if.cc

```
int main()
{
    int a {2};
    if ( a = 0 )
        cout << "A is zero\n";
    else
        cout << "Value of A is " << a << endl;
}</pre>
```

Correct answer is "Value of A is 0".

- 1. a has the value 2.
- 2. a is assigned the value o.
- 3. The return value of an assignment is a reference to the right hand side.
- 4. o is converted to false.
- 5. The printout comes from the else statement.

init.cc

```
int main()
{
    int i = 3.5;
    cout << i;
}</pre>
```

Correct answer is "3"

1. 3.5 is truncated to 3

init-2.cc

```
int main()
{
    int i (3.5);
    cout << i;
}</pre>
```

Correct answer is "3"

1. 3.5 is truncated to 3

init-3.cc

```
int main()
{
    int i {3.5};
    cout << i;
}</pre>
```

Correct answer is "doesn't compile"
1. Initialization with braces (list-initialization) forces the compiler
to check for narrowing conversion.

ref.cc

```
void fun(int const &){
    cout << 1;
void fun(int &){
    cout << 2;
void fun(int &&){
    cout << 3;
int main(){
   int a;
   int const c {};
    fun(23);
    fun(a);
    fun(c);
```

Correct answer is "321"

- 1. 23 is a pr-value and bind to r-value reference
- 2. a is an I-value, bind to I-value reference
- 3. c is const, bind to const-ref

ref-2.cc

```
struct T{};
void fun(T const &){
    cout << 1;
}

void fun(T &&){
    cout << 3;
}

int main(){
    T a;
    T const c {};
    fun(T{});
    fun(a);
    fun(c);
}</pre>
```

Correct answer is "311"

- 1. TO creates a temporary T object. Binds to r-value reference
- 2. a is an I-value, cannot bind to r-value reference adding const is better
 - 3. c bind to const-ref

templ.cc

```
template <typename T>
void foo(T) {
    cout << 1;
}

void foo(int const &) {
    cout << 2;
}

int main() {
    int a;
    int const b{};
    foo(a);
    foo(3);
    foo(b);
}</pre>
```

Correct answer is "222"				
The template can't be instantiated to get a perfect match for				
either a or 3. (a requires reference and 3 r-value reference for				

perfect match).

templ-2.cc

```
template <typename T>
void foo(T &) {
    cout << 1;
}

void foo(int const &) {
    cout << 2;
}

int main() {
    int a;
    int const b{};
    foo(a);
    foo(3);
    foo(b);
}</pre>
```

_		_		
Corre	∽t ⊃n	swer i	c ''1	22"

matches

1. The template function requires an I-value reference, only a

templ-3.cc

```
template <typename T>
void foo(T &b) {
    cout << 1;
}

void foo(int const &) {
    cout << 2;
}

int main() {
    int a;
    int const b{};
    foo(a);
    foo(3);
    foo(b);
}</pre>
```

Correct answer is "112"

- 1. b matches the normal function perfectly
- 2. T & is a forwarding reference.
 - ▶ First call gives T=int &, T&& \Rightarrow int &
 - ▶ Second call gives T=int, $T\&\& \Rightarrow int \&\&$

virt.cc

```
struct Base
    void fun() {
        cout << "Base::fun";</pre>
};
struct Derived: Base
   void fun() {
        cout << "Derived::fun";</pre>
};
void foo(Base const & b) {
    b.fun();
int main() {
    foo(Derived{});
```

Doesn't compile. b (in foo) is const and we call a non-const	t

function.

virt-2.cc

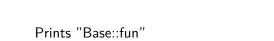
```
struct Base
    void fun() {
        cout << "Base::fun";</pre>
};
struct Derived: Base
   void fun() {
        cout << "Derived::fun";</pre>
};
void foo(Base & b) {
    b.fun();
int main() {
    foo(Derived{});
```

Doesn't compile. b (in foo) is reference and we pass a temporary

object.

virt-3.cc

```
struct Base
   void fun() {
        cout << "Base::fun";</pre>
};
struct Derived: Base
   void fun() {
        cout << "Derived::fun";</pre>
};
void foo(Base & b) {
    b.fun();
int main() {
    Derived d;
    foo(d);
```



1. b in foo is of type Base & and fun is non-virtual.

virt-4.cc

```
struct Base
   virtual void fun() {
        cout << "Base::fun";
};
struct Derived: Base
   virtual void fun() const {
        cout << "Derived::fun";</pre>
};
void foo(Base & b) {
    b.fun();
int main() {
    Derived d;
    foo(d);
```

Prints "Base::fun"

- 1. Derived::fun doesn't have the same signature as Base::fun and will not override.
- 2. virtual will Derived::fun as a new virtual function that can be overrridden in subsequent subclasses.

virt-5.cc

```
struct Base
   virtual void fun() const {
        cout << "Base::fun";
};
struct Derived: Base
   void fun() const override {
        cout << "Derived::fun";</pre>
};
void foo(Base const & b) {
    b.fun();
int main() {
    Derived d;
    foo(d);
```

Correct answer is "Derived::fun"

- 1. We finally have polymorphism!
- 2. Polymorphism requires two things;
 - ► A reference or pointer to base class
 - ▶ A call to a virtual function

unique.cc

```
int main()
{
   vector<int> vals {2,1,4,1};
   unique(begin(vals), end(vals));
   copy(begin(vals), end(vals), ostream_iterator<int>{cout, " "});
}
```

Correct answe	r is "2 1 4 1"			
1. unique wi	ll only "remove	e" duplicates th	nat occur in s	equence.

init-cls.cc

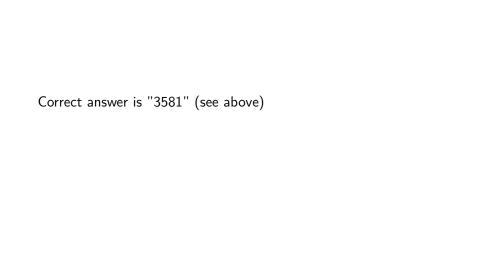
```
struct Base{
    Base(){
        cout << 3;
    ~Base(){
       cout << 1;
};
struct Derived{
    Derived(){
        cout << 5;
    ~Derived(){
        cout << 8;
};
int main(){
    Derived{};
```

Correct answer is "3581"

- 1. Initialization order:
 - 1.1 Initialize base class(es) (Base())
 - 1.2 Initialize data members (in declaration order)1.3 Run constructor (Derived())
 - 1.4 Run destructor (~Derived())
 - 1.5 Destroy data members (in reverse declaration order)
 - 1.6 Destroy base class (-Base())

init-member.cc

```
struct Data{
    Data(){
        cout << 3;
    ~Data(){
        cout << 1;
};
struct My_Class{
    My_Class(){
        cout << 5;
    ~My_Class(){
        cout << 8;
    Data a;
};
int main(){
    My_Class{};
```



sfinae.cc

```
template <typename T>
enable_if_t<is_integral<T>::value>
foo(T &&) {
    cout << 1;
}
int main()
{
    foo(2);
}</pre>
```

Correct answer is "1"

1. T=int, enable_if_t<...> \rightarrow void

sfinae-2.cc

```
stinae-2.cc

template <typename T>
enable_if_t<is_integral<T>::value>
foo(T &&) {
    cout << 1;
}

int main() {
    int i{};
    foo(i);
}</pre>
```

Doesn't compile.

1. T=int &, is_integral<T>:::value is false, enable_if_t<...> undefined \Rightarrow template is not instantiated, program ill-formed.

sfinae-3.cc

```
template <typename T>
enable_if_t<is_integral<decay_t<T>>::value>
foo(T &k) {
    cout << 1;
}
int main() {
    int i{};
    foo(i);
}</pre>
```

Correct answer is "1"

1. $_{\text{decay_t}}$ removes cv-qualification (and others) \Rightarrow valid again ($_{\text{T}}$ is still $_{\text{int \&}}$

str.cc

All compile, but only two gives a string with ten asterisks.

- 1. string{"*", 10} takes the ten first characters in the c-string literal "*" (undefined behavior).
- 2. string('*', 10) initializer-list constructor. Will get two characters, '*' and char{10} (newline)
- 3. string('*', 10) calls the correct constructor (repeat char a number of times), but order of arguments are wrong (gives int{'*'} number of char{10}).
 - 4. string{"*"s, 10} takes substring of string{"*"}, index 10 to the last.
- 5. string(10, '*') correct
- 6. string{"********* also correct

vartemp.cc

```
int foo(int, char)
{
    return 4;
}

template <typename Ret, typename Fun, typename ...Args>
Ret call(Fun f, Args... args)
{
    return f(args...);
}

int main()
{
    int a;
    char c;
    cout << call(foo, a, c);
}</pre>
```

Doesn't compile

1. The compiler is unable to deduce Ret

vartemp-2.cc

```
int foo(int, char)
{
    return 4;
}

template <typename Ret, typename Fun, typename ...Args>
Ret call(Fun f, Args... args)
{
    return f(args...);
}

int main()
{
    int a;
    char c;
    cout << call<int>(foo, 4, c);
}
```

Correct answer is "4"

1. 4 and c is passed by value to foo

vartemp-3.cc

```
int foo(int, char &)
{
    return 4;
}

template <typename Ret, typename Fun, typename ...Args>
Ret call(Fun f, Args... args)
{
    return f(args...);
}

int main()
{
    int a;
    char c;
    cout << call<int>(foo, 4, c);
}
```

Doesn't compile

bind to an I-value reference.

1. Arguments passed by value from call to foo. The char can't

vartemp-4.cc

```
int foo(int &&, char &)
{
    return 4;
}

template <typename Ret, typename Fun, typename ...Args>
Ret call(Fun f, Args... &&args)
{
    return f(forward<Args>(args)...);
}

int main()
{
    int a;
    char c;
    cout << call<int>(foo, 4, c);
}
```

Correct answer is "4"
1. The forwarding reference is forwarded to foo

vartemp-5.cc

```
int foo(int &&, char &)
{
    return 4;
}

template <typename Ret, typename Fun, typename ...Args>
Ret call(Fun f, Args... &&args)
{
    return f(forward<Args>(args)...);
}

int main()
{
    int a;
    char c;
    cout << call<int>(foo, a, c);
}
```

Doesn't compile.

1. a can't bind to the int && in foo.