Informatik - Exercise Session
Pointers and Dynamic Data Structures

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values
a	1

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values
a	1
b	2

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values
a	1
b	2
x	$\hookrightarrow$ a

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values
a	1
b	2
x	$\hookrightarrow$ a
y	$\hookrightarrow$ x

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values
a	1
b	2
x	$\hookrightarrow$ a
y	$\hookrightarrow$ a

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values	
a	1	
b	2	
x	$\hookrightarrow$ a	
y	$\hookrightarrow$ a	$\uparrow$ 2

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values	
a	$\cancel{1}$	2
b	2	
x	$\hookrightarrow$ a	
y	$\hookrightarrow$ a	

Recap: References

What is the output of the following program?

```
int a = 1;
int b = 2;
int& x = a;
int& y = x;
y = b;
assert(a == b);
std::cout << a << " " << b << " " << x << " " << y << std::
endl;
```

Variable	Values	
a	$\cancel{1}$	2
b	2	
x	$\hookrightarrow$ a	
y	$\hookrightarrow$ a	

And thus the output is: 2 2 2 2.

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *access the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Meanings of & and *

The symbol & can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *access the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol * can be used:

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *access the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol `*` can be used:

1. as the arithmetic multiplication operator (e.g. `z = x * y;`)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *access the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol `*` can be used:

1. as the arithmetic multiplication operator (e.g. `z = x * y;`)
2. to *declare* a pointer variable (e.g. `int *ptr_a = &a;`)

Meanings of & and *

The symbol `&` can disorient many people approaching C++. It is important to understand that this symbol has 3 different meanings in C++, depending on the position:

1. the logical AND operator (e.g. `z = x && y;`) (there is also bitwise AND which is however not covered by this course)
2. to *declare* a variable as a reference (e.g. `int& y = x;`)
3. to *access the address* of a variable (address operator) (eg. `int *ptr_a = &a;`)

Similarly, the symbol `*` can be used:

1. as the arithmetic multiplication operator (e.g. `z = x * y;`)
2. to *declare* a pointer variable (e.g. `int *ptr_a = &a;`)
3. to *access the content* of a variable via its pointer (dereference operator) (e.g. `int a = *ptr_a;`)

Example: Pointers

What happens in this snippet?

```
int a = 5;
int* x = &a;
*x = 6;
```

Example: Pointers

What happens in this snippet?

```
int a = 5;
int* x = &a;
*x = 6;
```

Variable	Values

Example: Pointers

What happens in this snippet?

```
int a = 5;
int* x = &a;
*x = 6;
```

Variable	Values
a	5

Example: Pointers

What happens in this snippet?

```
int a = 5;
int* x = &a;
*x = 6;
```

Variable	Values
a	5
x	$\hookrightarrow$ a

Example: Pointers

What happens in this snippet?

```
int a = 5;
int* x = &a;
*x = 6;
```

Variable	Values
a	5
x	$\hookrightarrow$ a $\uparrow$ 6

Example: Pointers

What happens in this snippet?

```
int a = 5;
int* x = &a;
*x = 6;
```

Variable	Values	
a	$\cancel{5}$	6
x	$\hookrightarrow$ a	

this pointer

Consider the following struct:

```
struct WeirdNumber {
    int number;

    void increment_by(int number) {
        (*this).number = (*this).number + number;
    }
};
```

this pointer

Consider the following struct:

```
struct WeirdNumber {
    int number;

    void increment_by(int number) {
        (*this).number = (*this).number + number;
    }
};
```

Whenever we implement a method (i.e. member function), the `this` pointer refers to the object we are currently *inside* of. It is unique to each object and only available inside methods.

Example: this pointer

An example with explanations:

```
#include <iostream>
int main() {
    WeirdNumber a = {42};
    WeirdNumber b = {-17};
    a.increment_by(3); // 'this' in the call of the increment_by function
                       // refers to the object a
    b.increment_by(2); // 'this' in the call of the increment_by function
                       // refers to the object b
    std::cout << a.number << ' ' << b.number << std::endl;
    return 0;
}
```

this->

To improve our notation with `(*this).var`, C++ introduces a convenient and intuitive shorthand: `this->var`.

this->

To improve our notation with `(*this).var`, C++ introduces a convenient and intuitive shorthand: `this->var`.

Another example: `*(*(*(*ptr1).ptr2).ptr3).ptr4` becomes `ptr1->ptr2->ptr3->ptr4`.

this->

To improve our notation with `(*this).var`, C++ introduces a convenient and intuitive shorthand: `this->var`.

Another example: `*(*(*(*ptr1).ptr2).ptr3).ptr4` becomes `ptr1->ptr2->ptr3->ptr4`.

An improve version of the `WeirdNumber struct`:

```
struct WeirdNumber {
    int number;

    void increment_by(int number) {
        this->number = this->number + number;
    }
};
```