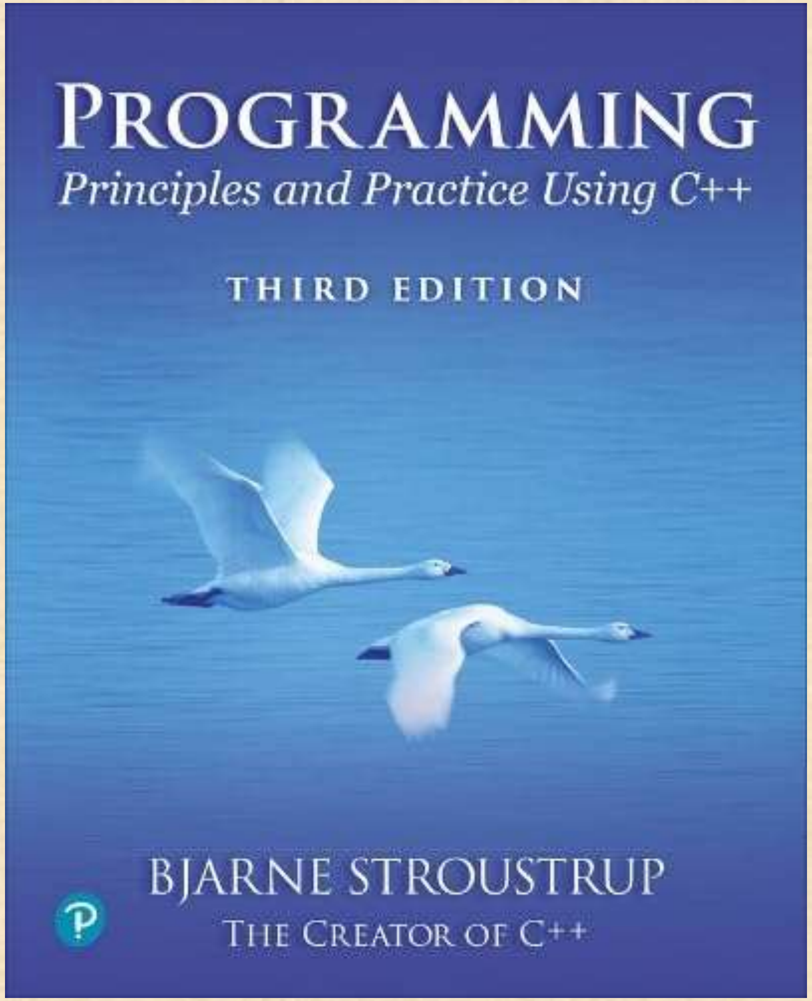
Chapter 15 – **vector** and Free Store



Use ***vector*** *as the default!*
– Alex Stepanov

Overview: evolving a **vector** type

(and spanning the low-level/high-level language gap in the process)

- Chapter 15
  - Dealing with “raw” memory: pointers and free store
  - Destructors
- Chapter 16
  - Arrays and pointers
  - Pointers and references
  - C-style strings
  - Alternatives to low level facilities: **span**, **array**, and **string**
- Chapter 17
  - Copying and moving
  - Essential operations for managing object lifecycles
- Chapter 18
  - Templates and generic programming
  - Exceptions and scope-based resource management and error handling (RAII)
  - Resource management pointers: **unique_ptr** and **shared_ptr**

Vector

- **vector** is the most useful container
  - ISO standard
  - Simple to use
  - Compactly stores elements of a given type
  - Efficient access
  - Expands to hold any number of elements
  - Optionally range-checked access
    - the PPP version is range checked
- How is that done?
  - That is, how is **vector** implemented?
    - We'll answer that gradually, feature after feature
- Prefer **vector** for storing elements unless there's a good reason not to

Building from the ground up

- The hardware provides memory and addresses
  - Low level
  - Untyped
  - Fixed-sized chunks of memory
  - No checking
  - As fast as the hardware architects can make it
- The application builder needs something like a **vector**
  - Higher-level operations
  - Type checked
  - Size varies (as we get more data)
  - Run-time range checking
  - Close to optimally fast

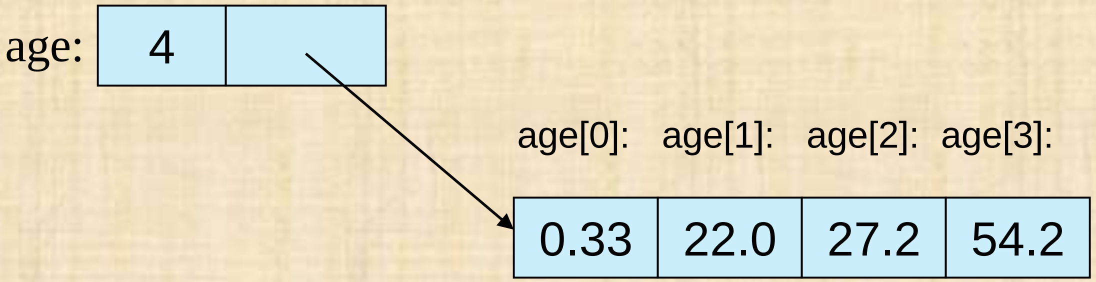
Building from the ground up

- At the lowest level, close to the hardware, life's simple and brutal
  - You have to program everything yourself
  - You have no type checking to help you
  - Run-time errors are found when data is corrupted or the program crashes
- We want to get to a higher level as quickly as we can
  - To become productive and reliable
  - To use a language "fit for humans"
- Chapters 15-18 basically show all the steps needed
  - The alternative to understanding is to believe in "magic"
  - The techniques for building **vector** are the ones underlying all higher-level work with data structures

Vector

- A **vector**
  - Our **Vector** will gradually be improved to approximate the standard **vector**
  - Can hold an arbitrary number of elements
    - Up to whatever physical memory and the operating system can handle
  - That number can vary over time
    - E.g. by using **push_back()**
  - Example

```
Vector<double> age(4);
age[0]=.33;   age[1]=22.0;   age[2]=27.2;   age[3]=54.2;
```



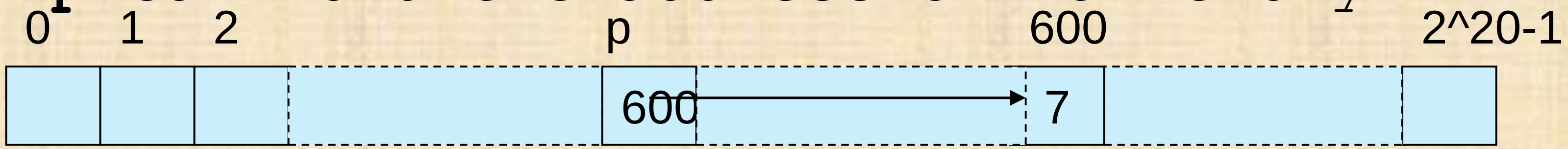
Vector

```
class Vector {     // a very simplified vector of doubles (like
 vector<double>)
        int sz;                    // the number of elements ("the size")
        double* elem;              // pointer to the first element
public:
        Vector(int s);                   // constructor: allocate s elements,
 let elem point to them
        int size() const { return sz; }      // the current size
};
```

- ***** means "pointer to" so **double*** is a "pointer to **double**"
  - What is a "pointer"?
  - How do we make a pointer "point to" elements?
  - How do we "allocate" elements?

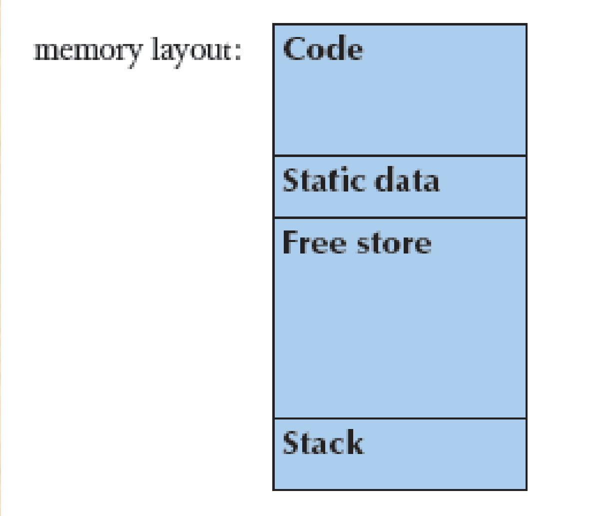
Pointer values

- 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 but not, say, concatenated

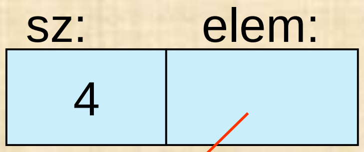
The computer's memory



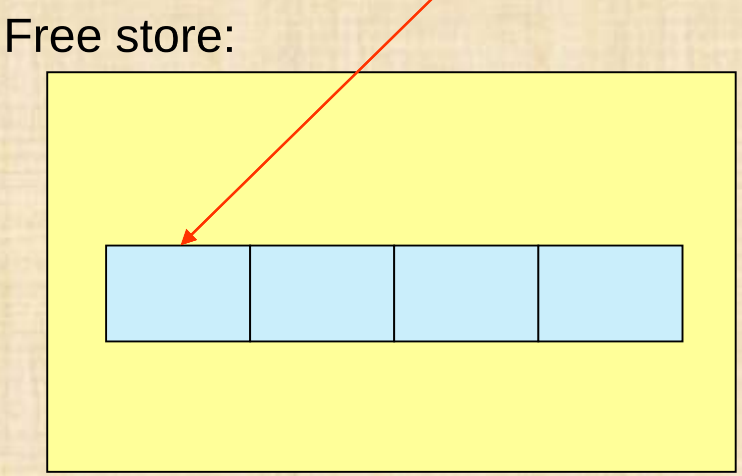
- As a program sees it
  - Local variables "live on the stack" (including function arguments)
  - Global variables are "static data"
  - The executable code is in "the code section"

Vector (constructor)

```
Vector::Vector(int s)    // Vector's constructor
       :sz(s),                 // store the size s in sz
       elem(new double[s])     // allocate s doubles on the free
                         // store a pointer to those doubles in elem
{
       // Note: new does not initialize elements (but the standard vector
  does)
}
```

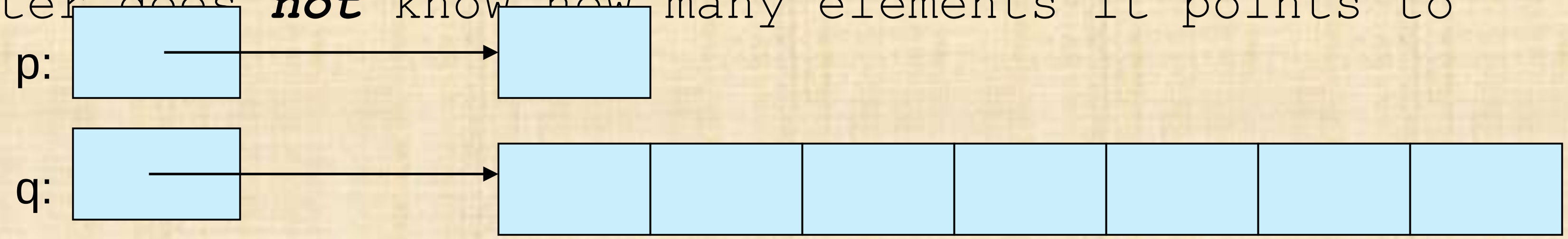


- **new** allocates memory from the free store and returns a pointer to the allocated memory
- We use **new** to allocate objects that have to outlive the function that creates them



The free store
(sometimes called "the heap" or "dynamic memory")

- 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
    - `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

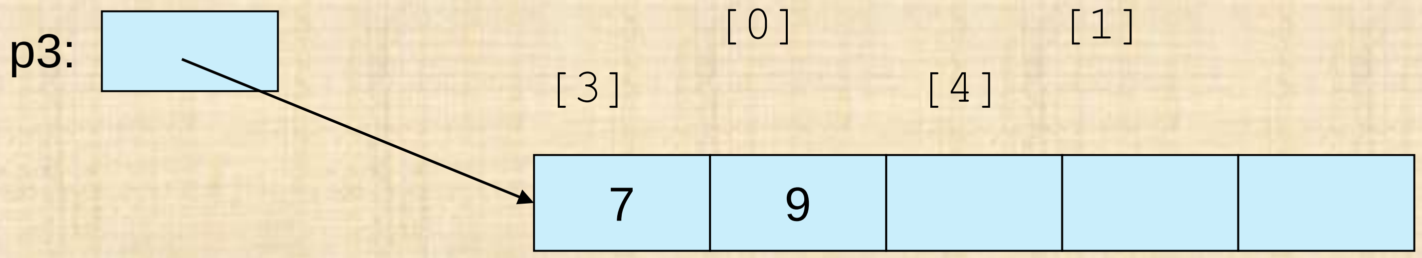


Access

- 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

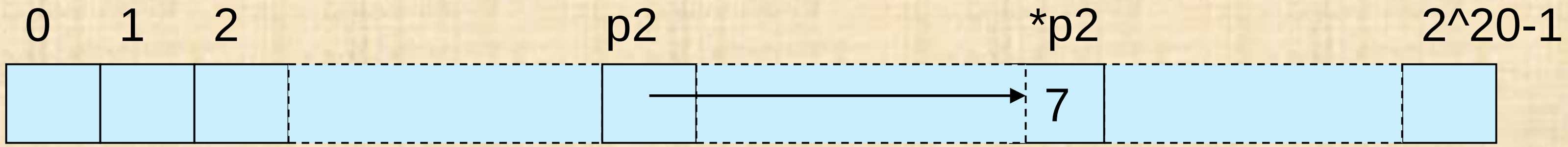


- Arrays (sequences of elements)

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

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



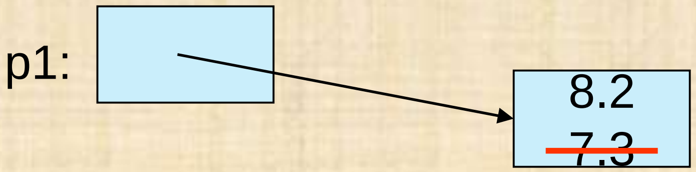
// you can see a pointer value (but you rarely need/want to):

```
int* p1 = new int(7);                                // allocate an int and initialize it to 7
double* p2 = new double(7);                          // allocate a double and initialize it to 7.0
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
```



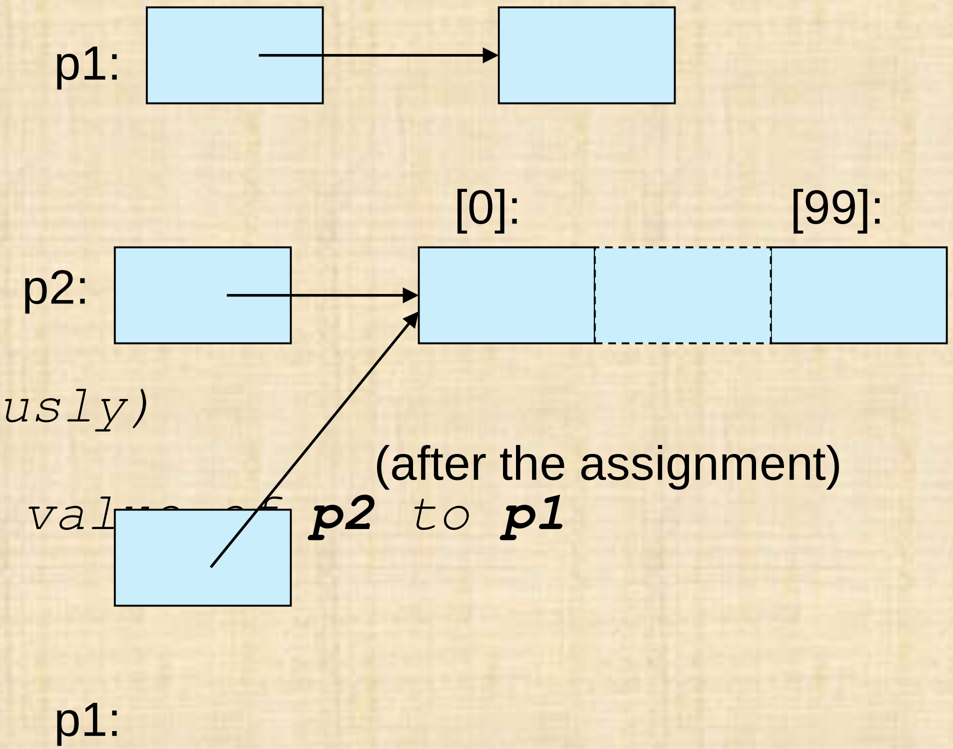

```
double* p2 = new double[100];
*p2 = 7.3;              // ok
p2[17] = 9.4;           // ok
p2[-4] = 2.4;           // ouch! Undetected error
```

- Fortunately, we have ways of avoiding such errors

Access

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

```
double* p1 = new double;
double* p2 = new double[100];
```



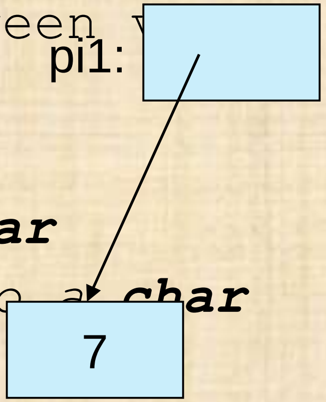

```
p1[17] = 9.4;     // error (obviously)

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

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

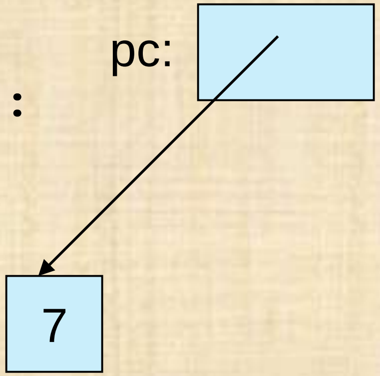
Access

- 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
```

The null pointer

- Sometimes, we need to say "this pointer doesn't point to anything just now"

```
double* p = nullptr;
// …
If (p!=nullptr)         // p points to something
    *p = 7;
else
    *p = 9;             // No! never do this, p doesn't point to anything
```

- More concisely, we can leave out the **!=nullptr**

```
If (p)          // p points to something, aka "p is valid''
    *p = 7;
```

- **nullptr** is commonly used to indicate
  - End of a linked list
  - No pointer value available just now
  - No object to return a pointer to

A problem: memory leak

```
double* calc(int result_size, int max)
{
	double* p = new double[max];		// allocate max doubles on 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 by **new** back to the free store
  - That doesn't happen automatically ("no garbage collection")
  - 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

We can give memory back to the free store: **detete**

```
double* calc1(int result_size, int max)
{
	double* tmp = new double[max];		// allocate max doubles on free store
	double* result = new double[result_size];
	// … use tmp to calculate results to be put in result …
	delete[] tmp;				// return the memory pointed to by tmp to the free store
	return result;
}

double* r = calc1(200,100);	// oops!
// … use r …
delete[ ] r;			// return the memory pointed to by r to the free store
```

Memory leaks – resource leaks

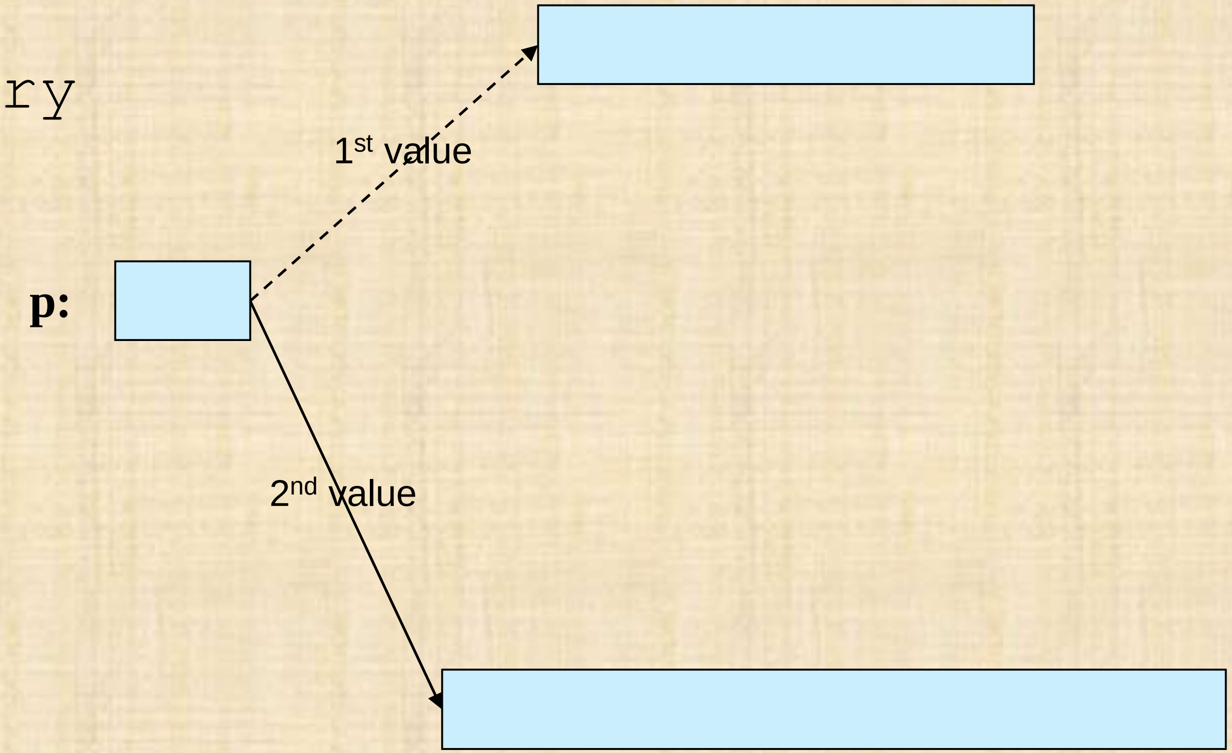
- A program that needs to run “forever” can’t afford any memory leaks
  - An operating system is an example of a program that “runs forever”
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  - Trick question: not enough data to answer, 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 is a major real-world problem
- Memory leaks is just a special case of resource leaks
  - E.g., files, locks, sockets

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?
  - Use resource handles
    - Use **vector**, etc.
    - don't mess directly with **new** and **delete**
  - 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 common in C++
      - not covered in this course; see http://www.stroustrup.com/C++.html
    - Unfortunately, a garbage collector doesn't prevent all resource leaks
      - Only memory leaks

A problem: memory leak

```
void f(int x)
{
	Vector v(x);		// define a Vector  (which allocates x
  doubles on the free store)
	// … use v …

	// give the memory allocated by v back to the free store
	// but how? (Vector's elem data member is private)
}
```

Vector (destructor)

```
// a very simplified Vector of doubles:
class Vector {
 int sz;                    // the size
 double* elem;              // a pointer to the elements
public:
 Vector(int s)  :sz(s), elem(new double[s]) { }       // constructor: allocates/acquires memory
 ~Vector() { delete[ ] elem; }                        // destructor: de-allocates/releases memory
     // …
};
```

- Note: this is an example of a general and important technique:
  - acquire resources in a constructor
  - release them in the destructor
- Examples of resources: memory, files, locks, threads, sockets

Implicitly give memory back to the free store

```
Vector<double> calc2(int result_size, int max)
{
      Vector<double> tmp(max);            // allocate max doubles
 on free store
      Vector<double> result(result_size);
      // … use tmp to calculate results to be put in result …
      return result;
} // tmp destroyed upon return

void user()
{
 // …
 auto res = calc2(200,100);   // oops!
 // … use r …
}     // res destroyed upon return
```

Simpler and probably more efficient than using **new** and **delete** explicitly

(yes, we can avoid copying a result
- Next lecture)

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 (out of memory)
- 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 = nullptr;**

Avoid “naked new” and “naked delete”

- Manual resource allocation and deallocation is error-prone
  - We forget to hand resources back (Ask any librarian)
  - We use pointers after the memory they point to has been deleted
  - Use of “raw pointers” to manage memory leads to overuse of pointers
- Using **vector** leads to simpler code
  - Compare calc1() and calc2()
  - See the following lectures and chapters

Generated destructors

- If a member of a class has a destructor, then that destructor will be called when the object containing the member is destroyed.

```
struct Customer {
     string name;
     vector<string> addresses;
     // ...
};

void some_fct()
{
     Customer fred { "Fred", {"17 Oak St.", "232 Rock Ave."}};
     // ... use fred ...
}
```

- That saves us a lot of work
  - And avoid a lot of bugs

Virtual destructors

- Destructors work correctly in class hierarchies
  - Provided you declare the destructor **virtual**

```
Shape* fct()
{
    Text tt {Point{200,200},"Anya"};                // local Text variable
    // ...
    return new Text{Point{100,100},"Courtney"};           // Text object
on the free store
}

void user()
{
    Shape* q = fct();
    // ... use the Shape without caring exactly which kind of shape it
is ...
    delete q;          // Shape's destructor is virtual so
Text::~Text() is called if *q is a Text
```

But, what about "no naked **delete**s"?

- Use "resource-manegement" pointers ()

```
unique_ptr<Shape> fct()
{
        Text tt {Point{200,200},"Annemarie"};                        // local Text variable
        // ...
        return make_unique<Text>(Point{100,100},"Nicholas");         // Text object on the free store
}

void user()
{
        unique_ptr<Shape> q = fct();
        // ... use the Shape without caring exactly which kind of shape it is ...
}
```

- Equivalent to the previous example, but simpler

Access to elements

- But our Vector doesn't have access operations
  - So, let's add very simple ones

```
class Vector {                          // a very simplified vector of doubles
public:
        Vector(int s) :sz{s}, elem{new double[s]} { /* ... */ }      // constructor
        ~Vector() { delete[] elem; }                                  // destructor

        int size() const { return sz; }                               // the current size

        double get(int n) const { return elem[n]; }                   // access: read
        void set(int n, double v) { elem[n]=v; }                      // access: write
private:
        int sz;                         // the size
        double* elem;                   // a pointer to the elements
};
```

Access to elements

- Not very elegant
  - But it’ll do for now

```
Vector v(5);
for (int i=0; i<v.size(); ++i) {
	v.set(i,1.1*i);
	cout << "v[" << i << "]==" << v.get(i) << '\n';
}
```

Reminder

- Why look at the **Vector** implementation?
  - To see how the standard library **vector** really works
  - To introduce basic concepts and language features
    - Free store (heap)
    - Copying
    - Dynamically growing data structures
  - To see how to directly deal with memory
  - To introduce the techniques and concepts we need to understand C
    - Including the dangerous ones (and see how to avoid those in C++)
  - To demonstrate class design techniques
  - To see examples of “neat” code and good design

Next lecture

- We'll see how we can change our **Vector**'s implementation to better allow for changes in the number of elements. Then we'll modify **Vector** to take elements of an arbitrary type and add range checking. That'll imply looking at templates and revisiting exceptions.