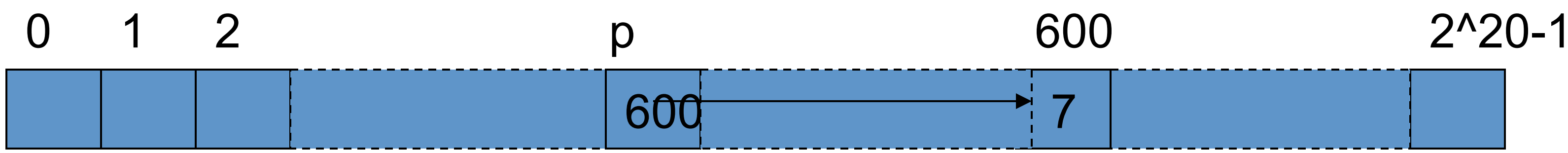
Pointers and Memory

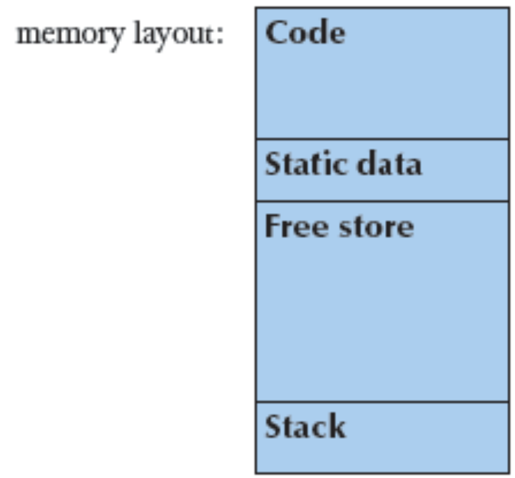
Pointer values

- Pointer values are memory addresses
  - Think of them as a kind of integer values
  - The first byte of memory is 0, the next 1, and so on
  - A pointer **p** can hold the address of a memory location



- A pointer points to an object of a given type
  - E.g. a **double*** points to a **double**, not to a **string**
- A pointer's type determines how the memory referred to by the pointer's value is used
  - E.g. what a **double*** points to can be added not, say, concatenated

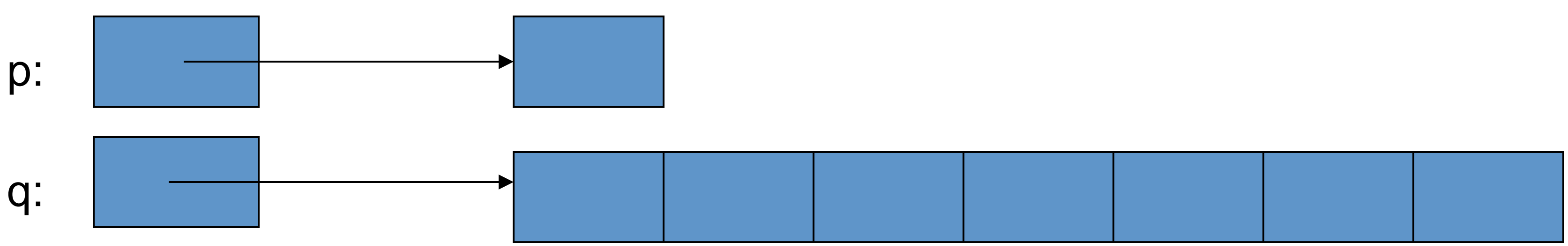
The computer's memory



- As a program sees it
  - Local variables "lives on the stack"
  - Global variables are "static data"
  - The executable code are in "the code section"

The free store
(sometimes called "the heap")

- You request memory "to be allocated" "on the free store" by the **new** operator
  - The **new** operator returns a pointer to the allocated memory
  - A pointer is the address of the first byte of the memory
  - For example
    - **int* p = new int;** *// allocate one uninitialized* ***int***
      // ***int**** *means "pointer to* ***int****"*
    - **int* q = new int[7];** *// allocate seven uninitialized* ***ints***
      // "an array of 7 ***ints****"*
    - **double* pd = new double[n];** *// allocate* ***n*** *uninitialized* ***doubles***
  - A pointer points to an object of its specified type
  - A pointer does ***not*** know how many elements it points to



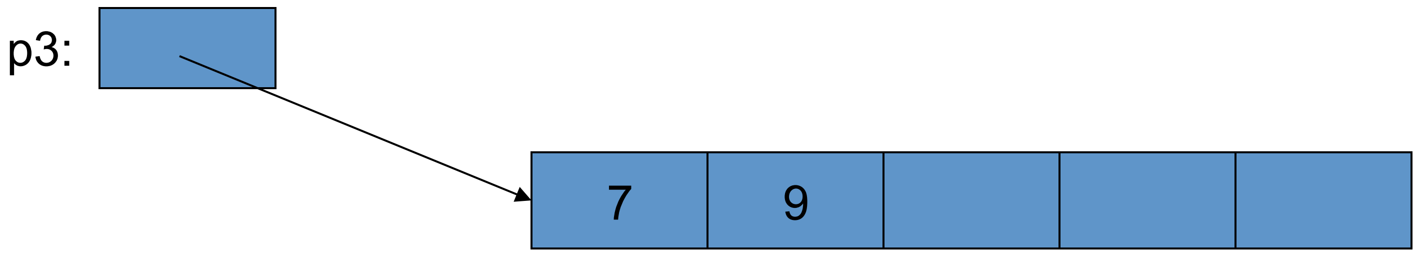
Access

- Individual elements

```
int* p1 = new int;          // get (allocate) a new uninitialized int
int* p2 = new int(5);       // get a new int initialized to 5

int x = *p2;          // get/read the value pointed to by p2
                      // (or "get the contents of what p2 points to")
                      // in this case, the integer 5
int y = *p1;          // undefined: y gets an undefined value; don't do that
```

Access



- Arrays (sequences of elements)

  int* p3 = new int[5]; *// get (allocate) 5* ***int****s*

  // array elements are numbered 0, 1, 2, …

  p3[0] = 7; *// write to ("set") the 1st element of p3*

  p3[1] = 9;

  int x2 = p3[1]; *// get the value of the 2nd element of p3*

  int x3 = *p3; *// we can also use the dereference operator * for an array*

  *// *p3 means p3[0] (and vice versa)*

Why use free store?

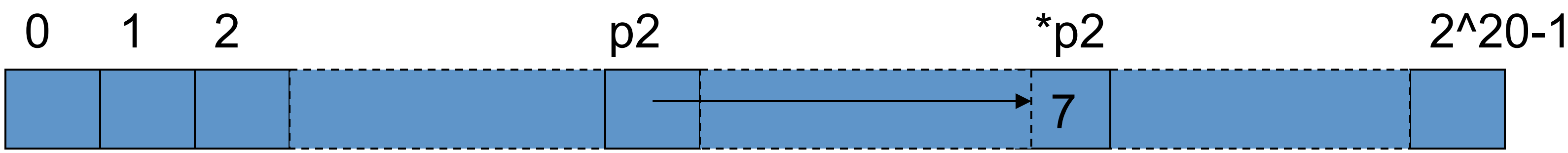
- To allocate objects that have to outlive the function that creates them:
  - For example

```
double* make(int n)  // allocate n ints
{
    return new double[n];
}
```

  - Another example: vector's constructor

Pointer values

- Pointer values are memory addresses
  - Think of them as a kind of integer values
  - The first byte of memory is 0, the next 1, and so on

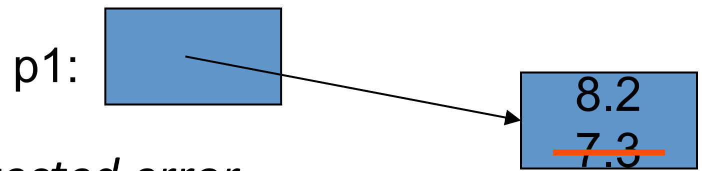



```
// you can see pointer value (but you rarely need/want to):
char* p1 = new char('c');          // allocate a char and initialize it to 'c'
int* p2 = new int(7);              // allocate an int and initialize it to 7
cout << "p1==" << p1 << " *p1==" << *p1 << "\n";  // p1==??? *p1==c
cout << "p2==" << p2 << " *p2==" << *p2 << "\n";  // p2==??? *p2=7
```

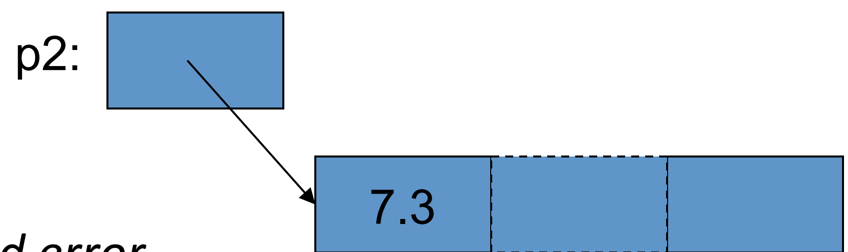
Access

- A pointer does **not** know the number of elements that it's pointing to (only the address of the first element)

```
double* p1 = new double;
*p1 = 7.3;          // ok
p1[0] = 8.2;        // ok
p1[17] = 9.4;       // ouch! Undetected error
p1[-4] = 2.4;       // ouch! Another undetected error
```




```
double* p2 = new double[100];
*p2 = 7.3;          // ok
p2[17] = 9.4;       // ok
p2[-4] = 2.4;       // ouch! Undetected error
```

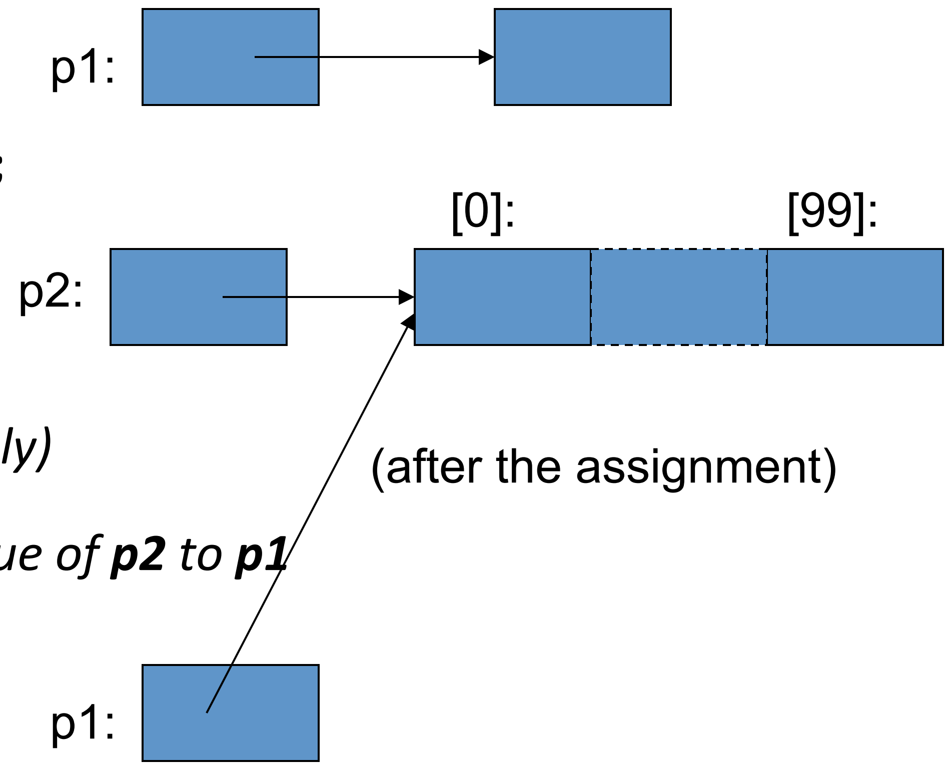


Access

- A pointer does ***not*** know the number of elements that it's pointing to

double* p1 = new double;

double* p2 = new double[100];



p1[17] = 9.4; ***//*** *error (obviously)*

p1 = p2; ***//*** *assign the value of **p2** to **p1***

p1[17] = 9.4; ***//*** *now ok: **p1** now points to the array of 100 **doubles***

Access

- A pointer ***does*** know the type of the object that it's pointing to

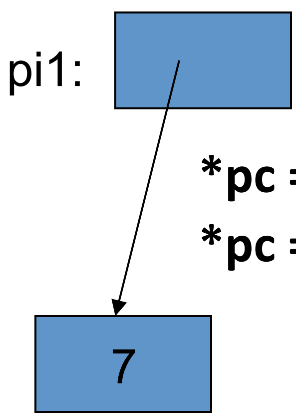
  int* pi1 = new int(7);

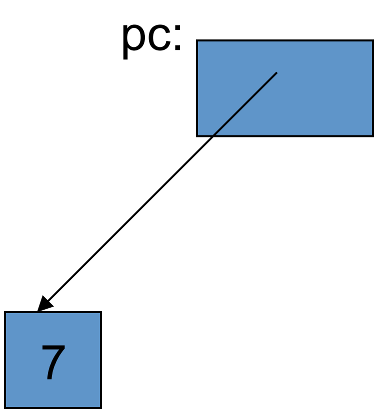
  int* pi2 = pi1; ***//*** *ok:* ***pi2*** *points to the same object as* ***pi1***

  double* pd = pi1; ***//*** *error: can't assign an* ***int**** *to a* ***double****

  char* pc = pi1; ***//*** *error: can't assign an* ***int**** *to a* ***char****

  - There are no implicit conversions between a pointer to one value type to a pointer to another value type
  - However, there are implicit conversions between value types:





pc = 8;** ***// *ok: we can assign an* ***int*** *to a* ***char***

pc = *pi1;** ***// *ok: we can assign an* ***int*** *to a* ***char***

Pointers, arrays, and vector

- Note
  - With pointers and arrays we are "touching" hardware directly with only the most minimal help from the language. Here is where serious programming errors can most easily be made, resulting in malfunctioning programs and obscure bugs
    - Be careful and operate at this level only when you really need to
  - vector is one way of getting almost all of the flexibility and performance of arrays with greater support from the language (read: fewer bugs and less debug time).

A problem: memory leak

```
double* calc(int result_size, int max)
{
    double* p = new double[max];// allocate another max doubles
                    // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // ... use p to calculate results to be put in result ...
    return result;
}

double* r = calc(200,100);    // oops! We “forgot” to give the memory
                    // allocated for p back to the free store
```

- Lack of de-allocation (usually called "memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks

A problem: memory leak

```
double* calc(int result_size, int max)
{
    int* p = new double[max];       // allocate another max doubles
                                    // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    delete[ ] p;                    // de-allocate (free) that array
                                    // i.e., give the array back to the free store
    return result;
}

double* r = calc(200,100);
// use r
delete[ ] r;                        // easy to forget
```

Memory leaks

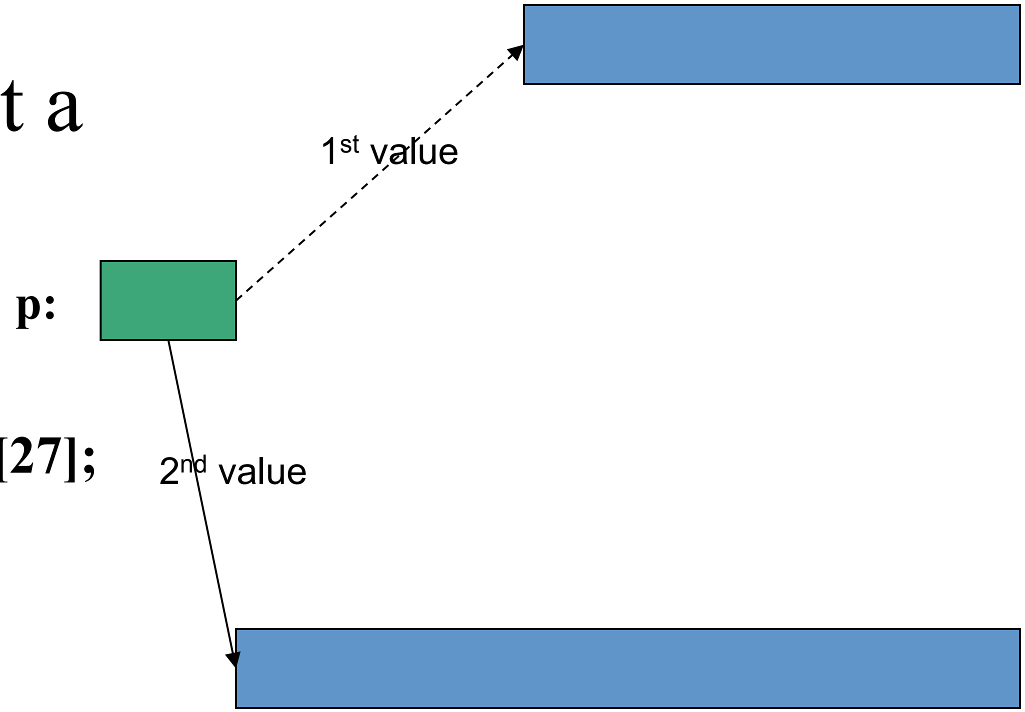
- A program that needs to run "forever" can't afford any memory leaks
  - An operating system is an example of a program that "runs forever"
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  - Trick question: not enough data to answer, but about 130,000 calls
- All memory is returned to the system at the end of the program
  - If you run using an operating system (Windows, Unix, whatever)
- Program that runs to completion with predictable memory usage may leak without causing problems
  - *i.e.,* memory leaks aren't "good/bad" but they can be a major problem in specific circumstances

Memory leaks

- Another way to get a memory leak

```
void f()
{
    double* p = new double[27];
    // …
    p = new double[42];
    // …
    delete[] p;
}
```

// 1st array (of 27 doubles) leaked



Memory leaks

- How do we systematically and simply avoid memory leaks?
  - don't mess directly with **new** and **delete**
    - Use **vector**, etc.
  - Or use a garbage collector
    - A garbage collector is a program the keeps track of all of your allocations and returns unused free-store allocated memory to the free store (not covered in this course; see http://www.research.att.com/~bs/C++.html)
    - Unfortunately, even a garbage collector doesn't prevent all leaks
    - See also Chapter 25

A problem: memory leak

```
void f(int x)
{
    int* p = new int[x];    // allocate x ints
    vector v(x);            // define a vector (which allocates another x ints)
    // ... use p and v ...
    delete[ ] p;     // deallocate the array pointed to by p
    // the memory allocated by v is implicitly deleted here by vector's destructor
}
```

- The **delete** now looks verbose and ugly
  - How do we avoid forgetting to **delete[ ] p**?
  - Experience shows that we often forget
- Prefer **delete**s in destructors

Free store summary

- Allocate using **new**
  - New allocates an object on the free store, sometimes initializes it, and returns a pointer to it
    - **int* pi = new int;** *// default initialization (none for **int**)*
    - **char* pc = new char('a');** *// explicit initialization*
    - **double* pd = new double[10];** *// allocation of (uninitialized) array*
  - New throws a **bad_alloc** exception if it can't allocate
- Deallocate using **delete** and **delete[ ]**
  - **delete** and **delete[ ]** return the memory of an object allocated by **new** to the free store so that the free store can use it for new allocations
    - **delete pi;** *// deallocate an individual object*
    - **delete pc;** *// deallocate an individual object*
    - **delete[ ] pd;** *// deallocate an array*
  - Delete of a zero-valued pointer ("the null pointer") does nothing
    - **char* p = 0;**
    - **delete p;** *// harmless*

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
  - Example: the arguments of a callback function
- 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*

- 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*
                  // good! The int 7 is not represented  like the double 7.0)
    pv[2] = 9;         // error: you can't subscript a void*
    pv++;              // error: you can't increment a void*
    int* pi = static_cast<int*>(pv); // ok: explicit conversion
    // ...
}
```

- 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

void*

- **void*** is the closest C++ has to a plain machine address
  - Some system facilities require a **void***

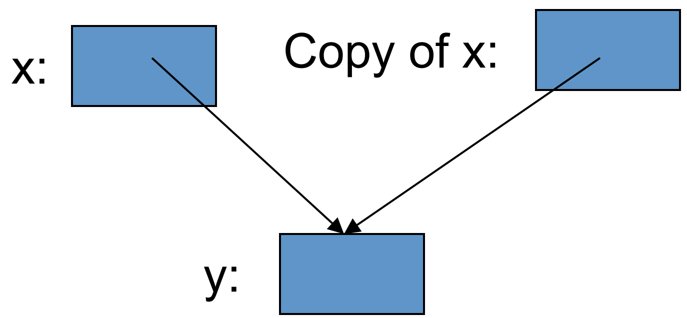
Pointers and references

- 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

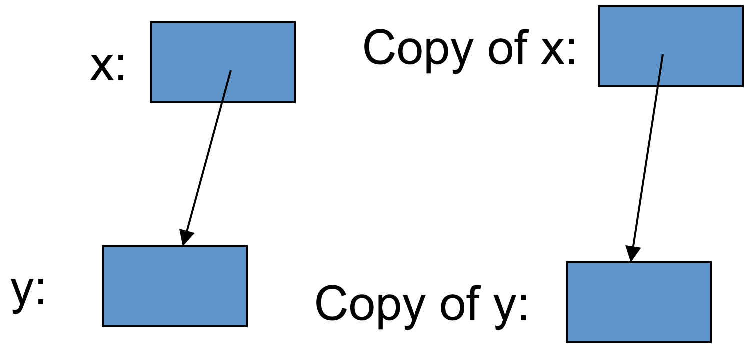
```
int x = 7;
int y = 8;
int* p = &x;   *p = 9;
p = &y;   // ok
int& r = x;     x = 10;
r = &y;   // error (and so is all other attempts to change what r refers to)
```

Copy terminology

- Shallow copy: copy only a pointer so that the two pointers now refer to the same object
  - What pointers and references do
- Deep copy: copy the pointer and also what it points to so that the two pointers now each refer to a distinct object
  - What **vector**, **string**, etc. do
  - Requires copy constructors and copy assignments for container classes



Shallow copy



Deep copy

Arrays

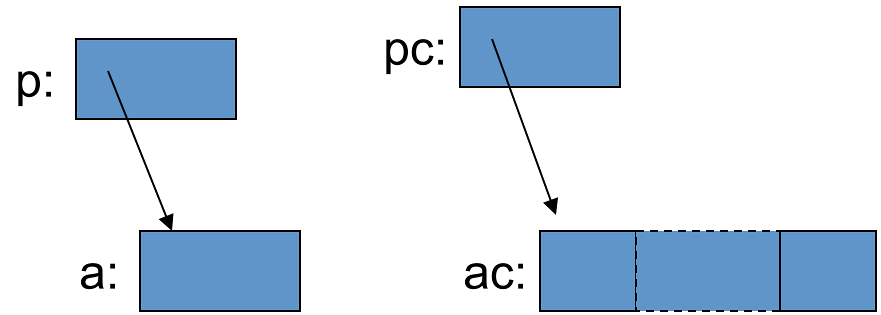
- Arrays don't have to be on the free store

```
char ac[7];          // global array – "lives" forever – "in static storage"
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
                     // vector<double> lx(n); would work
    // …
}
```

Address of: &

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

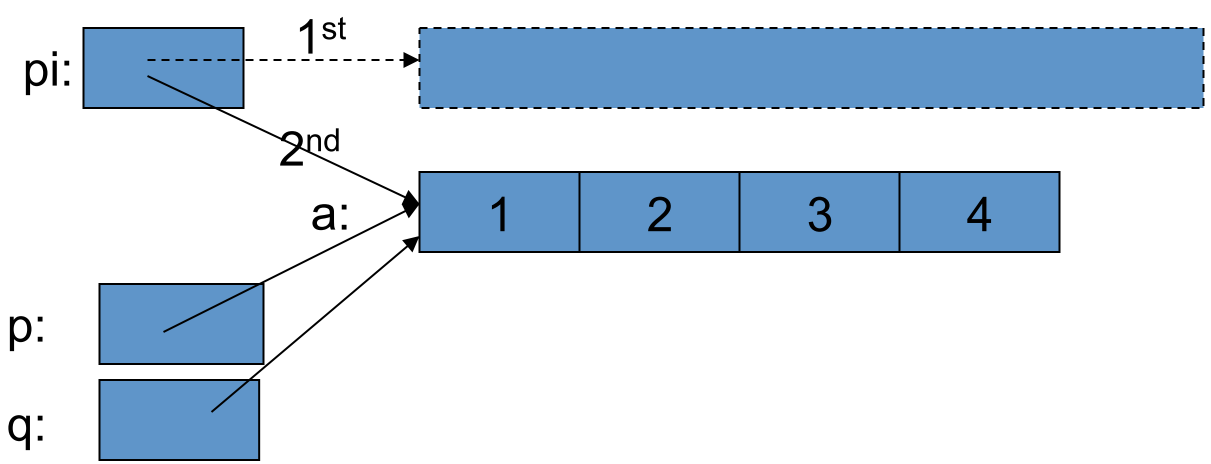



```
int a;
char ac[20];

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 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
}
```

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)
}
```

Why bother with arrays?

- It's all that C has
  - In particular, C does not have vectors
  - There is a lot of C code "out there"
    - Here "a lot" means N*1B lines
  - There is a lot of C++ code in C style "out there"
    - Here "a lot" means N*100M lines
  - You'll eventually encounter code full of arrays and pointers
- They represent primitive memory in C++ programs
  - We need them (mostly on free store allocated by **new**) to implement better container types
- Avoid arrays whenever you can
  - They are the largest single source of bugs in C and (unnecessarily) in C++ programs
  - They are among the largest sources of security violations (usually (avoidable) buffer overflows)

Initialization syntax
(array's one advantage over vector)

```
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
```