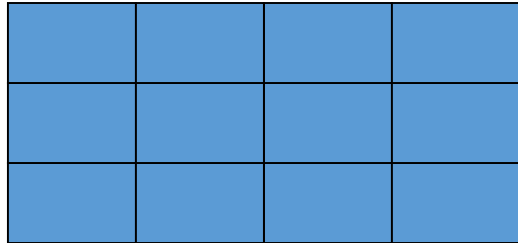


Класа матрице,
конструктор премештања,
паметни показивачи, нека
својства основних типова

Матрице

- Стандардни вектор (**vector**) и уграђени низ (тзв. „цеовски“) су једнодимензионални
- Шта ако нам требају две, или више димензија?
- Било би лепо да имамо нешто овако:

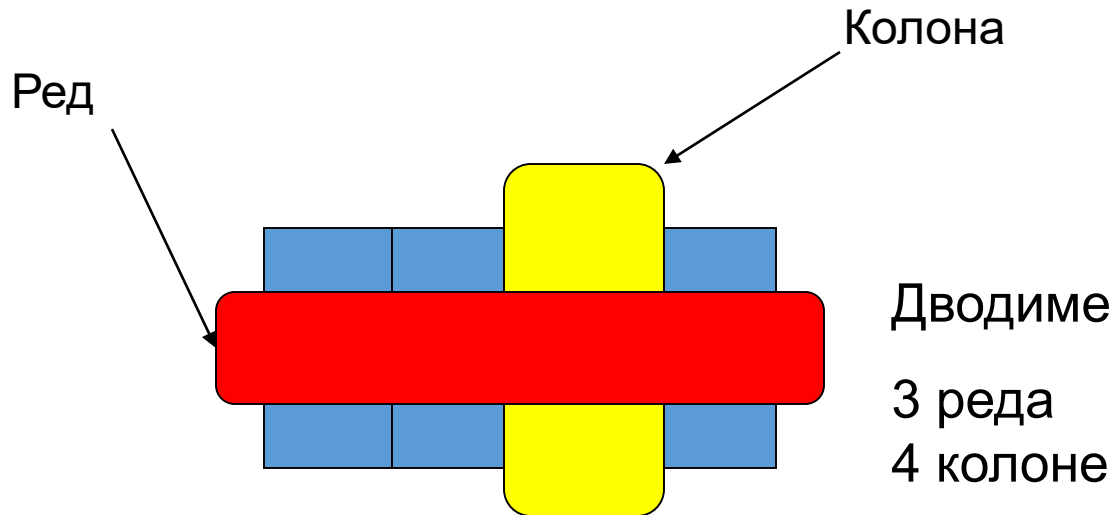


Matrix<int> m(3,4)

дводимензионална матрица, 3 пута 4

Матрице

- `Matrix<int> m(3, 4);`



Цеоовски вишедимензионални низови

- Уграђена подршка

```
int ai[4];  
double ad[3][4];  
char ac[3][4][5];  
ai[1] = 7;  
ad[2][3] = 7.2;  
ac[2][3][4] = 'c';
```

- У суштини, низ низова

Цеовски вишедимензионални низови

- Проблеми

- Фиксне величине (тј. морају бити познате током превођења)
 - Одређивање величине током извршавања захтева динамичко заузимање меморије
- Компликовано преношење параметара и повратне вредности
 - Цеовски низ се своди на показивач на први елемент
- Нема могућности провере опсега приликом приступа
 - Цеовски низ не зна своју величину
- Не постоје операције које раде над низовима
 - Нема чак ни доделе (копирања)

- Један од **главних** извора багова

Цеовски вишедимензионални низови

- Проблем преношења цеовског низа као параметра постаје још израженији код вишедимензионалних низова

```
void f1(int a[3][5]);
```

```
void f1(int a[][5]);      // исто као горње: 3 се игнорише
```

```
void f2(int[ ][5], int dim1); // прва димензија може варирати
```

```
void f3(int[ ][ ], int dim1, int dim2); // грешка!
```

```
void f4(int* m, int dim1, int dim2) // ово ради... али је мало чудно
{
    for (int i = 0; i < dim1; ++i)
        for (int j = 0; j < dim2; ++j)
            m[i*dim2+j] = 0;
}
```

Matrix библиотека

- Погледати у књизи поглавља 24.5 и 24.6
- На наредним слајдовима је представљена наша варијанта матрице, које је мало другачија (поједностављенија) од оне која је у књизи дата.

Matrix библиотека у књизи:

- Обавља провере током превођења и током извршавања
- Нуди матрице произвољних димензија
- Матрице су регуларне променљиве
 - Могу се нормално прослеђивати функцијама
- Уобичајене матричне операције
 - Индексирање: ()
 - Обраћање делу матрице: [] и .slice()
 - Додела: =
 - Скалирање: +=, -=, *=, % =, итд.
 - Сабирање матрица
 - Векторске операције (нпр., $\text{res}[i] = \text{a}[i] * \text{c} + \text{b}[i]$)
 - Скаларни производ ($\text{res} = \text{sum of } \text{a}[i] * \text{b}[i]$)
- За општи тип матрице, ефикасност је подједнака коду писаном на nižем нивоу апстракције
- Библиотеку можете проширивати по потреби (нема никакве магије)

Наша Matrix библиотека:

- Обавља провере током превођења и током извршавања
- Нуди **двострумензионалне матрице**
- Матрице су регуларне променљиве
 - Могу се нормално прослеђивати функцијама
- Уобичајене матричне операције
 - Индексирање: ()
 - Обраћање делу матрице: [] и .slice()
 - Додела: =
 - Скалирање: +=, -=, *=, % =, итд.
 - Сабирање матрица
 - Векторске операције (нпр., $\text{res}[i] = \text{a}[i] * \text{c} + \text{b}[i]$)
 - Скаларни производ ($\text{res} = \text{sum of } \text{a}[i] * \text{b}[i]$)
 - **Испис на стандардни излаз и учитавање матрице из датотеке**
- За општи тип матрице, ефикасност је подједнака коду писаном на nižем нивоу апстракције
- Библиотеку можете проширивати по потреби (нема никакве магије)

Matrix библиотека

- 2д матрицу индексирамо паром (ред, колона):

`Matrix<int> a(3,4);`

a[0]:	00	01	02	03
a[1]:	10	11	12	13
a[2]:	20	21	22	23

Diagram illustrating matrix indexing. The matrix is a 3x4 grid. The element at row 1, column 2 is labeled `a[1][2]` and `a(1,2)`.

- Елементи су поређани у меморији један иза другог, **ред по ред** (могло би бити колона по колона, али није):

00	01	02	03	10	11	12	13	20	21	22	23
----	----	----	----	----	----	----	----	----	----	----	----

Matrix библиотека

```
void init(Matrix<int>& a)
{
    for (int i = 0; i < a.d1(); ++i)
        for (int j = 0; j < a.d2(); ++j)
            a(i, j) = 10*i+j;
}
```

```
void print(const Matrix<int>& a) // да ли се штампа по редовима или по колонама?
{
    for (int i = 0; i < a.d1(); ++i) {
        for (int j = 0; j < a.d2(); ++j) {
            cout << a(i, j) << '\t';
        }
        cout << '\n';
    }
}
```

Матрица

```
Matrix<int> a(10, 20);  
a.size();           // број елемената  
a.d1();             // број елемената у реду  
a.d2();             // број елемената у колони  
int* p = a.data();  
a(i, j);            // (i, j)-ти елемент  
  
Matrix<int> a2{a}; // a2 = a  
a = a2;  
a = a + a2;
```

Имплементација класе Matrix

- Желимо да имплементирамо матрицу чији елементи могу да буду било шта.
- Ево неколико покушаја... (погледати кодове уз ово предавање)

Матрица генерички у Јави

```
public class Matrix<T> {
    protected Object m_elem[];
    protected int m_sz;
    protected int m_d1;
    protected int m_d2;

    public Matrix(int d1, int d2) {
        m_elem = new Object[d1 * d2];
        m_sz = d1 * d2;
        m_d1 = d1;
        m_d2 = d2;
    }

    public T get(int i, int j) { return (T)m_elem[i * m_d1 + j]; }
    public void set(int i, int j, T e) { m_elem[i * m_d1 + j] = e; }

    public int d1() { return m_d1; }
    public int d2() { return m_d2; }
}
```

...

Матрица генерички у Јави

...

```
public static <T> Matrix<T> add(Matrix<T> x, Matrix<T> y) {  
    Matrix<T> z = new Matrix<T>(x.d1(), x.d2());  
    for (int i = 0; i < x.m_sz; ++i) {  
        z.m_elem[i] = ??????????;  
    }  
    return z;  
}
```

Матрица генерички у Јави

...

```
public static <T> Matrix<T> add(Matrix<T> x, Matrix<T> y){  
    Matrix<T> z = new Matrix<T>(x.d1(), x.d2());  
    for (int i = 0; i < x.m_sz; ++i) {  
        z.m_elem[i] = (Integer)x.m_elem[i] + (Integer)y.m_elem[i];  
    }  
    return z;  
}
```

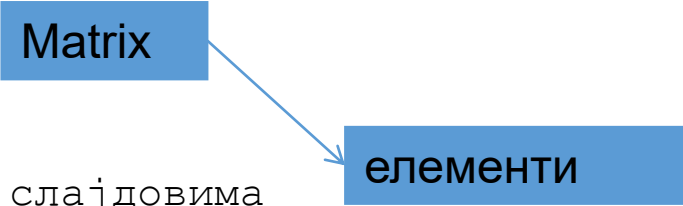

Резултати

- C++, void*: 0,044
 - Јава: 0,028
 - C++, шаблони: 0,0044
-
- ~10 пута брже од void* и ~6х брже од Јаве.

Имплементација класе Matrix

- Кроз Matrix променљиву се управља приступом елементима

```
template<class T> class Matrix {  
public:  
    // спрега, као што је описана на ранијим слајдовима  
protected:  
    T* m_elem; // елементи су поређани један за другим, као Цеоовски низ  
    const int m_sz;  
    const int m_d1;  
    const int m_d2; // број елемената по свакој димензији  
};
```

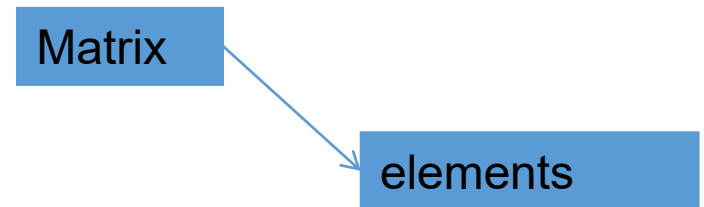


```
graph LR; Matrix[Matrix] --> elementi[елементи]
```

Имплементација класе Matrix

- Руковање ресурсима:

- Променљива типа Matrix мора руковати својим елементима
- Конструктор заузима меморију за елементе и иницијализује их
- Деструктор уништава елементе и ослобађа меморију која је за њих заузета
- Додела копира елементе



Имплементација класе Matrix

- Требају нам конструктори:

- Конструктор за подразумеване елементе

- `Matrix<T>::Matrix(Index, Index);`

- `Matrix<int> m0(2, 3); // сви елементи постављени на 0`

- Конструктор са иницијализационом листом (C++11)

- `Matrix<T>::Matrix(std::initializer_list<T>);`

- `Matrix<int> m1 = {{1, 2, 3}, {4, 5, 6}};`

- Конструктор копије

- `Matrix<T>::Matrix(const Matrix<T>&);`

- `Matrix<int> m2 = m1;`

Преношење матрице

- Посматрајмо ову функцију:

```
Matrix operator+(Matrix a, Matrix b)
{
    Matrix res;
    // Сабери матрице и резултат смести у res
    return res;
}
```

```
Matrix x, y, z;
```

```
z = x + y; // Колико пута се копирају матрице?
```

Преношење матрице

- Улазне матрице се беспотребно копирају
- Компајлер може оптимизовати код тако што ће уклонити беспотребна копирања, али се на то не можемо ослањати у општем случају.
- Решење за улазне параметре:

```
Matrix operator+(const Matrix& a, const Matrix& b)
```

- Шта са повратном вредношћу?

Преношење матрице

- Једна идеја:

- Враћамо показивач на објекат заузет помоћу **new**

```
Matrix* operator+(const Matrix& a, const Matrix& b);  
Matrix& z = *(x + y);
```

- Проблеми:
 - Ружно на месту позива.
 - Ко зове **delete**?

Преношење матрице

- Друга идеја:

- Враћамо референцу на објект заузет помоћу **new**

```
Matrix& operator+(const Matrix& a, const Matrix& b);  
Matrix& z = x + y;
```

- Проблеми:

- ~~Ручно на месту позива.~~

- Ко зове **delete**?

- Који **delete**? Где је овде показивач?

Преношење матрице

- Трећа идеја:
 - Прослеђујемо референцу на већ заузет објекат у који треба да се смести резултат

```
void operator+(const Matrix& a, const Matrix& b, Matrix& res);  
Matrix res = x + y;  
void plus(const Matrix& a, const Matrix& b, Matrix& res);  
plus(x, y, res);
```

- Проблеми:
 - Ружно на месту позива.
 - А и оператор сабирања прима само два параметра

~~• Ко зове delete?~~

Преношење матрице - потпуно решење

- Требају нам конструктори:

- Конструктор за подразумеване елементе

- `Matrix<T>::Matrix(Index, Index);`

- `Matrix<int> m0(2,3);` // сви елементи постављени на 0

- Конструктор са иницијализационом листом

- `Matrix<T>::Matrix(std::initializer_list<T>);`

- `Matrix<int> m1 = {{1,2,3}, {4,5,6}};`

- Конструктор копије

- `Matrix<T>::Matrix(const Matrix<T>&);`

- `Matrix<int> m2 = m1;`

- Мув (move) конструктор (Конструктор премештања)

- `Matrix<T>::Matrix(Matrix<T>&&);`

- `return m1;`



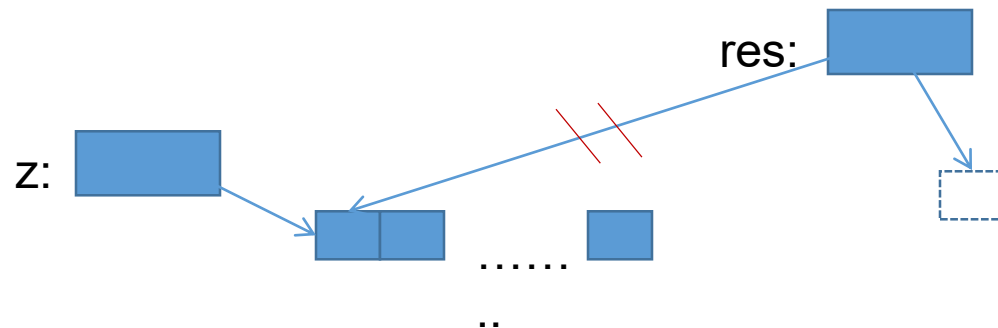
два &

Мув семантика

- Функција враћа променљиву типа **Matrix**

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
    Matrix res;
    // Сабери матрице и резултат смести у res
    return res;
}
Matrix z = x + y;
```

- Али се позива мув конструктор (конструктор премештања)
 - Нема копирања: само се „преузима репрезентација“

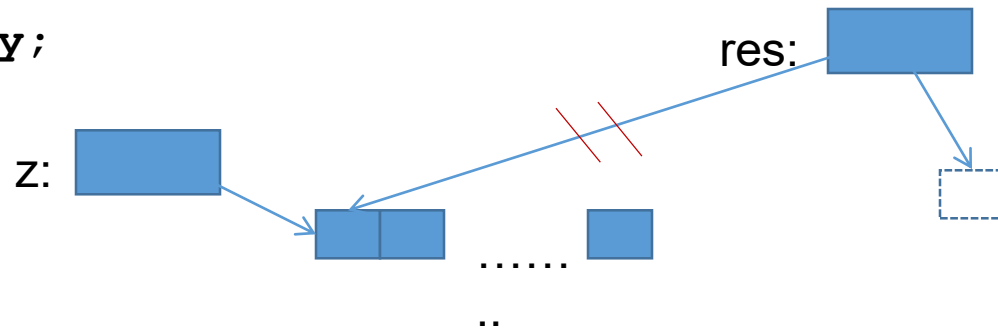


Мув семантика

- Мув конструктор

```
class Matrix {  
    // ...  
    Matrix(Matrix&& a)  
        : m_elem(a.m_elem),  
          m_sz(a.m_sz),  
          m_d1(a.m_d1),  
          m_d2(a.m_d2)  
    {  
        a.m_elem = nullptr;  
    }  
};
```


`Matrix z = x + y;`



Мув семантика

- А може и мув оператор доделе

```
class Matrix {  
    // ...  
    Matrix& operator=(Matrix&& a)  
    {  
        delete[] m_elem;  
        m_elem = a.m_elem;  
        a.m_elem = nullptr;  
        m_sz = a.m_sz;  
        m_d1 = a.m_d1;  
        m_d2 = a.m_d2;  
  
        return *this;  
    }  
};  
  
Matrix z;  
z = x + y;
```



Мув семантика

- Мув конструктор или мув додела се имплицитно позивају у следећа два случаја (суштински, када компајлер неоспорно зна да се животни век десног операнда завршава):

```
Matrix operator+(const Matrix& a, const Matrix& b) {  
    Matrix res;  
    // ... //  
    return res; //<- Овде.  
}
```

```
Matrix z = x + y; //<- Овде. Десни операнд је привремени  
Matrix z{x + y}; // објекат, резултат сабирања, који живи  
Matrix z;        // само до краја наредбе  
z = x + y;
```

Класе - подсећање

- Кључне операције:
 - Подразумевани конструктор (своди се на празан код)
 - Поништава се ако се декларише било који други конструктор
 - Конструктор копије (подразумевано се своди на копирање података)
 - Додела копије (подразумевано се своди на копирање података)
 - Деструктор (подразумевано се своди на празан код)
- Правило тројке: „Ако вам не одговара подразумевана верзија бар једне од ове три функције, онда вам највероватније не одговара подразумевана верзија ни једне од њих.“
- То јест: „Најчешће ћете дефинисати или све три функције, или ниједну“

Класе

- Кључне операције:
 - Подразумевани конструктор (своди се на празан код)
 - Поништава се ако се декларише било који други конструктор
 - Конструктор копије (подразумевано се своди на копирање података)
 - Додела копије (подразумевано се своди на копирање података)
 - Деструктор (подразумевано се своди на празан код)
 - Конструктор премештања (мув конструктор)
 - Додела премештањем (мув додела)

Ако мув верзија нема, онда ће бити позване верзије са копирањем.
- Правило петице: „Ако вам не одговара подразумевана верзија бар једне од ових пет функција, онда вам највероватније не одговара подразумевана верзија ни једне од њих.“
- То јест: „Најчешће ћете дефинисати или свих пет функција, или ниједну“ 32

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара

```
int foo(const int* x) {  
    return *x + 1;  
}
```

```
int foo(const int& x) {  
    return x + 1;  
}
```

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара
 - Униформан приступ атрибуту класе

```
struct vector {  
    int* at(int x) {  
        return &elem[x];  
    }  
}
```

```
*v.at(i) = 5;  
cout << *v.at(i);
```

```
struct vector {  
    int& at(int x) {  
        return elem[x];  
    }  
}
```

```
v.at(i) = 5;  
cout << v.at(i);
```

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара
 - Униформан приступ атрибуту класе
 - За краћи назив променљиве

```
if (x->clan1->clan2->clan3.get() == 5)
{
    x->clan1->clan2->clan3.set(8);
}
```

```
clan3Type* y = &(x->clan1->clan2->clan3);
if (y->get() == 5)
{
    y->set(8);
}
```

```
clan3Type& y = x->clan1->clan2->clan3;
if (y.get() == 5)
{
    y.set(8);
}
```

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
 - Пренос параметара
 - Униформан приступ атрибуту класе
 - За краћи назив променљиве
 - За везе између објеката које увек морају постојати

```
struct zavisnaKlasa{  
    refKlasa* ptr; // мора увек показивати  
                  // не неку инстанцу refKlasa типа  
    zavisnaKlasa() : ptr(&globRefKlasa) {}  
    zavisnaKlasa(int count) {} // грешка неће бити откривена  
};
```

```
struct zavisnaKlasa{  
    refKlasa& ptr; // мора увек показивати  
                  // не неку инстанцу refKlasa типа  
    zavisnaKlasa() : ptr(globRefKlasa) {}  
    zavisnaKlasa(int count) {} // грешка у превођењу  
};
```

Показивачи

- Још неки случајеви:

- Када динамички заузимамо објекат зато што желимо да му животног век траје дуже од досега у којем је направљен. Референца овде не помаже.

```
мојТип* makeМојТип() {  
    return new мојТип(1, 2, 3);  
}
```

```
void foo() {  
    //...  
    мојТип* p = makeМојТип();  
    // p је сада "власник" објекта  
    //...  
    // је ли било delete?  
}
```

Паметни показивачи

- Још неки случајеви:

- Када динамички заузимамо објекат зато што желимо да му животног век траје дуже од досега у којем је направљен. Референца овде не помаже.

```
std::unique_ptr<mojTip> makeMojTip() {  
    return std::unique_ptr<mojTip>(new mojTip(1, 2, 3));  
}
```

```
void foo() {  
    //...  
    std::unique_ptr<mojTip> p = makeMojTip();  
    // p је сада "власник" објекта  
    //...  
    // не треба delete... јер ће бити позвано у деструктору p-а  
}
```

Паметни показивачи

- Још неки случајеви:

- Када динамички заузимамо објекат зато што желимо да му животног век траје дуже од досега у којем је направљен. Референца овде не помаже.

```
std::unique_ptr<mojTip> makeMojTip() {  
    return std::make_unique<mojTip>(1, 2, 3);  
}
```

```
void foo() {  
    //...  
    std::unique_ptr<mojTip> p = makeMojTip();  
    // p је сада "власник" објекта  
    //...  
    // не треба delete... јер ће бити позвано у деструктору p-а  
}
```

Паментни показивачи

- `Y <memory>`
 - `unique_ptr`

```
struct mojTip {  
    mojTip(int a, int b, int c) : x(a), y(b), z(c) {}  
    int x, y, z;  
};
```

```
void bar(mojTip& x);
```

```
void foo() {  
    //...  
    std::unique_ptr<mojTip> p = std::make_unique<mojTip>(1, 2, 3);  
    // auto p = std::make_unique<mojTip>(1, 2, 3);  
    // p је сада "власник" објекта  
    //...  
    cout << p->x;  
    bar(*p);  
    p++; // грешка!  
    p[5]; // грешка!  
}
```


Паметни показивачи

- `Y <memory>`
 - `unique_ptr`

```
struct mojTip {
    mojTip(int a, int b, int c) : x(a), y(b), z(c) {}
    int x, y, z;
};

void zol(mojTip* x);

void foo() {
    //...
    std::unique_ptr<mojTip> p = std::make_unique<mojTip>(1, 2, 3);
    // p је сада "власник" објекта
    //...
    zol(p.get()); // али zol не сме звати delete!
    std::unique_ptr<mojTip> q{p}; // грешка! може бити само један
    q = p; // грешка!
    q = std::move(p);
    mojTip* r = q.release(); // p више није власник објекта
}
```

Паметни показивачи

- Основни принцип RAll: „власништво“ над ресурсом (објектом који нема досег) доделити некој променљивој која има досег.

Прецизност итд.

- Бројеви у покретном зарезу су само апроксимација реалних бројева

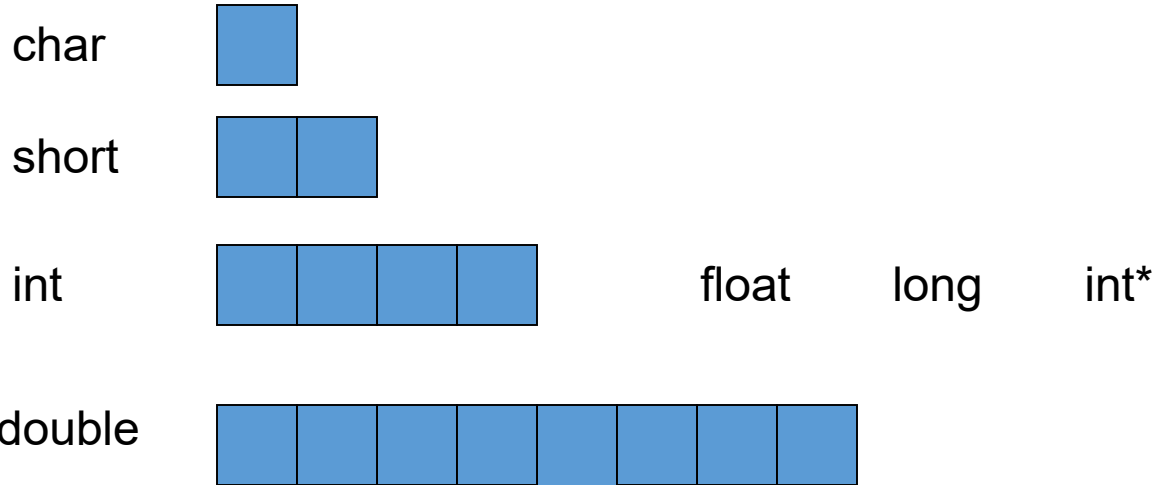
```
float x = 1.0/333;  
float sum = 0;  
for (int i = 0; i < 333; ++i)  
    sum += x;  
cout << sum << "\n";    // 0.999999
```

- Целобројни типови представљају само (релативно) мале целе бројеве

```
short y = 40000;  
int i = 1000000;  
cout << y << " " << i*i << "\n";    // -25536 -727379968  
// али не морају бити ови резултати
```

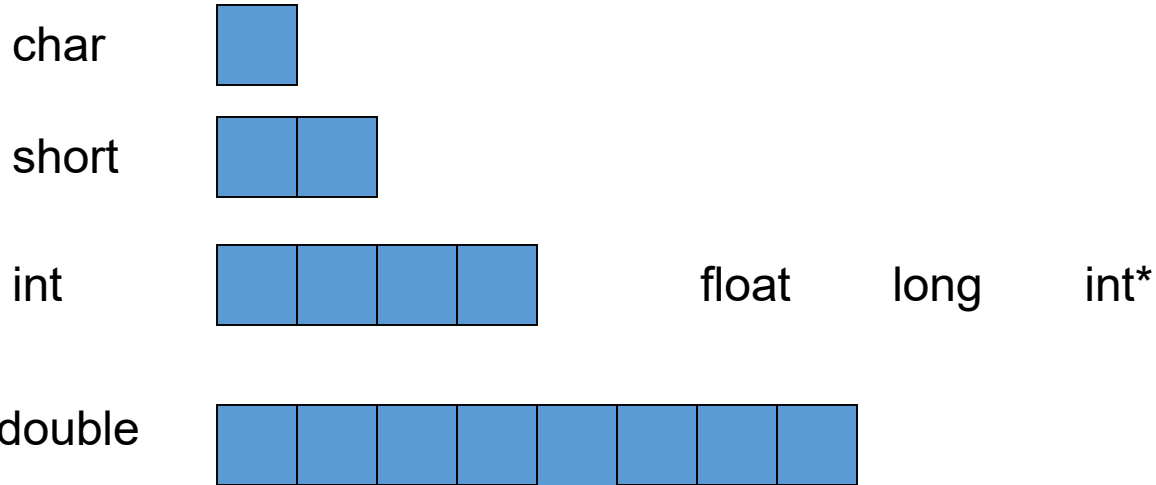
- Једноставно, не постоји ефикасан начин да у рачунару представимо сваки могући број

Величине



- У Це++ величине уграђених типова зависе од хардвера и компајлера
 - Горње величине су за уобичајени персонални рачунар, за уобичајене оперативне системе и уобичајене компајлере на њима.
 - **sizeof(x)** даје величину типа **x** изражену у бајтовима
 - По дефиницији, **sizeof(char)==1**
 - По дефиницији, 1 бајт = ? бита

Величине



- У Це++ величине уграђених типова зависе од хардвера и компајлера
 - Горње величине су за уобичајени персонални рачунар, за уобичајене оперативне системе и уобичајене компајлере на њима.
 - **sizeof(x)** даје величину типа **x** изражену у бајтовима
 - По дефиницији, **sizeof(char)==1**
 - По дефиницији, ~~1 бајт = ? бита~~
 - По дефиницији, 1 бајт одговара ширини меморије, колико год она бита била (а минимално је 8), тј. величини најмање адресиве јединице 45

Прекорачење и одсецање

- Током рачунања може доћи до прекорачења или одсецања
- Це++ неће ухватити такве проблеме уместо вас

```
void f(char c, short s, int i, long lg, float fps, double fpd)
{
    c = i;           // char је у суштини мали интеџер, али ово ипак није
    препоручљиво
    s = i;
    i = i+1; // шта ако је у i највећи број представљив у int типу?
    lg = i*i; // long не мора бити већи тип од int
               // осим тога, i*i је int типа, без обзира ког типа је lg
    fps = fpd;
    i = fpd; // одсецање: нпр. 5.7 -> 5
    fps = i; // за велике целе бројеве може се изгубити прецизност

    char ch = 0;    // пробајте ово
    for (int i = 0; i < 500; ++i)
        { cout << int(ch) << "\t"; ++ch; }}
}
```

Прекорачење и одсецање

- Једноставан начин провере
 - Додела и накнадно поређење

```
void f(int i)
{
    char c = i;
    if (c != i) {
        // грешка! изгубили смо део информација
    }
    // ...
}
```

- У **std_lib_facilities.h** постоји **narrow_cast**, и он у суштини обавља ову проверу

Нумеричка ограничења типова

- Свака имплементација Це++ компајлера специфицира особине уграђених типова
- **<limits>**
 - за сваки тип постоји
 - `min()` `// numeric_limits<int>::min()`
 - `max()`
 - ...
 - за типове бројева у покретном зарезу
 - Нпр. `numeric_limits<float>::max_exponent10`
 - Плус још гомила особина – ако вам буду требале, потражићете их
- **<climits>** и **<cfloat>**
 - Цеовски начин саопштавања ових информација
 - `INT_MAX` `// највећа int вредност`
 - `DBL_MIN` `// најмања позитивна double вредност`

Нумеричка ограничења типова

- Ова ограничења су важна за програмирање на nižем нивоу, али и у случајевима када наш код обавља мало озбиљније нумеричке прорачуне
- У осталим случајевима су врло ретко од интереса и већином их можете игнорисати (бар у почетним фазама развоја програма, а некада и до самог краја)
- „Правило десне руке“: **bool** за логичке вредности, **char** за знакове, **int** за целе бројеве, **double** за реалне бројеве.

Грешке код математичких функција

- Ако нека математичка функција наиђе на проблем она поставља **errno** променљиву декларисану у **<cerrno>**

```
void f(double negative, double very_large)
{
    errno = 0;
    sqrt(negative);
    if (errno != 0) { /* ... */ } // errno != 0 значи „јавила се нека грешка“
    if (errno == EDOM)
        cerr << "sqrt() not defined for negative argument\n";

    pow(very_large, 2);
    if (errno == ERANGE)
        cerr << "pow(" << very_large << " ,2) too large for a double\n";
}
```

Комплексни бројеви

- У стандардној библиотеци **<complex>**

```
template<class T> class complex {
    T re, im;
public:
    complex(const T& r, const T& i) : re(r), im(i) { }
    complex(const T& r) : re(r), im(T()) { }
    complex() : re(T()), im(T()) { }
    // или: complex(const T& r = T(), const T& i = T()) : re(r), im(i) { }

    T real() { return re; }
    T imag() { return im; }

    // оператори:  =  +=  -=  *=  /=
};

// operators:  +  -  /  *  ==  !=
// стандардне математичке функције раде и са комплексним бројевима:
// pow(), abs(), sqrt(), cos(), log(), итд. као и norm() (abs() на
    квадрат)
```

Комплексни бројеви

// complex<T> можемо третирати као уграђени тип.
// Обратите пажњу да неке операције нису дефинисане за
// комплексне бројеве (нпр. <).

```
typedef complex<double> dcmplx;  
  
void f(dcmplx z, vector<complex<double>>& vc)  
{  
    dcmplx z2 = pow(z, 2);  
    dcmplx z3 = z2 * 9 + vc[3];  
    dcmplx sum = accumulate(vc.begin(), vc.end(), dcmplx());  
}
```