Chapter 17: Vector and Free Store

Plan for today

- We will talk about:
  - `vector` class (again, but in more details)
  - Memory
  - Addresses
  - Pointers

Vector

- Vector is the most useful container
  - Simple
  - Compactly stores elements of a given type
  - Efficient access
  - Expands to hold any number of elements
  - Optionally range-checked access

Vector

- Vector is the most useful container
  - Simple
  - Compactly stores elements of a given type
  - Efficient access
  - Expands to hold any number of elements
  - Optionally range-checked access
- How is that done?
  - That is, how is vector implemented?
    - We'll answer that gradually, feature after feature

Vector

- Vector is the most useful container
  - Simple
  - Compactly stores elements of a given type
  - Efficient access
  - Expands to hold any number of elements
  - Optionally range-checked access
- How is that done?
  - That is, how is vector implemented?
    - We'll answer that gradually, feature after feature
- Vector is the default container
  - 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

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

Vector

- A `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()`

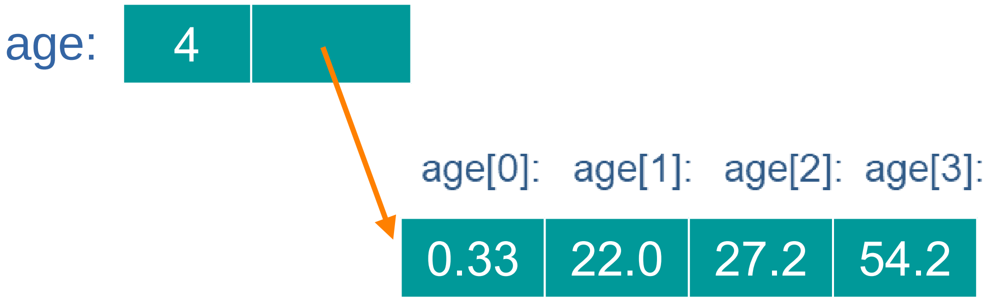
Vector

- A `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

- A `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:



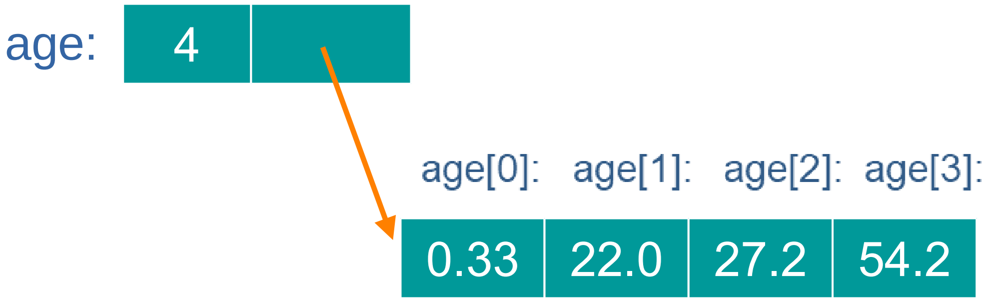

```
vector<double> age(4);
age[0]=.33;
age[1]=22.0;
age[2]=27.2;      age[3]=54.2;
```

Vector

- A `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:

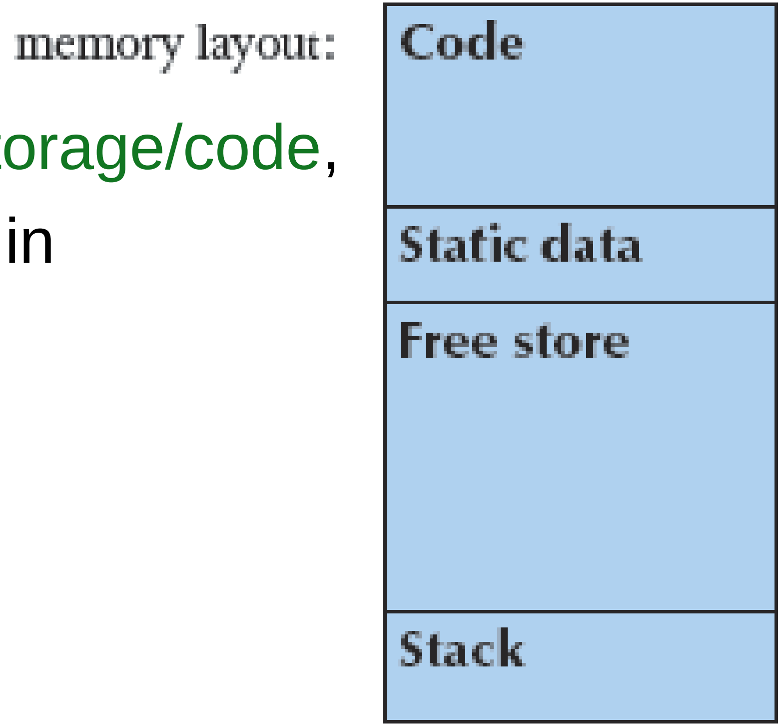
```
vector<double> age(4);
age[0]=.33;
age[1]=22.0;
age[2]=27.2;      age[3]=54.2;
```

From where does the vecor gets the space for its elements?



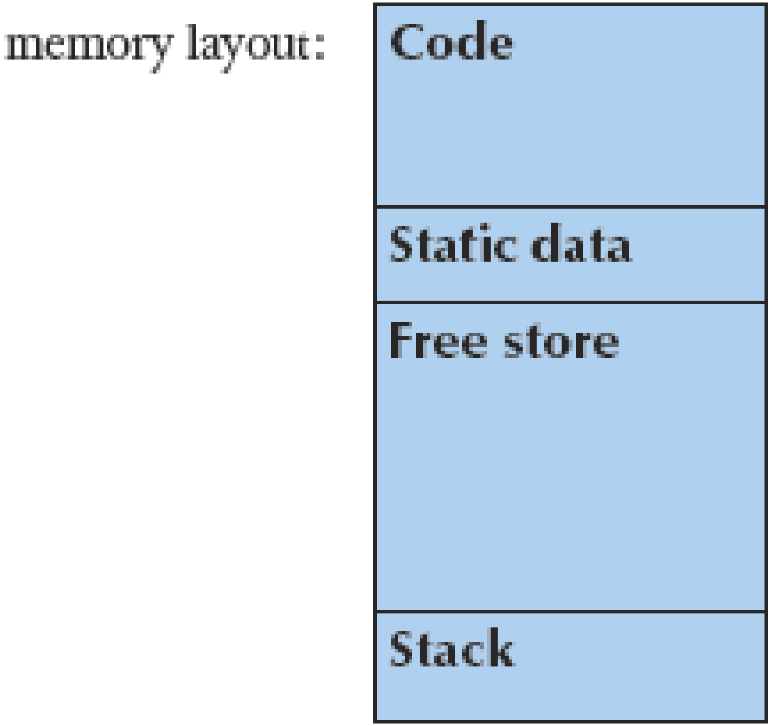
The computer’s memory

- When we start a C++ program,
  the compiler sets aside memory for
  - our code, called code storage/text storage/code,
  - local variables, including arguments in function calls, called stack
  - global variables we define, called static storage / static data



The free store

- The free store is sometimes called "the heap" and is used for *dynamic memory allocation*



The free store

- The free store is sometimes called "the heap" and is used for *dynamic memory allocation*
- We 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

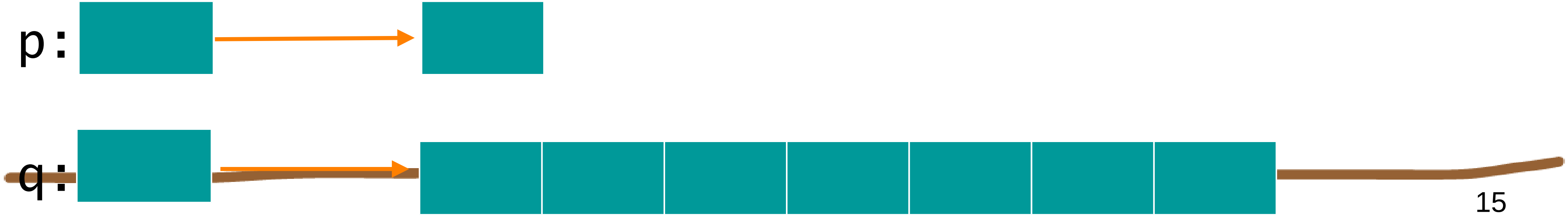
memory layout:

The free store

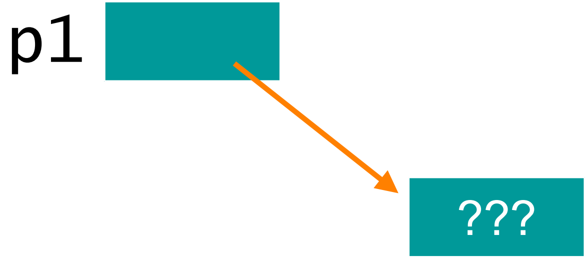
- Example:

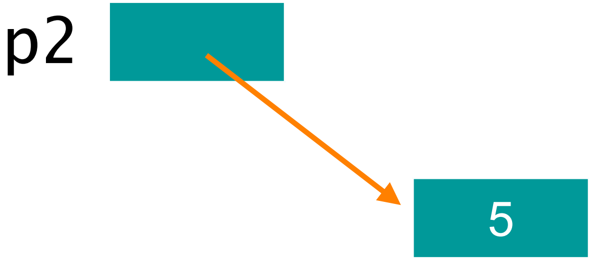
```
int* p = new int; // allocate one uninitialized  int
                  // int* means “pointer to int”
int* q = new int[7];      // allocate seven uninitialized ints
                          // “an array of 7 ints”
double* pd = new double[n]; // allocate n uninitialized doubles
```

- A pointer points to an object of its specified type
- A pointer does not know how many elements it points to



Access

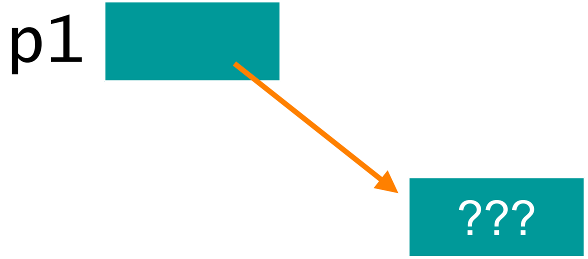


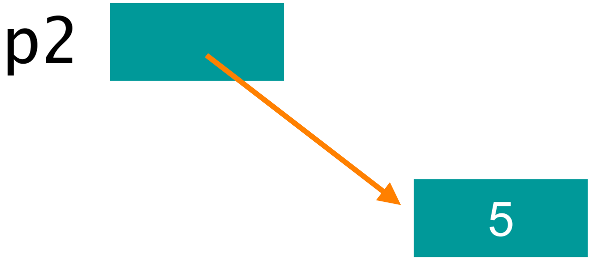


- Individual elements

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

Access



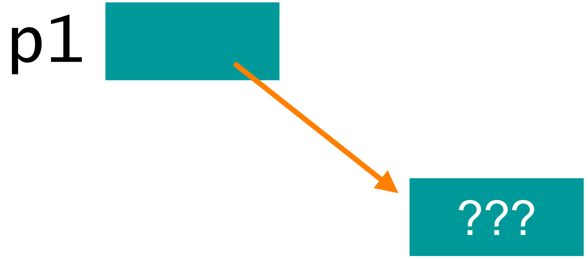


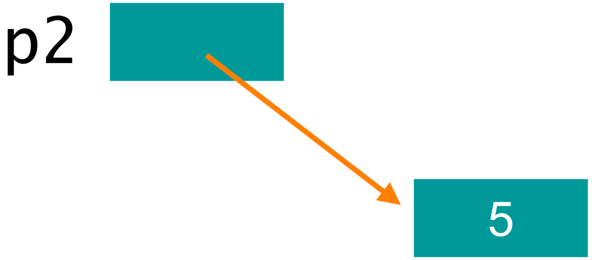
- Individual elements

```
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;     // what does it do?
```

Access



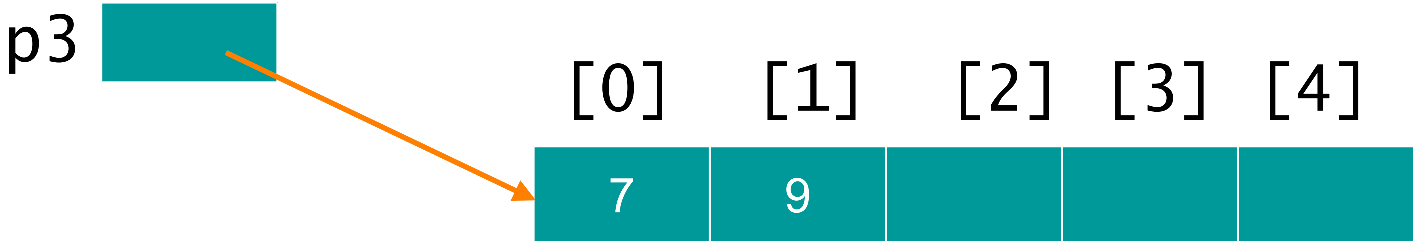


- Individual elements

```
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
```

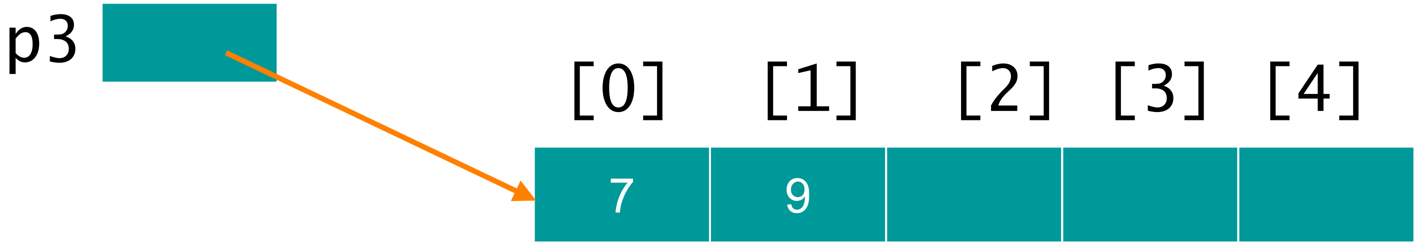
Access



- Arrays (sequences of elements)

```
int* p3 = new int[5];          // get (allocate) 5 ints
                               // array elements are numbered [0], [1], [2], …
```

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;                // 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

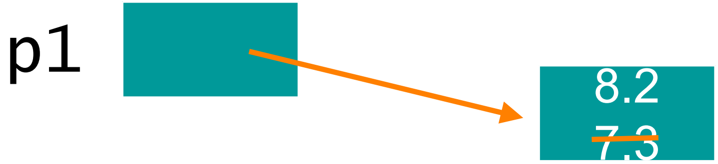
- Pointer values are memory addresses
  - Think of them as a kind of integer values
  - The first byte of memory is 0, the next 1, and so on


```
// you can see 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
```

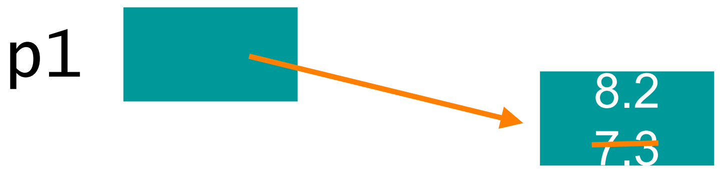


Access

- 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
```

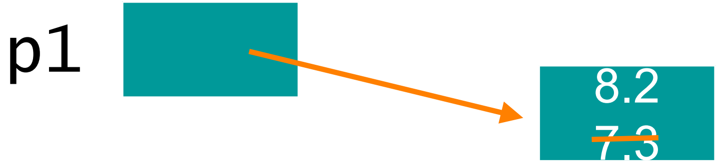


Access

- 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
```

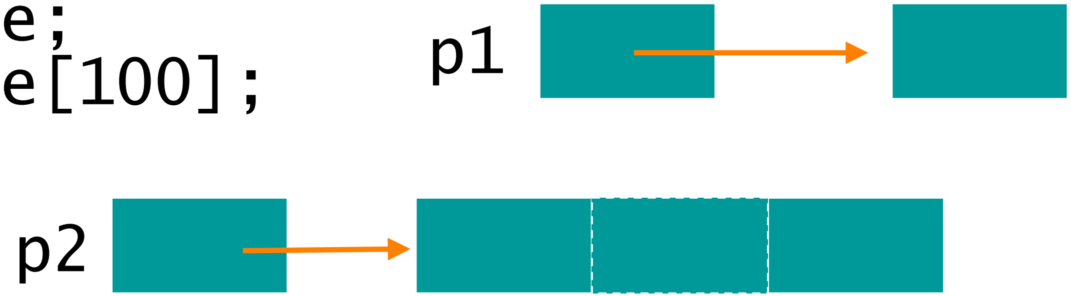



```
double* p2 = new double[100];
*p2 = 7.3;        // ok
p2[17] = 9.4;         // ok
```

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



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


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

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


```
p1[17] = 9.4;     // error (obviously)
p1 = p2;          // assign the value of p2 to p1
```

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;
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 **not** know the number of elements that it's pointing to (only the address of the first element)

```
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*
```

Access

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

```
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

Access

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

```
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:

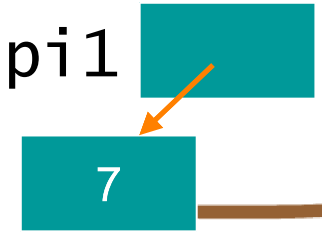
```
*pc = 8;      // ok: we can assign an int to a char
*pc = *pi1;  // ok: we can assign an int to a char
```

Access

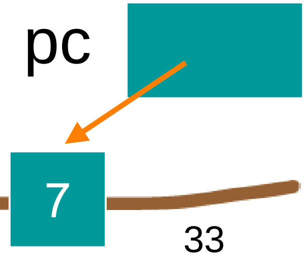
- A pointer does **not** know the number of elements that it's pointing to (only the address of the first element)

```
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
```



References

- “reference” is a general concept

```
int i = 7;
int& r = i;
r = 9;  // i becomes 9
const int& cr = i;
// cr = 7;    // error: cr refers to const
i = 8;
cout << cr << endl; // write out the value of i (that’s 8)
```

- You can think of a reference as an alternative name for an object (alias)
- You can’t modify an object through a `const` reference (recall passing parameters by reference)
- You can’t make a reference refer to another object after initialization

For loop example with and without references

- Consider the following range-for loops:

```
for (string s : v) cout << s << "\n";
// s is a copy of some v[i]

for (string& s : v) cout << s << "\n";
// no copy

for (const string& s : v) cout << s << "\n";
// and we don't modify v
```

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

Pointers and references

- Think of a reference as an automatically <u>dereferenced</u> 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)
```

In-class practice

- Allocate an array of 100 *floating point values* on the free store using new. Initialize it with values 2*i+1, where I runs from 0 to 99. Display all the values using cout. Deallocate the array (using `delete []`) and announce that it was deallocated.
- Write a function `displayArray(ostream& out, double *a, int n)` that prints out the values of `a`, assuming that a has `n` elements, to `out`.
- Consider the following code and make a sketch of the memory for it:

```
int x = 7, y = 8;
int* p = &x;    *p += 5;
int *p2 = new int;
*p2 = *p;
delete p2;
```

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html
- Class textbook