Chapter 17: Vector and Free Store

Plan for today

- We will talk about:
  - Pointers to class objects (17.7)
  - Pointer and reference parameters (17.9.1)
  - `void*` (17.8)
  - `this` pointer

Pointer and reference parameters

- When we want to change the value of a variable to a value computed by the function, we have three choices:
  - Compute a new value and return it:
  - Pass a pointer, dereference it and increment the result:
  - Pass a reference:

Pointer and reference parameters

- When we want to change the value of a variable to a value computed by the function, we have three choices:
  - Compute a new value and return it:

```
int f1(int a) { return a+1; }
```

  - Pass a pointer, dereference it and increment the result:

```
int f2(int* a) { return ++(*a); }
```

  - Pass a reference:

```
int f3(int& a) { return ++a; }
```

Pointer and reference parameters

- When we want to change the value of a variable to a value computed by the function, we have three choices:
  - Compute a new value and return it:

```
int f1(int a) { return a+1; }
```

```
int x=7;
x = f1(x);
```

  - Pass a pointer, dereference it and increment the result:

```
int f2(int* a) { return ++(*a); }
```

```
f2(&x);
```

  - Pass a reference:

```
int f3(int& a) { return ++a; }
```

```
f3(x);
```

Pointer and reference parameters

- When we want to change the value of a variable to a value computed by the function, we have three choices:
  - Compute a new value and return it:

```
int f1(int a) { return a+1; }
```

```
int x=7;
x = f1(x);
```

  - Pass a pointer, dereference it and increment the result:

```
int f2(int* a) { return ++(*a); }
```

```
f2(&x);
```

  - Pass a reference:

```
int f3(int& a) { return ++a; }
```

```
f3(x);
```

- How do we choose?

Pointer and reference parameters

- When we want to change the value of a variable to a value computed by the function, we have three choices:
  - Compute a new value and return it:

```
int f1(int a) { return a+1; }        int x=7;
                                      x = f1(x);
```

  - Pass a pointer, dereference it and increment the result:

```
int f2(int* a) { return ++(*a); }    f2(&x);
```

  - Pass a reference:

```
int f3(int& a) { return ++a; }       f3(x);
```

- How do we choose?
  - We prefer the first option for small objects; the 3rd option has a slight preference because we can see that x "will be changed"

Pointers to class objects

- The notion of pointer is general
  - We can point to just about anything we can place in memory

```
vector* myFuncion1(int newSize) {
    vector* p = new vector(newSize);
    // add values to p
    return p;
}

void myFunction2() {
    vector* q = myFunction(10);
    cout << *q << endl; // ok, we overloaded operator<<
    delete q; }
```

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: we 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

`this` pointer

- When we work with classes, the identified `this` is a pointer that points to the object for which the member function was called.
- Recall the get() method:

```
double get(int n) const  // access:read
{
   return elem[n];
};
```

`this` pointer

- When we work with classes, the identified `this` is a pointer that points to the object for which the member function was called.
- Recall the get() method:

```
double get(int n) const  // access:read
{
    return *this.elem[n];
};
```

`this` pointer

- When we work with classes, the identified `this` is a pointer that points to the object for which the member function was called.
- Recall the get() method:

```
double get(int n) const  // access:read
{
   return this→elem[n];
};
```

`this` pointer

- When we work with classes, the identified `this` is a pointer that points to the object for which the member function was called.
- Recall the get() method:

```
double get(int n) const  // access:read
{
   return this→elem[n];
};
```

In Python, we explicitly mention this “pointer” when defining a method: `def add(self, …)`, it is usually given name `self`.

`this` pointer

- Assume we would like to overload the assignment operator for the vector objects: `operator=`
- To allow chaining, v1 = v2 = v3, we need to return a reference to vector object:

```
vector& operator=(const vector& other)
{
    // some work on copying elements...
    return *this;
};
```

In-class practice

- (pointers to objects) Consider the following code fragment:

```
vector *c = new vector(5);
// some work on vector is done ...
int x = c → size(); // it's the same as *c.size()
// since c is a pointer, we need to dereference it
// to access the object, and then we can call
// object's public method
```

Put the line of code that would access vector's element at position 3.

In-class practice

- (pointers and reference parameters)
  - Define a function that takes three parameters: two integers, and one integer pointer. It should calculate the product and the difference of the two first integer parameters. The product should be returned, and the difference is “returned” to the caller via the integer pointer.

This is the test code you can use:

```
int x = 10, y = 12, p, d;
p = func(x,y,&d);
cout << “the product of 10 and 12 is “ << p
     << “ and their difference is” << d << endl;
```

In-class practice

- (pointers and reference parameters)
  - Define a function that takes three parameters: two integers, and one integer pointer. It should calculate the product and the difference of the two first integer parameters. The product should be returned, and the difference is “returned” to the caller via the integer pointer.

A follow-up question: what will change in the test code if the third parameter of the function will be integer passed by reference?

This is the test code you can use:

```
int x = 10, y = 12, p, d;
p = func(x,y,&d);
cout << “the product of 10 and 12 is “ << p
     << “ and their difference is” << d << endl;
```

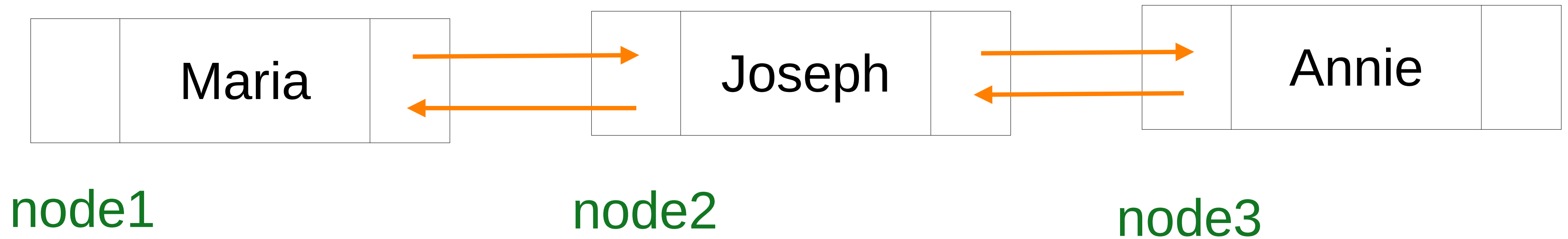
In-class practice

- Consider the following struct:

```
struct Link{
    string value;
    Link* prev;
    Link* succ;
    Link(const string& str, Link* p = nullptr,
            Link* s = nullptr):
        value{str}, prev{p}, succ{s}
    {}
};
```

In-class practice

– Let’s create the following connected list of those Links:



Resources used for these slides

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