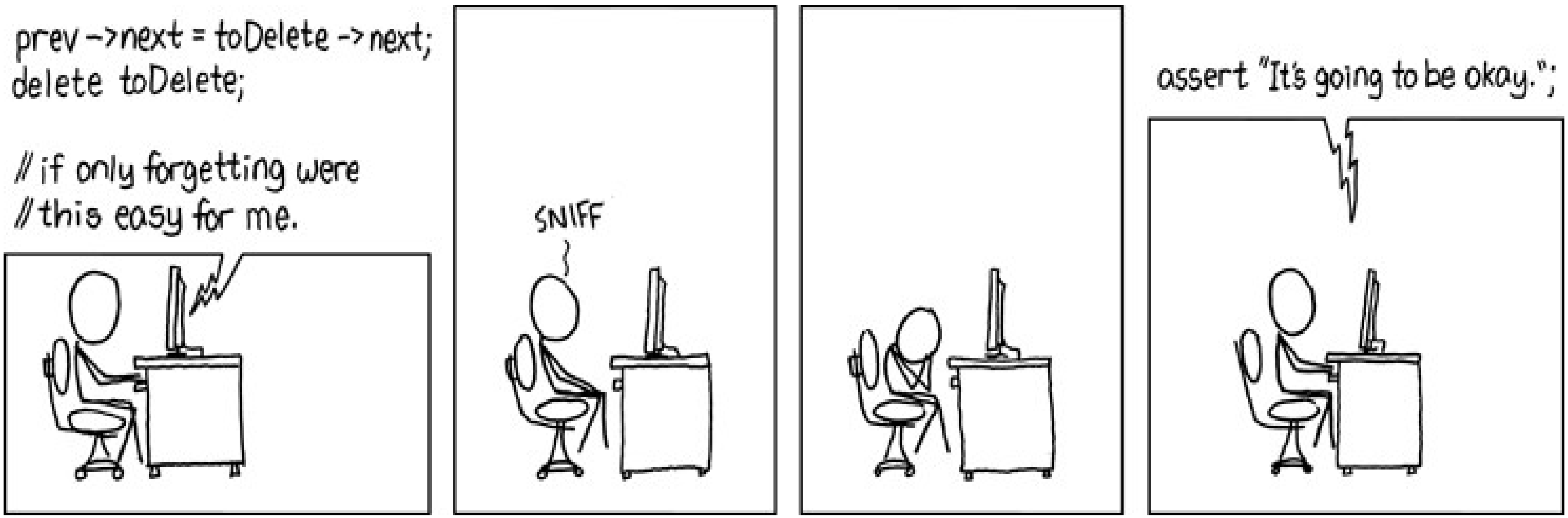
C++ Pointers (Part II), Arrays &References



(https://xkcd.com/379/)

Krishna Kumar

Pointers (recap)

- Pointer values are memory addresses
- A pointer **does not** know the **number of elements** that it's pointing to.
- A pointer ***does*** know the **type of the object** that it's pointing to
- **Stack:** Stores local data, return addresses, used for parameter passing
  - Local variables
  - Cleared when out of scope
- **Heap:** You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data
  - Accessed using *new* and *delete*
  - *Memory leaks – manual delete of used memory*

Void*

- **void*** means "pointer to some memory that the compiler doesn't know the type of"
- We use **void*** when we want to transmit an address between pieces of code that really don't know each other's types – so the programmer has to know
- The primary use for void* is for passing pointers to functions that are not allowed to make assumptions about the type of the object and for returning untyped objects from functions
- There are no objects of type **void**
  - **void v;**   *// error*
  - **void f();** *// f() returns nothing – f() does **not** return an object of type **void***
- Any pointer to object can be assigned to a **void***
  - **int* pi = new int;**
  - **double* pd = new double[10];**
  - **void* pv1 = pi;**
  - **void* pv2 = pd;**

void* (cont...)

- To use a **void*** we must tell the compiler what it points to

```
void f(void* pv) {
    void* pv2 = pv;       // copying is ok (copying is what void*s are for)
    double* pd = pv;      // error: can’t implicitly convert void* to double*
    *pv = 7;              // error: you can’t dereference a void*
    ++pv;                 // error: can't increment void*
                          // (the size of the object pointed to is unknown)
    pv[2] = 9;   // error: you can’t subscript a void*
    int* pi = static_cast<int*>(pv);      // ok: explicit conversion
                                          // Unsafe!!

    // …
}
```

- A **static_cast** can be used to explicitly convert to a pointer to object type
  - "**static_cast**" is a deliberately ugly name for an ugly (and dangerous) operation – use it only when absolutely necessary

Warnings

- There are some serious *gotchas* when using casts on pointers, so try to avoid them
  - There is precisely ***one*** safe use, to get back to the type you started with – and a compiler can't check you got it right

    ```
    myclass object();

    myclass* myptr = object.addr();

    void* rawptr = myptr;

    . . .

    myclass* newptr = static_cast<myclass*>(rawptr);
    ```

- You will often see C-style casts in C++ code derived from C – or by C programmers writing "C++"
  - They look like **`(newtype)expression`**
- But ***don't*** write them
  - They are extremely hard to spot in non-trivial code
  - They are even ***less*** safe than **`reinterpret_cast`**

void*

- **void*** is the closest C++ has to a plain machine address
- Functions using void* pointers typically exist at the very lowest level of the system, where real hardware resources are manipulated.

  void* my_alloc(size_t n); // allocate n bytes
  //from my special heap
- **Avoid using it if you possibly can**
  - It bypasses essentially all type checks
  - Occurrences of void*s at higher levels of the system should be viewed with great suspicion because they are likely indicators of design errors.
- Pointers to functions and pointers to members cannot be assigned to void*s.

nullptr

- A pointer that does not point to an object.
- It can be assigned to any pointer type, but not to other built-in types:
  - **int* pi = nullptr;**
  - **double* pd = nullptr;**
  - **int i = nullptr;** // error: i is not a pointer
  - **int* x = 0;** // x gets the value nullptr
  - **int* p = NULL;** // error: can't assign a void* to an int*

    // Macro NULL is typically (void*)0,

    // which makes it illegal in C++

Arrays

- For a type T, T[size] is the type "array of size elements of type T." indexed from 0 to size–1.

```
float v[3];     // an array of three floats: v[0], v[1], v[2]
char* a[32];    // an array of 32 pointers to char: a[0] .. a[31]
```

- Arrays don't have to be on the free store

```
char ac[7];                  // global array – "lives" forever –
                             // "in static  storage" –AVOID!
int max = 100;
int ai[max];
int f(int n) {
    char lc[20];  // local array – "lives" until the end of
                  // scope – on  stack
    int li[60];
    double lx[n]; // error: a local array size must be known
                  // at compile  time
    std::vector<double> lx(n); would work
// …
}
```

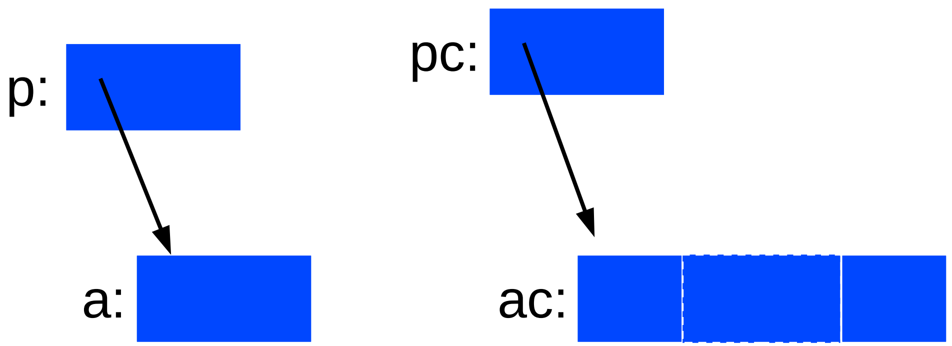
Initialization syntax

```
char ac[ ] = "Hello, world";          // array of 13 chars, not 12 (the compiler
                                      // counts them and then adds a null
                                      // character at the end
char* pc = "Howdy";                   // pc points to an array of 6 chars
char* pp = {'H', 'o', 'w', 'd', 'y', 0 };    // another way of saying the same

int ai[ ] = { 1, 2, 3, 4, 5, 6 };     // array of 6 ints
                                      // not 7 – the “add a null character at the end”
                                      // rule is for literal character strings only
int ai2[100] = { 0,1,2,3,4,5,6,7,8,9 };    // the last 90 elements are initialized to 0
double ad3[100] = { };                // all elements initialized to 0.0
```

Address of: &

- You can get a pointer to any object
  - not just to objects on the free store

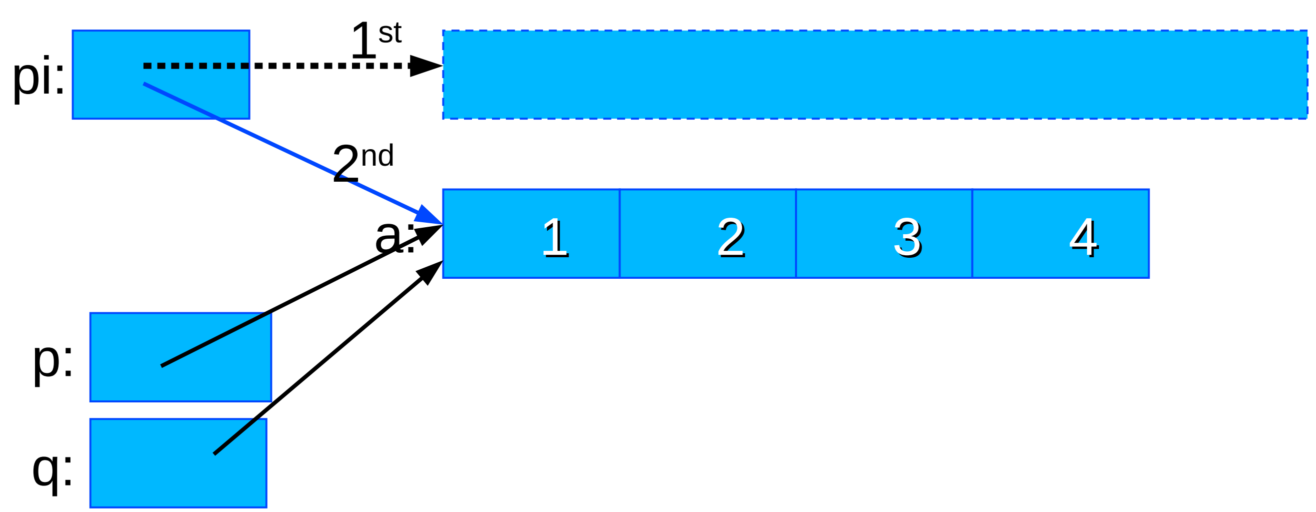



```
int a;
char ac[20];

void f(int n)
{
    int b;
    int* p = &b;        // pointer to individual variable
    p = &a;
    char* pc = ac;      // the name of an array names a pointer to its first element
    pc = &ac[0];        // equivalent to pc = ac
    pc = &ac[n];        // pointer to ac's nth element (starting at 0th)
                        // warning: range is not checked
    // …
}
```

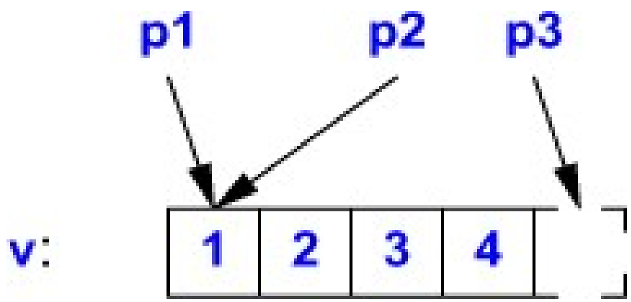
Arrays (often) convert to pointers

```
void f(int pi[ ])  // equivalent to void f(int* pi)
 {
    int a[ ] = { 1, 2, 3, 4 };
    int b[ ] = a;   // error: copy isn't defined for arrays
    b = pi;         // error: copy isn't defined for arrays. Think of a
                    // (non-argument) array name as an immutable pointer
    pi = a;         // ok: but it doesn't copy: pi now points to a's first element
                    // Is this a memory leak? (maybe)
    int* p = a;     // p points to the first element of a
    int* q = pi;    // q points to the first element of a
}
```



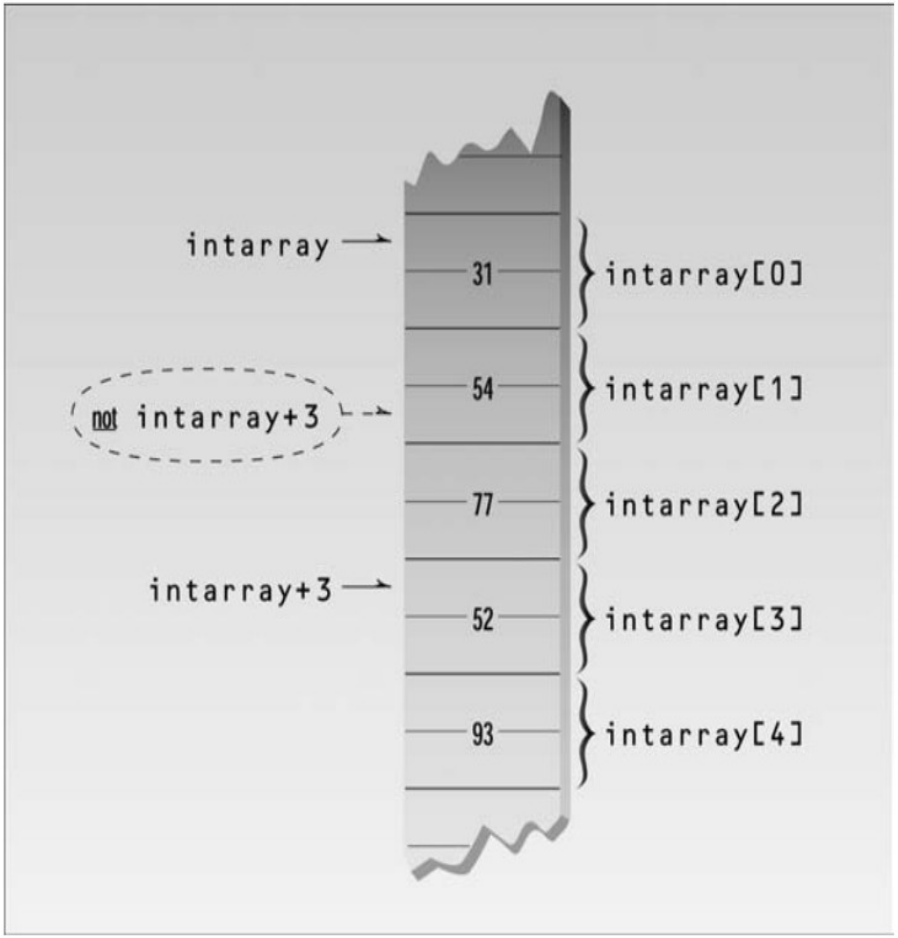
Arrays into pointers

int v[] = { 1, 2, 3, 4 };
int* p1 = v; // pointer to initial element (implicit
// conversion)
int* p2 = &v[0]; // pointer to initial element
int* p3 = v+4; // pointer to one-beyond-last element



int* p4 = v–1; // before the beginning, undefined: don't do it
int* p5 = v+7; // beyond the end, undefined: don't do it

Accessing Array Elements




```
void fi(char v[]) {
    for (int i = 0; v[i]!=0; ++i)
        use(v[i]);
}

void fp(char v[]) {  //Using Pointers
    for (char* p = v; *p!=0; ++p)
        use(*p);
}
```

Subscripting a built-in array a[j]:

a[j] == *(&a[0]+j) == *(a+j) == *(j+a) == j[a]

3["Texas"]=="Texas"[3]=='a' (But, don't do it!)

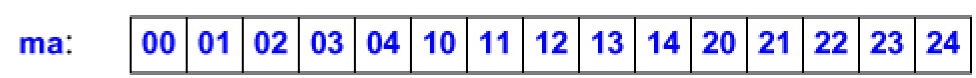
Multidimensional Arrays

- are represented as arrays of arrays.
- a 3-by-5 array is declared like this:

  int ma[3][5];   // 3 arrays with 5 ints each

- We can initialize ma like this:

```
void init_ma() {
    for (int i = 0; i!=3; i++)
        for (int j = 0; j!=5; j++)
            ma[i][j] = 10*i+j;
}
```



The array ma is simply 15 ints that we access as if it were 3 arrays of 5 ints. In particular, there is no single object in memory that is the matrix ma – only the elements are stored. The dimensions 3 and 5 exist in the compiler source only.

Be careful with arrays and pointers

```
char* f() {
        char ch[20];
        char* p = &ch[90];
        // …
        *p = 'a';           // we don't know what this'll overwrite
        char* q;            // forgot to initialize
        *q = 'b';           // we don't know what this'll overwrite
        return &ch[10];           // oops: ch disappear upon return from f()
                                  // (an infamous "dangling pointer")
}

void g() {
        char* pp = f();
        *pp = 'c';          // we don't know what this'll overwrite
        // (f's ch are gone for good after the return from f)
}
```

Passing Arrays

- Arrays **cannot** directly be passed by value

```
void comp(double arg[10]) {        // arg is a double*
    for (int i=0; i!=10; ++i)
        arg[i]+=99;
}
```

// writes to arg[i] are writes directly to the elements of comp()'s argument, rather than to a copy

```
void f() {
    double a1[10];
    double a2[5];
    double a3[100];

    comp(a1);
    comp(a2);    // disaster!
    comp(a3);    // uses only the first 10 elements
};
```

Arrays don't know their own size

```
void f(int pi[ ], int n, char pc[ ]) {
        // equivalent to void f(int* pi, int n, char* pc)
        // warning: very dangerous code, for illustration only,
        // never "hope" that sizes will always be correct
    char buf1[200];
    strcpy(buf1,pc);            // copy characters from pc into buf1
                                // strcpy terminates when a '\0' character is found
                                // hope that pc holds less than 200 characters
    strncpy(buf1,pc,200);       // copy 200 characters from pc to buf1
                                // padded if necessary, but final '\0' not guaranteed
    int buf2[300];  // you can't say char buf2[n]; n is a variable
    if (300 < n) error("not enough space");
    for (int i=0; i<n; ++i) buf2[i] = pi[i];    // hope that pi really has space for
                                                // n ints; it might have less
}
```

Pointers and Consts

- constexpr: Evaluate at compile time
- const: Do not modify in this scope

```
const int model = 90;            // model is a const
const int v[] = { 1, 2, 3, 4 };  // v[i] is a const
const int x;                     // error: no initializer
// Declaring something const ensures that its value will not change
within its scope:
void f() {
    model = 200;    // error
    v[2] = 3;       // error
}
```

Pointers and const

```
void f1(char* p) {
    char s[] = "Gorm";

    const char* pc = s;          // pointer to constant
    pc[3] = 'g';                 // error: pc points to constant
    pc = p;                      // OK

    char* const cp = s;          // constant pointer
    cp[3] = 'a';                 // OK
    cp = p;                      // error: cp is constant

    const char *const cpc = s;   // const pointer to const
    cpc[3] = 'a';                // error: cpc points to constant
    cpc = p;                     // error: cpc is constant
}
```

Pointers and references

- Think of a reference as an automatically dereferenced pointer
  - Or as "an alternative name for an object"
  - A reference must be initialized
  - The value of a reference cannot be changed after initialization

```
int x = 7;
int y = 8;
int* p = &x;
*p = 9;
p = &y;  // ok
int& r = x;
x = 10;
r  = 11;
r = &y;   // error (and so is all other attempts to change what r refers to)
```

Vectors and Arrays (C++11)

Vectors are sequence containers, that are dynamic and they know their size

- vector<int> vi; // create a zero-size vector of int

  array<int, 5> ai;

- // create array object of 5 ints

  array<double, 4> ad = {1.2, 2.1, 3.43. 4.3}

Iterating through vectors

```
// Normal way of iteration
for(int i=0; i < vector.size(); i++){
   std::cout << vector[i] << std::endl;
}

// Safe way to iterate – will never go over the size! Exception handling
for(int i=0; i < vector.size(); i++){
   std::cout << vector.at(i) << std::endl;
}
```

Iterating through vectors

```
// Normal
for(std::vector<T>::size_type i = 0; i != v.size(); i++)
    v[i].doSomething();

//  Iterators
for(std::vector<T>::iterator it = v.begin(); it != v.end(); ++it)
    it->doSomething();

// auto
for (auto it = v.begin(); it != v.end(); ++it)
    it->doSomething ();
// elements
for (auto & element : v) {
    element.doSomething ();
}
// for_each
std::for_each(v.begin(), v.end(), doSomething());
```