглављу stddef.h.
- Када пише 0, није одмах јасно да ли се ради о целим бројевима или показивачима.
- Осим тога, стварна вредност нул показивача заправо не мора бити баш 0.

- У Це++-у је проблем већи:
- foo(int), foo(char*) – foo(0) коју функцију зовем?

- Зато у Це++-у сада постоји и посебна вредност (кључна реч) **nullptr**.