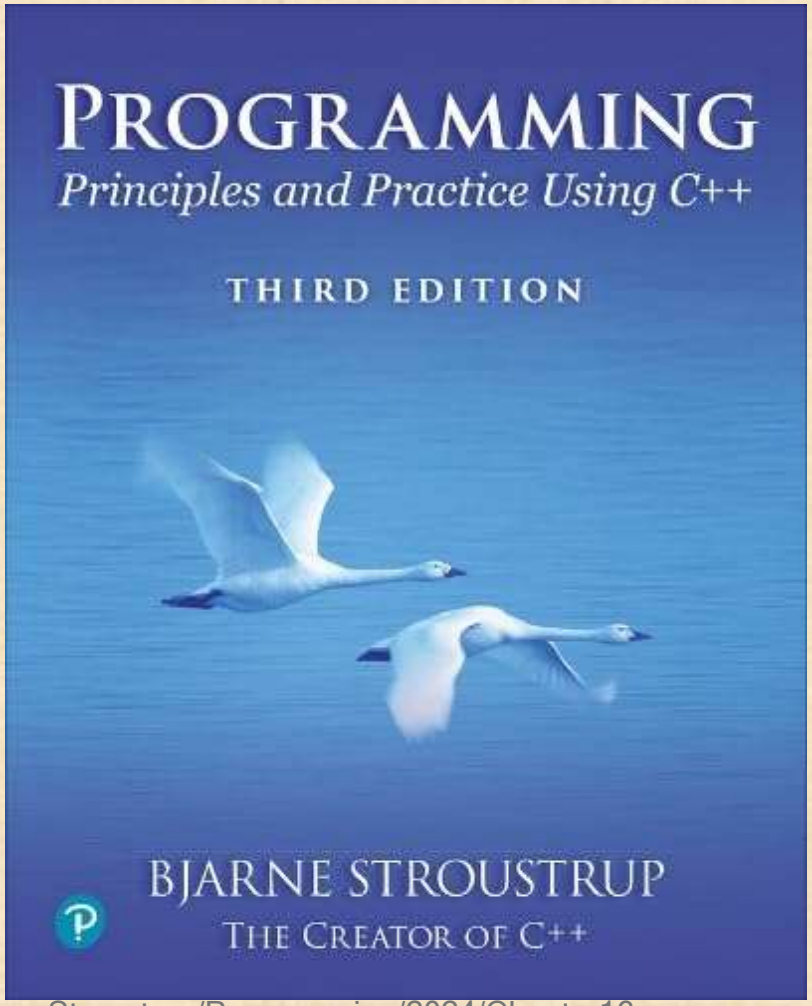
Chapter 16 – Arrays, Pointers, and References



Caveat emptor!
– Good advice

Abstract

This lecture describes the lower-level notions of arrays and pointers. We consider the uses of pointers, such as array traversal and address arithmetic, and the problems arising from such use. We also present the widely used C-style string; that is, a zero-terminated array of **char**s. Pointers and arrays are key to the implementation of types that save us from error-prone uses of pointers, such as **vector**, **string**, **span**, **not_null**, **array**, **unique_ptr**, and **shared_ptr**.

As an example, we show a few ways we can implement a function that determines whether a sequence of characters represents a palindrome.

Overview

- Arrays
- Pointer arithmetic
- Pointers and references
- Pointer and reference parameters
- Pointers as parameters
- C-style strings
- Alternatives to pointer use
  - **span**
  - **array**
  - **not_null**
- An example: palindromes

Arrays

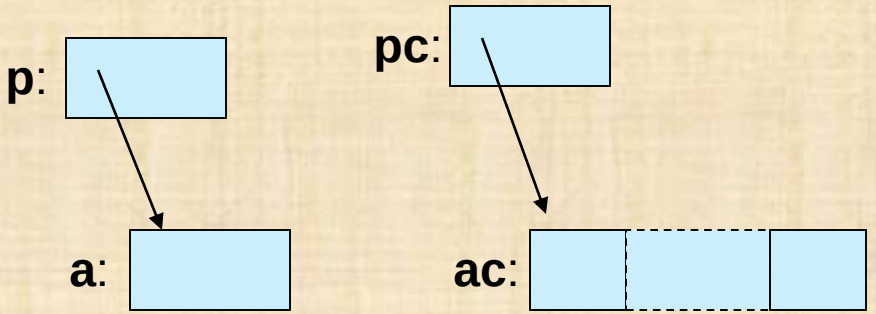
- Arrays don't have to be on the free store

```
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: &

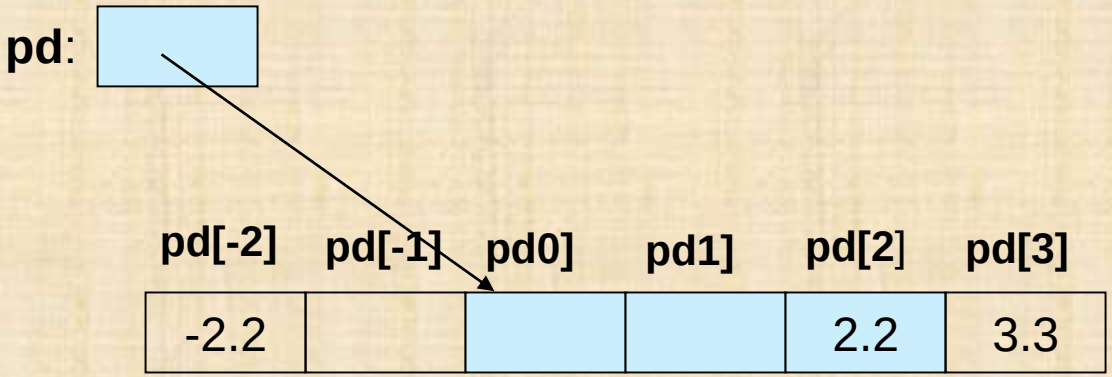
- You can get a pointer to any object
  - not just to objects on the free store

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

void f(int n)
{
    int b;
    int* p = &b;          // pointer to individual variable
    p = &a;               // now point to a different variable
    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 and pointers

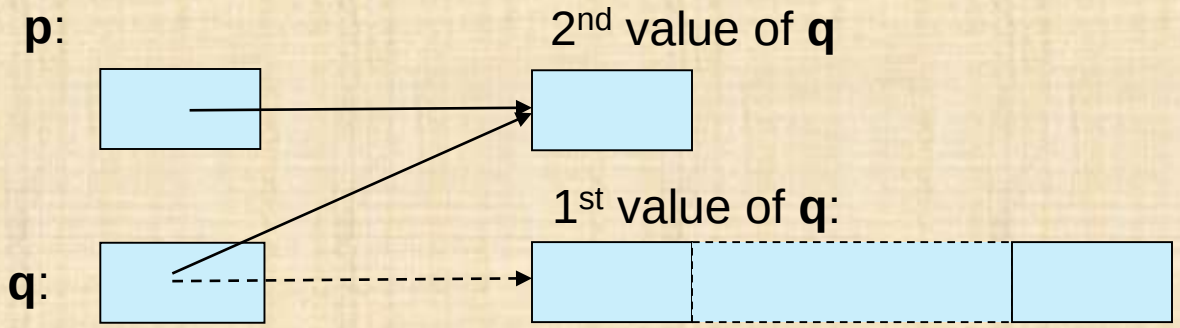
```
void use(double* pd)
{
        pd[2] = 2.2;
        pd[3] = 3.3;
        pd[-2] = -2.2;
}

void test()
{
        double arr[3];
        use(arr);                    // arr converts to a pointer to arr[0] when used as an argument
}
```



- But **pd** doesn't point to an array with elements **pd[-2]**, **pd[-1]**, or **pd[3]**!
- Don't subscript when you don't know the range of elements available
  - Leave subscripting of pointer to the implementations or range-checked containers
  - Use **vector**

Another way of getting into trouble with arrays

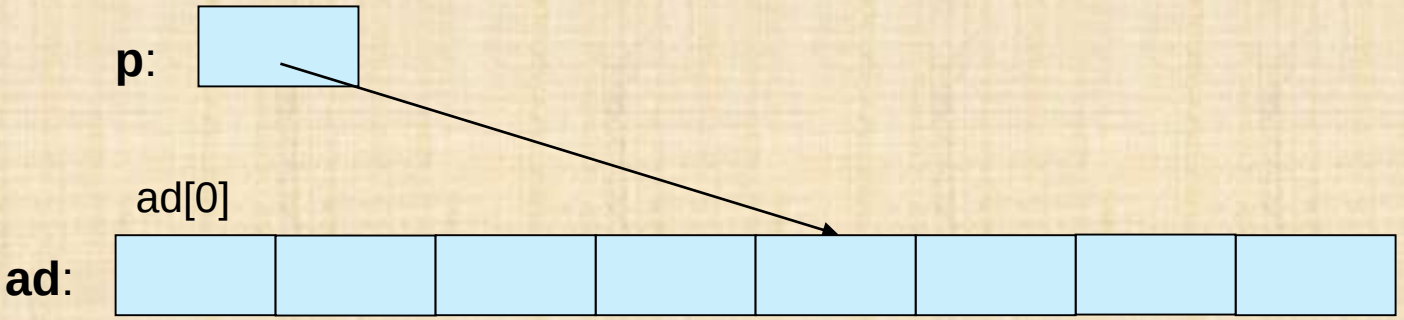
```
double* p = new double;// allocate a double
double* q = new double[1000];  // allocate 1000 doubles

q[700] = 7.7;                  // fine: q points to 1000 doubles
q = p;                         // let q point to the same object as p does
double d = q[700];             // bad: q points to a single double: out-of-range access!
```



- Likely disaster!
  - And we leaked 1000 doubles
  - Leave subscripting of pointer to the implementations or range-checked containers
  - Use **vector**

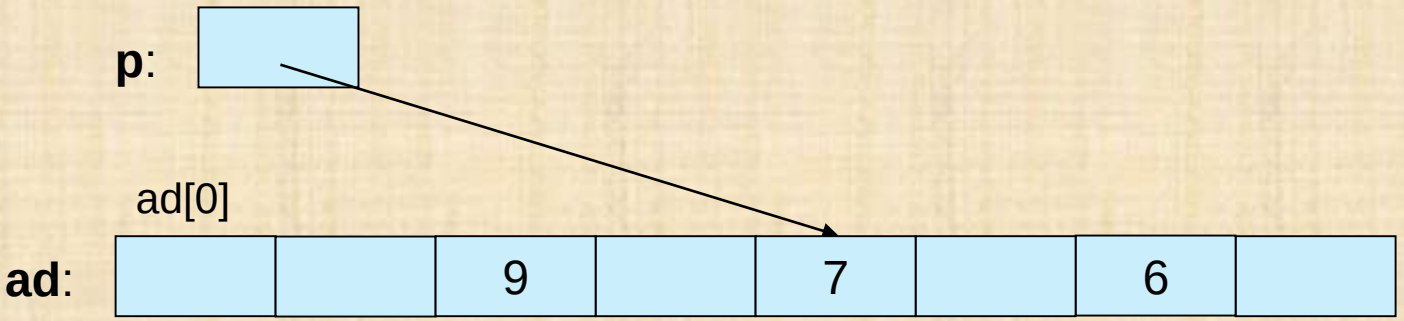
Pointer arithmetic

```
double ad[8];
double* p = &ad[4];          // point to ad[4]; the 5th element of ad
```



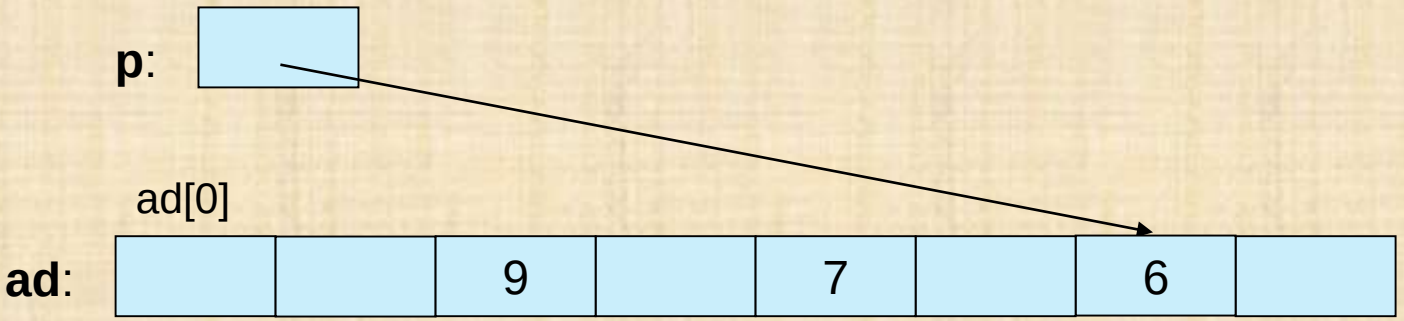
Pointer arithmetic

```
double ad[8];
double* p = &ad[4];          // point to ad[4]; the 5th element of ad

*p = 7;
p[2] = 6;
p[-2] = 9;
```



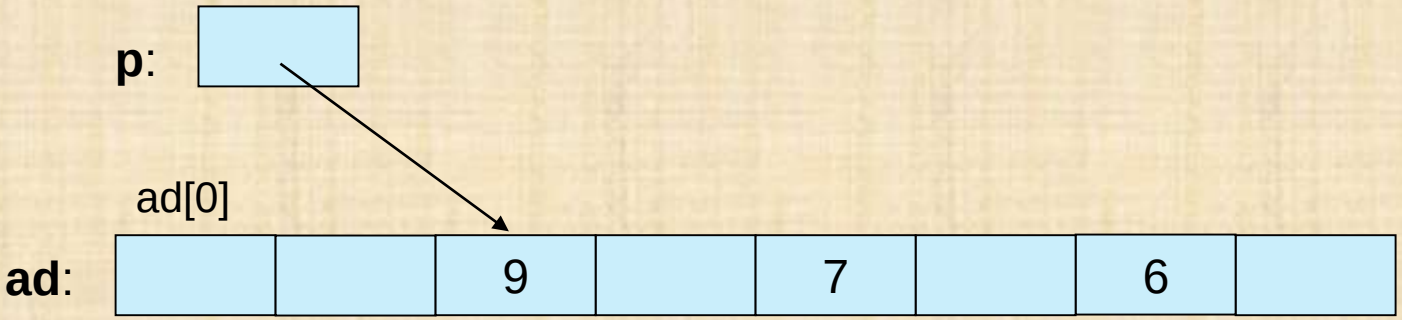
Pointer arithmetic

```
double ad[8];
double* p = &ad[4];          // point to ad[4]; the 5th element of ad

*p = 7;
p[2] = 6;
p[-2] = 9;

p+=2;
```



Pointer arithmetic

```
double ad[8];
double* p = &ad[4];        // point to ad[4]; the 5th element of ad

*p = 7;
p[2] = 6;
p[-2] = 9;

p += 2;

p -= 4;
```



- Clearly a powerful feature
  - Use rarely and with great care
  - Prefer the range-checked **vector** and **span**

Why bother with arrays and pointer arithmetic?

- They represent primitive memory in C++ programs
  - We need them to implement low-level features, such as memory managers (e.g., **new/delete**)
  - We need them to implement containers (e.g., **vector)**
- It's all that C has
  - In particular, C does not have **vector**
  - 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
- 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
  - Instead use types such as **vector**, **string**, and **span**

Pointers and references

- Think of a reference as an automatically dereferenced immutable pointer
  - or as an alternative name for an object.
- A reference must be initialized to refer to an object
  - and you cannot make it refer to some other object
- There is no “null reference””

pointer	reference	comment
int x = 10;	int x = 10;	
int* p = &x;	int& r = x;	&x to get a pointer
p = x;	r = &x;	type errors
*p = 7;	r = 7;	write to the object pointed/referred to
p = 7;	*r = 7;	type errors
int x2 = *p;	int x2 = r;	read the value of the object pointed/referred to
int* p2 = p;	int& r2 = r;	make p2 point to the object pointed to by p make r2 refer to the object referred to by r
p = nullptr;	r = “nullref”;	there is no nullref
p = &x2;		make p point to x2
	r = x2;	assign x2 to the object referred to by r

Pointer and reference arguments

- We have choices

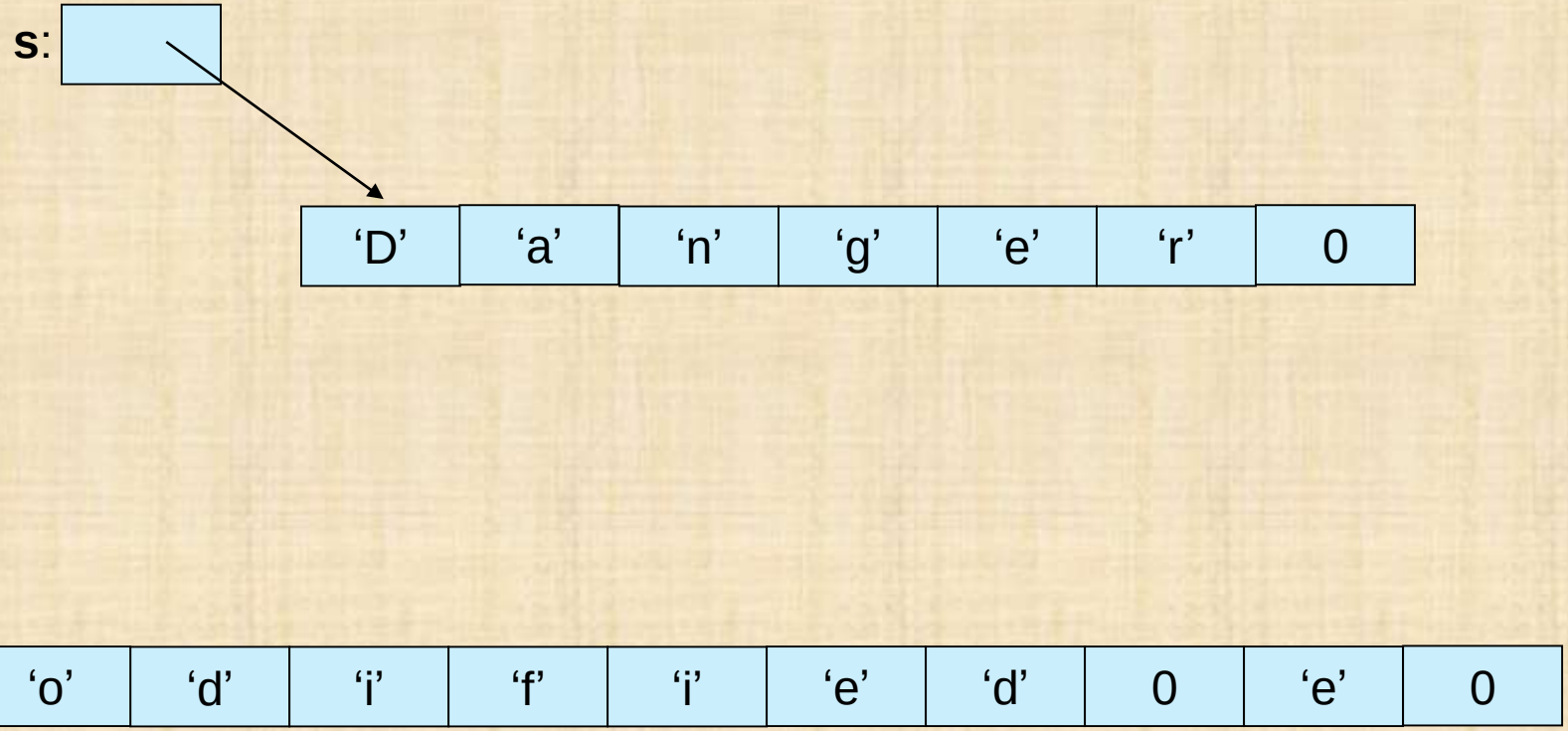
```
int incr_v(int x) { return x+1; }          // compute a new value and return it
void incr_p(int* p) { ++*p; }              // pass a pointer (dereference increment the result)
void incr_r(int& r) { ++r; }               // pass a reference
int incr_cr(const int& r) { return r+1; }// pass a reference to const
```

- Prefer the pass a value and return a result (here, **incr_v() or incr_cr()**)
  - Easier to read and don't change a value "behind our back"
- Use a pointer and non-const references only if want to change the value of an object
  - (here, **incr_p() or incr_r()**)
- Use a pointer only if "no object" is a valid argument
  - **incr_p(nullptr);** *// run-time error*
  - **incl_r(nullptr);** *// error*

C-style strings

- A C-style string is a pointer to a zero-terminated array of characters
- A C++ string literal is a C-style string of immutable characters

```
const char* s = "Danger!";
s[0] = ‘M’;          // error

char m[] = "Modifiable";
m[6] = ‘e’;
m[7] = ‘d’;
m[8] = 0;
```



C-style strings

- Using string leads to simpler and (therefore) less error-prone code
  - The C-style function has a potential memory leak

```
string cat(const string& name, const string& addr)
{
        return id + '@' + addr;
}
```

```
char* cat2(const char* name, const char* addr)
{
        int nsz = strlen(name);
        int sz = nsz+strlen(addr)+2;
        char* res = (char*) malloc(sz);
        strcpy(res,name);
        res[nsz+1] = '@';
        strcpy(res+2,addr);
        return res;
}
```

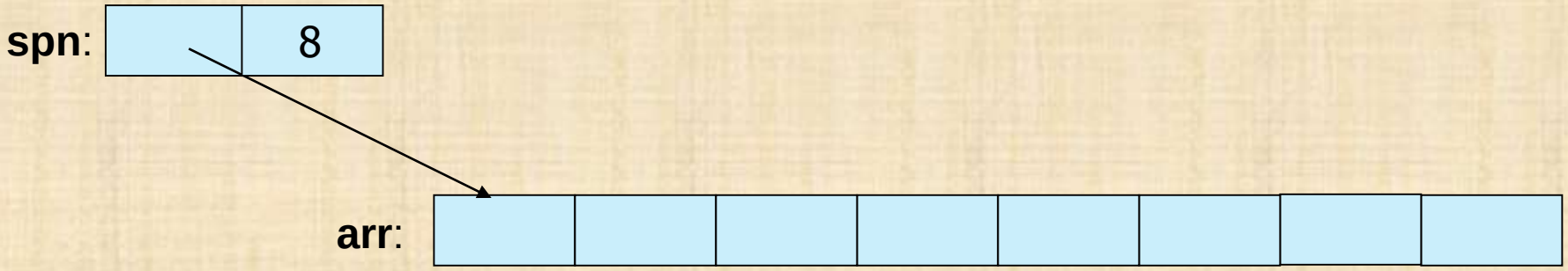
Alternatives to pointer use

- To hold a collection of values
  - use a standard-library container, such as **vector**, **set**, **map**, **unordered_map**, or **array**
- To hold a string of characters,
  - use the standard-library **string** (see also PPP2p 23.2).
- To point to an object, you own (i.e., must **delete**)
  - use the standard-library **unique_ptr** or **shared_ptr**
- To point to a contiguous sequence of elements that you don't own,
  - use the standard-library **span**
- To systematically avoid dereferencing a null pointer,
  - use **not_null**

span

- Think of a span as a pointer that “knows” how many elements it points to
  - And the PPP **span** does range checking

```
int arr[8];
span spn {arr};                    // a span<int> that points to 8 ints
```



span

- A **span** is simpler and safer to use than a pointer plus a size

```
int sum(span<int> s)
{
        int s = 0;
        for(int x : s)
                s += x;
        return s;
}

Int arr[100];
// …
int r = sum(arr);
```

```
int sum(int* p, int n)
{
        int s = 0;
        for(int I = 0; i<n; ++i)
                s += p[i];
        return s;
}
// what if p==nullptr?   What if n isn't the number of elements?

int arr[100];
// …
int r = sum(p,1000);      // Oops!
```

array

- **array** is a standard-library type with its size known to the compiler
  - Not the built-in array
  - An **array** does not implicitly convert to a pointer
  - Like a built-in array, an array can be statically allocated, put on the stack, and allocated on the free store

```
std::array<int,8> arr { 0,1,2,3,4,5,6,7 };
int* p = arr;                                   // error (and that's good)
int* q = arr.data();                            // if you must
```

not_null

- A not_null is an object that can be used as a built-in pointer
  - On initialization it is checked that it is not the **nullptr**

```
int strlen(const char* p)
{
        if (p==nullptr)
                return 0;
        int n = 0;
        while (*p++)
                ++n;
        return n;
}
```

- If we don't check, we might crash
- Failure to check for nullptr is a not uncommon bad bug

```
int strlen(not_null<const char*> p)
{
        int n = 0;
        while (*p++)
                ++n;
        return n;
}
```

- If we call with the nullptr, **not_null** throws a **not_null_error**

Palindromes – an example trying out techniques

- A palindrome is a word that's spelled the same forward and backwards
  - E.g., anna, ana, petep, and malayalam
- The outside-in algorithm using a **string**

```
bool is_palindrome(const string& s)
{
        int first = 0;                  // index of first letter
        int last = s.length()-1;        // index of last letter
        while (first < last) {          // we haven't reached the middle
                if (s[first]!=s[last])
                        return false;
                ++first;                // move forward
                --last;                 // move backward
        }
        return true;
}
```

Palindromes: string

- The standard-library string support makes this palindrome() easy to use

```
int main()
{
        for (string s; cin>>s; ) {
                cout << s << " is";
                if (!is_palindrome(s))
                        cout << " not";
                cout << " a palindrome\n";
        }
}
```

Palindromes: array

```
bool is_palindrome(const char s[], int n)
        // s points to the first character of an array of n characters
{
        int first = 0;                  // index of first letter
        int last = n-1;                 // index of last letter
        while (first < last) {          // we haven't reached the middle
                if (s[first]!=s[last])
                        return false;
                ++first;                // move forward
                --last;                 // move backward
        }
        return true;
}
```

Palindromes: array

```
istream& read_word(istream& is, char* buffer, int max)          // read at most max-1 characters
{
        is.width(max);          // read at most max-1 characters in the next >>
        is >> buffer;           // read whitespace-terminated word and add zero
        return is;
}
int main()
{
        constexpr int max = 128;
        for (char s[max]; read_word(cin,s,max); ) {
                cout << s << " is";
                if (!is_palindrome(s,strlen(s))) cout << " not";
                cout << " a palindrome\n";
        }
}
```

But what if there are more characters on input?

Palindromes: pointers

```
bool is_palindrome(const char* first, const char* last)
        // first points to the first letter, last to the last letter
{
        while (first < last) {                  // we haven't reached the middle
                if (*first!=*last)
                        return false;
                ++first;                        // move forward
                --last;                         // move backward
        }
        return true;
}
```

Elegant, but no checks.
Trust the caller

Palindromes: pointers

```
int main()
{
        const int max = 128;
        for (char s[max]; read_word(cin,s,max); ) {          // use read_word from "arrays"
                cout << s << " is";
                if (!is_palindrome(&s[0],&s[strlen(s)-1]))
                        cout << " not";
                cout << " a palindrome\n";
        }
}
```

Palindrome: recursive

```
bool is_palindrome_r(const char* first, const char* last)
        // first points to the first letter, last to the last letter
{
        if (first<last)
                return (*first==*last) ? is_palindrome_r(first+1,last-1) : false;
        return true;
}
```

Which version is better?
Iterative or recursive?
Measure!

Palindrome: span

```
bool is_palindrome(span<char> s)
{
        return (s.size()) ? is_palindrome(s.data(),s.data()+s.size()) : true; // implemented using pointers
}
```

- Often, we implement a function by calling others
  - Here, the version using **span** makes sure that the ranges is right for the pointer version

Programming

- Work at the highest level that you can afford
  - **vector**, **string**, **span**, …
- Use the low-level facilities and techniques to implement the highest levels
  - Pointers, arrays, **new**, **delete**, …

Next lecture

- The next lecture describes how **vector**s are copied and accessed through subscripting. To do that, we discuss copying in general and present the essential operations that must be considered for every type: construction, default construction, copy, move, and destruction.
- Like many types, **vector** offers comparisons, so we show how to provide operations such as **==** and **<**.
- Finally, we grapple with the problems of changing the size of a **vector**: why and how?