



Програмски језик Це - неформално



- Императивни језик (на супрот декларативном)
Наредбе се извршавају редом и у складу са значењем мењају **стање** програма.
 - Стање
 - Променљиве и парчад меморије - објекти
 - Наредбе
 - Искизи 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



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**
 - минимум 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



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

`intN_t` и `uintN_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_leastN_t` и `uint_leastN_t`, за `N` 8, 16, 32 и 64 постоје увек.

- најбржи бар толике величине

`int_fastN_t` и `uint_fastN_t`, за `N` 8, 16, 32 и 64 постоје увек.

- довољно велик да у себи држи показивач, тј. адресу

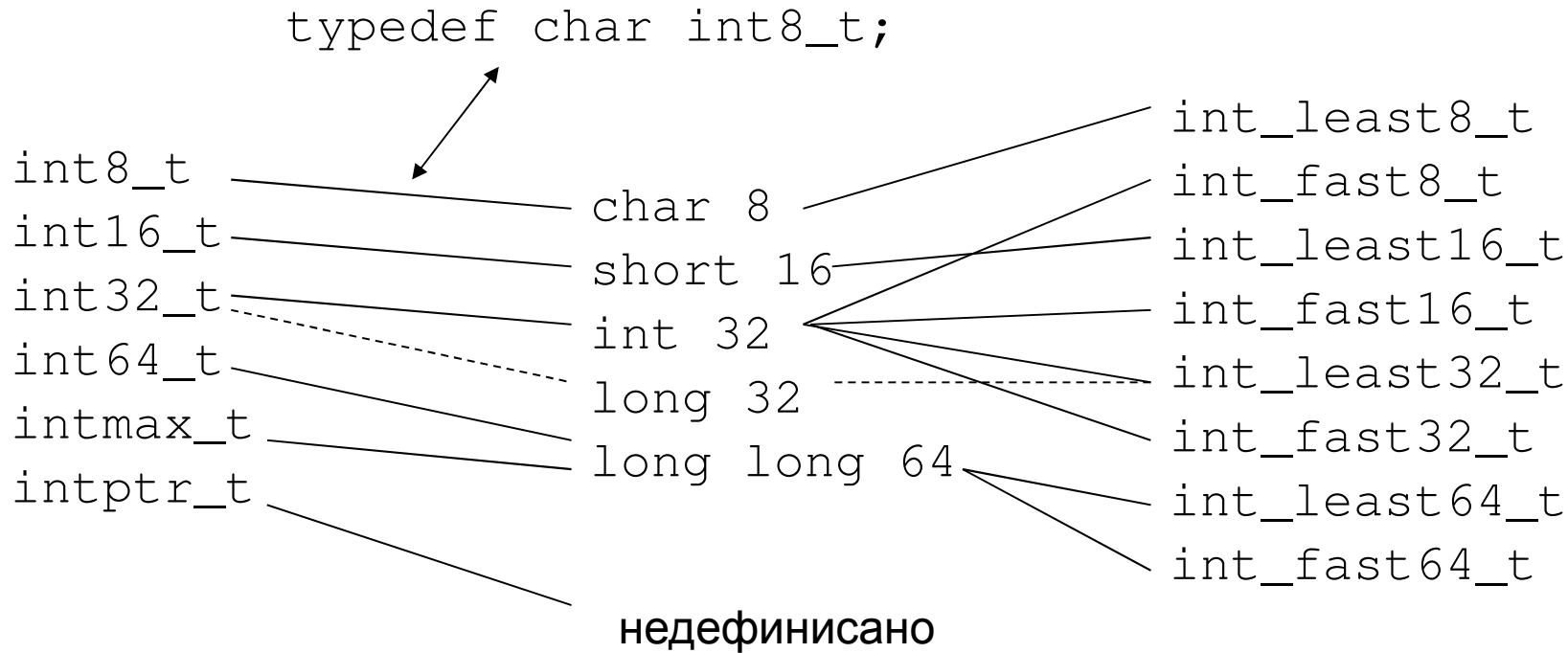
`intptr_t` и `uintptr_t`, али нису обавезни. (Зашто?)

- највећи интеџер тип

`intmax_t` и `uintmax_t`

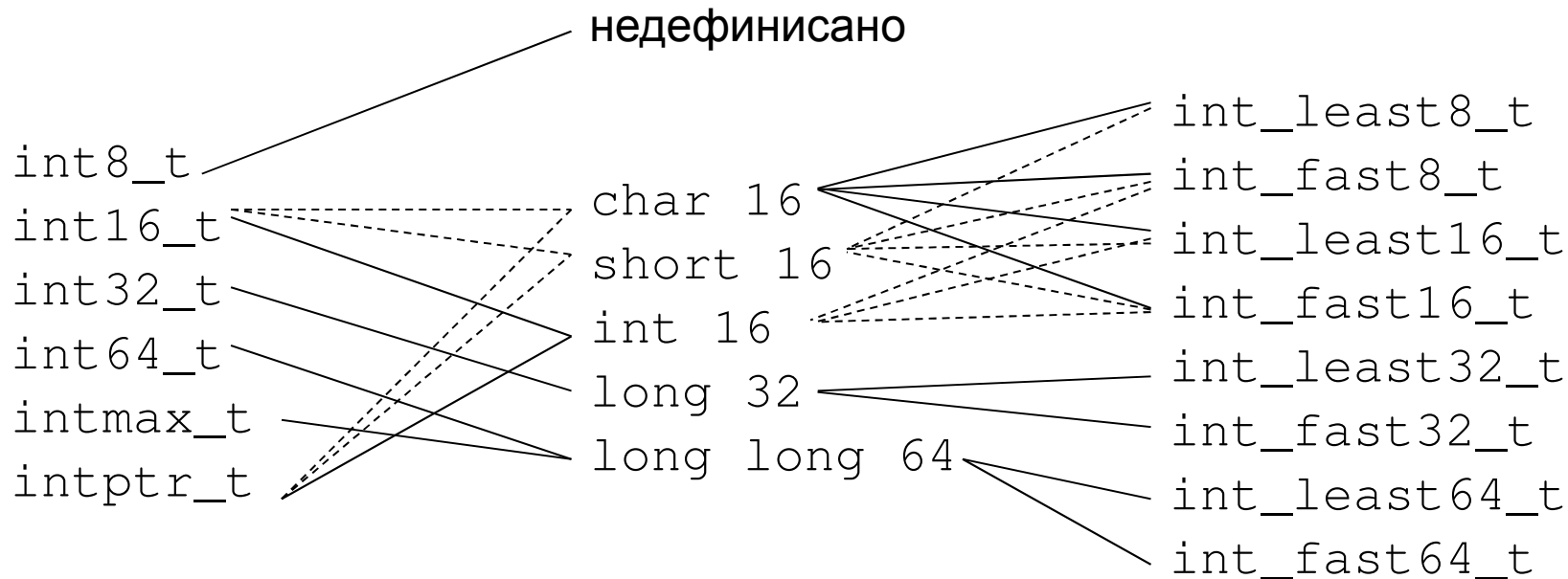


Ширина 8 бита Регистар 32 бита Адресни простор 70 бита



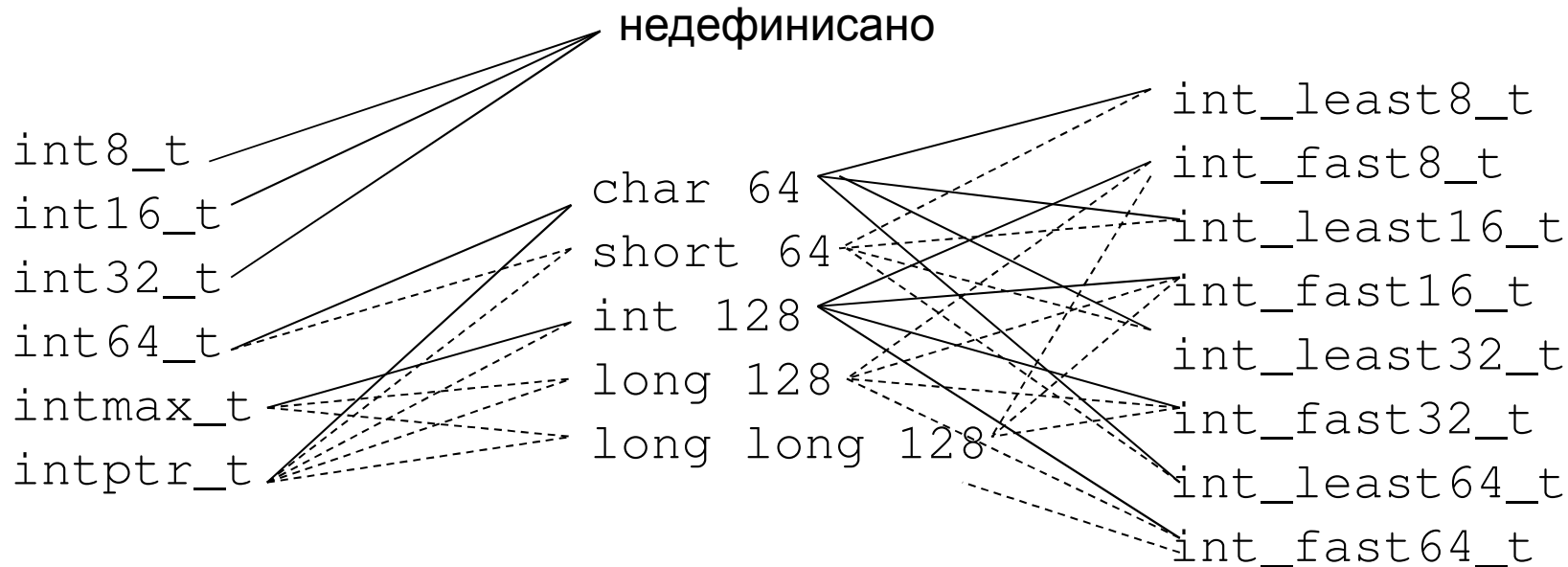


Ширина 16 бита Регистар 16 бита Адресни простор 16 бита





Ширина 64 бита Регистар 128 бита Адресни простор 16 бита





```
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
```



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



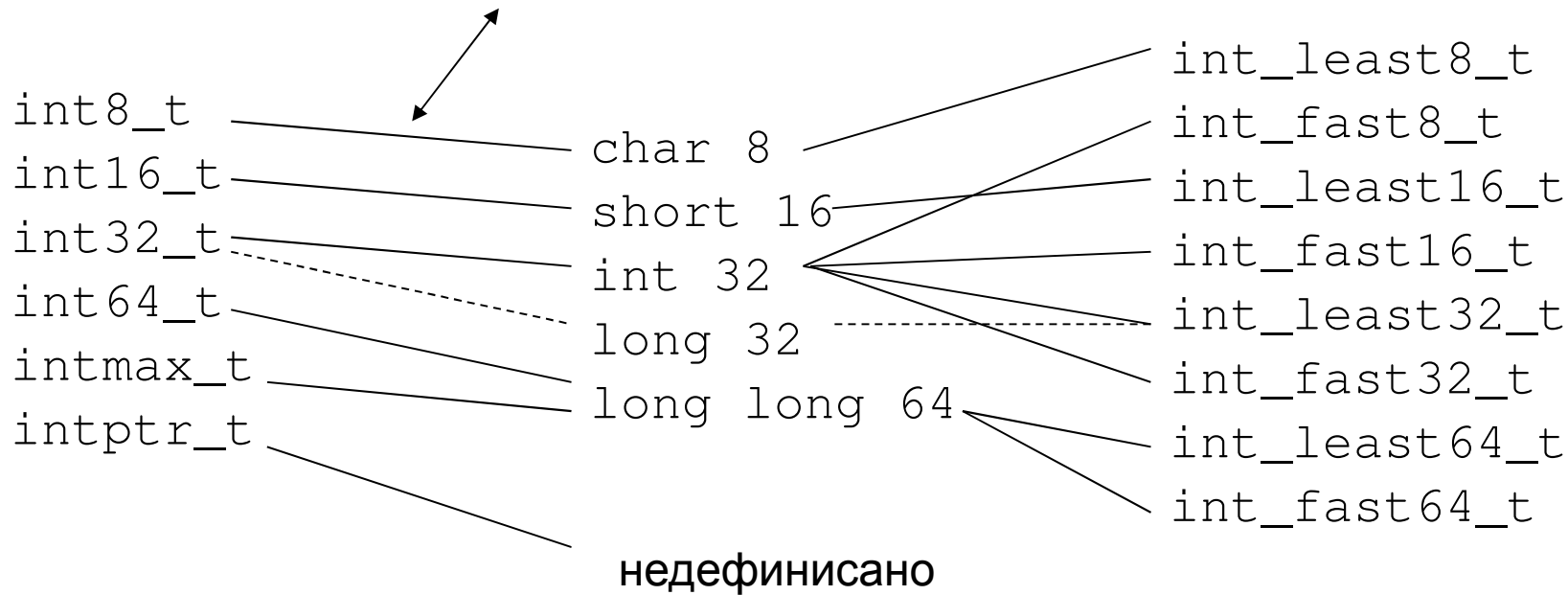
- Основно решење је прерадити код
- Неки компајлери, као што је GCC нуде опцију којом се одређује која ће бити подразумевана означеност char типа:
-fsigned-char и -funsigned-char.



Типови у `stdint.h` воде рачуна о означености `char` типа



```
typedef signed char int8_t;
```





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;
```

Нул (NULL) показивач:

```
#include <stddef.h>
```

```
p = NULL;
```

✗ ***p = y;**

Показивачи на функције:

```
int* (*p)(int a, float b);
```

```
int* foo(int x, float y);
```

```
p = foo;
```

```
ip = p(n, m);
```

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



Набројиви типови - Енумерације



- Уведени у C89 стандарду
- Прдстављају једну вредност 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 бројем од стране компајлера



Енум пример

- Првом наведеном симболу придружује се 0, осим ако експлицитно није придружен неки други број.
- Неки пут (обично у старијим кодовима) користи се `#defines` да би се направио скуп симбола

```
enum Days
{
    Sunday = 1,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

```
#define Sunday 1
#define Monday 2
#define Tuesday 3
#define Wednesday 4
#define Thursday 5
#define Friday 6
#define Saturday 7
```

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



Величина енум типа

- Енум тип се своди на целобројан тип
- Остављено је компајлеру да одлучи на који конкретан тип ће свести одређени енум тип, али 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.
- Неки компајлери нуде могућност прављења енума који су већи од `int` типа



Означеност енум типа

- Као величина целобројног типа тако и његова означеност је остављена компајлеру на избор
- 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!



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' // nije isto sto i "c"
3L 31 31	051	// 41	
5UL	0x2b	// 43	
7LL	0xFFFFFFFFD1	// -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
```



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;
```

- Промоције флоутова
 - Тип `float` се промовише у `double`



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;
```



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 операнда 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.



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



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; /* incompatible alignment */
```



Конверзије показивача

- Валидне конверзије:
 - Из показивача на воид (`void*`) у показивач на неки други тип, и обрнуто.
 - Конверзија симбола `NULL` у показивач. `NULL` је практично константа нула (у многим имплементацијама конвертована у показивач на воид) и то представља једину валидну конверзију из интеџера у показивач.
 - Показивач чија је вредност `NULL` у показивач на било који други тип.
 - Савет: користити симбол `NULL`, а не `0`.
- 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.