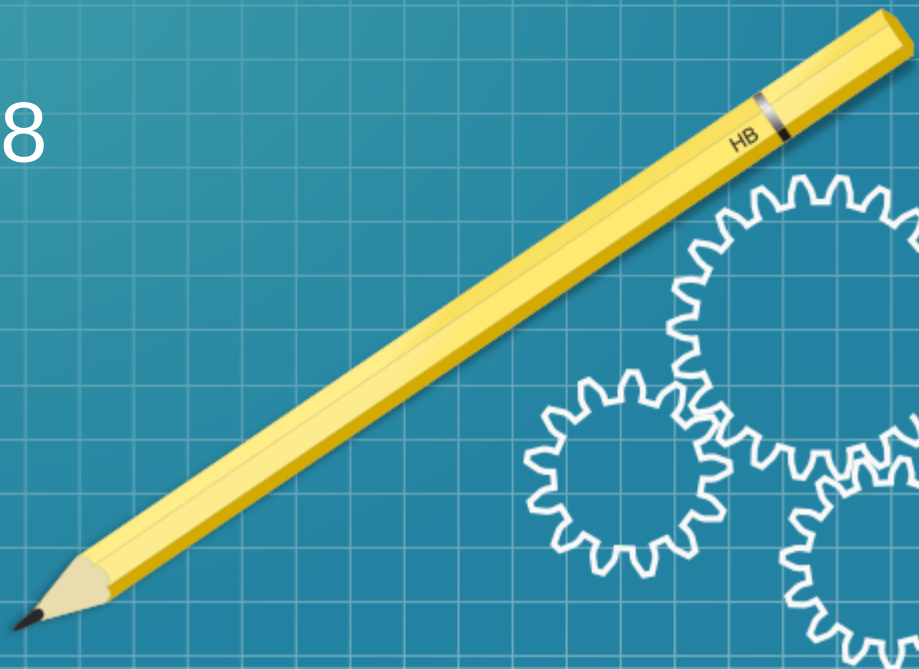
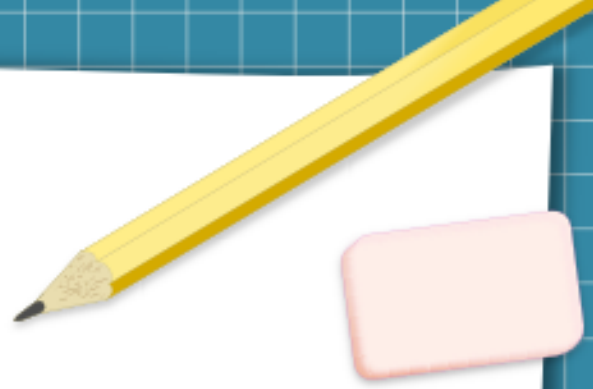
Pointers (continues)

Chapter 8



Today we will discuss:



- Passing pointers as parameters to functions

Precedence and associativity of some operators

postfix ++ and – – : *left to right*
prefix (unary) – – , ++, and * : *right to left*

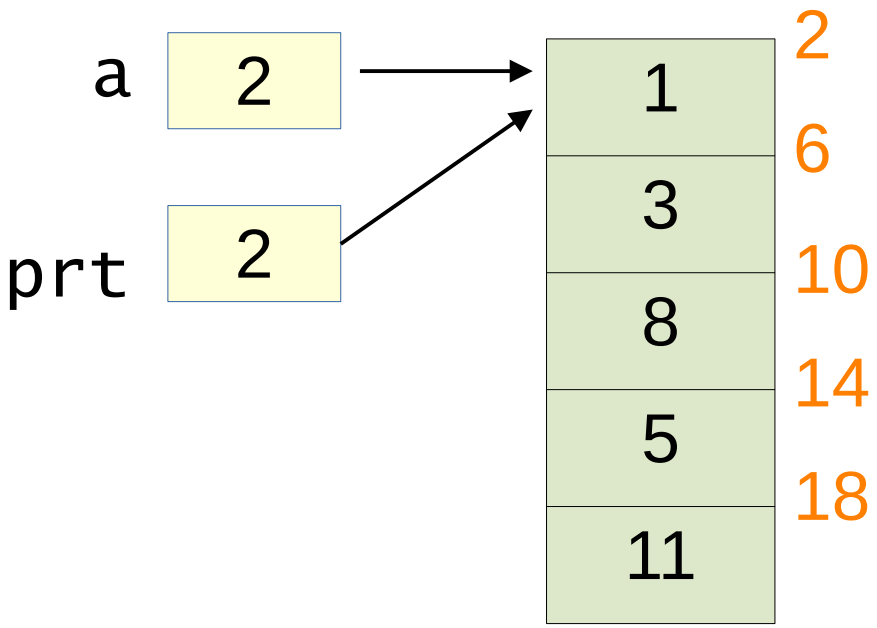
higher

lowe

Example:

```
int a[5] = {1,3,8,5,11};
int * ptr = a;

*ptr+1
*(ptr+1)
*++ptr
*ptr++
```



See operationsPrecedence.cpp

Passing pointers as parameters

There are three ways in C++ to pass arguments to functions:

- pass-by-value
- pass-by-reference with a reference argument
- pass-by-reference with a pointer argument

Passing pointers as parameters

There are three ways in C++ to pass arguments to functions:

- pass-by-value
- pass-by-reference with a reference argument
- pass-by-reference with a pointer argument

```
int add(int a, int b) { return a + b; }
int main() {
   int x = 10, y = 12;
   cout << add(x, y) << endl;
```

Passing pointers as parameters

There are three ways in C++ to pass arguments to functions:

- pass-by-value
- pass-by-reference with a reference argument
- pass-by-reference with a pointer argument

```
int add_r(int& a, int& b) { return a + b; }
int main() {
   int x = 10, y = 12;
   cout << add_r(x, y) << endl;
```

Passing pointers as parameters

There are three ways in C++ to pass arguments to functions:

- pass-by-value
- pass-by-reference with a reference argument
- pass-by-reference with a pointer argument

```
int add_p(int* a, int* b) { return *a + *b; }
int main() {
   int x = 10, y = 12;
   cout << add_p(&x, &y) << endl;
```

Note: pointers are passed by value

Passing pointers as parameters

```
int add(int a, int b) { return a + b; }

int add_r(int& a, int& b) { return a + b; }

int add_p(int* a, int* b) { return *a + *b; }


int main() {

   int x = 10, y = 12;

   cout << add(x, y) << endl;
   cout << add_r(x, y) << endl;
   cout << add_p(&x, &y) << endl;
}
```

see passingByReference.cpp

What can go wrong when *passing by reference*?

When we pass parameters *by reference*, with *reference* or with a *pointer* as arguments, we give *direct access* to the original/outside variable memory location.

Therefore, we we modify the value at that memory location, the change will be affecting the original/outside variable.
It is often called a *side effect*.

What can go wrong when *passing by reference*?

When we pass parameters *by reference*, with *reference* or with a *pointer* as arguments, we give *direct access* to the original/outside variable memory location.

Therefore, we we modify the value at that memory location, the change will be affecting the original/outside variable.
It is often called a *side effect*.

```
int add_r(int& a, int& b) {
   a = 5;
   return a + b; }

int main() {

int x = 10, y = 12;

cout << add_r(x, y) << endl;
```

What can go wrong when *passing by reference*?

When we pass parameters *by reference*, with *reference* or with a *pointer* as arguments, we give *direct access* to the original/outside variable memory location.

Therefore, we we modify the value at that memory location, the change will be affecting the original/outside variable.
It is often called a *side effect*.

```
int add_p(int* a, int* b) {
   *a = 6;
   return *a + *b; }

int main() {

int x = 10, y = 12;

cout << add_p(&x, &y) << endl;
```

Why do we need to *passing by reference*?

- to reduce the overhead time and space used for the copy of the value
- to overcome the limitation of "*a C++ function can return only one value*"

Why do we need to *passing by reference*?

- to reduce the overhead time and space used for the copy of the value

If we do not want the original values be modified:

```
int mult_r(const int& a, const int& b);
```

- to overcome the limitation of "*a C++ function can return only one value*"

Why do we need to *passing by reference*?

- to reduce the overhead time and space used for the copy of the value

If we do not want the original values be modified:

```
int mult_r(const int& a, const int& b);
int mult_p(const int* a, const int* b);
```

- *nonconstant pointer* (it can be modified) *to constant data*

(a's and b's "parent" values cannot be modified)

- to overcome the limitation of "*a C++ function can return only one value*"

Why do we need to *passing by reference*?

- to reduce the overhead time and space used for the copy of the value

If we do not want the original values be modified:

```
int mult_r(const int& a, const int& b);
int mult_p(const int* a, const int* b);
```

- *nonconstant pointer* (it can be modified) *to constant data*

(a's and b's "parent" values cannot be modified)

- to overcome the limitation of "*a C++ function can return only one value*"

```
double stats(int* min, int* max,
const int& a, const int& b, const int& c)
```

see passingByReference2.cpp

Passing pointer to a function

There are four ways to pass a pointer to a function:

- a non-constant pointer to non-constant data
  - data can be modified through the dereferenced pointer, and
  - pointer can be modified to point to other data
- a non-constant pointer to constant data
  - pointer can be modified to point to other data, but
  - data cannot be modified through the pointer
- a constant pointer to constant data
  - pointer always points to the same location, and
  - data at that location cannot be modified via the pointer
- a constant pointer to non-constant data
  - pointer always points to the same location, but
  - data at that location can be modified through the pointer

Passing pointer to a function

There are four ways to pass a pointer to a function:

- a non-constant pointer to non-constant data
  - data can be modified through the dereferenced pointer, and
  - pointer can be modified to point to other data

passingByReference3.cpp

- a non-constant pointer to constant data
  - pointer can be modified to point to other data
  - data cannot be modified through the pointer

passingByReference2.cpp

- a constant pointer to constant data
  - pointer always points to the same location, and
  - data at that location cannot be modified via the pointer

passingByReference3.cpp

- a constant pointer to non-constant data
  - pointer always points to the same location, but
  - data at that location can be modified through the pointer

`sizeof()` operator

`sizeof()` is a *compile time* unary operator that determines the size of any *data type*, *variable*, or *constant* in *bytes*.

sizeof() operator

sizeof() is a *compile time* unary operator that determines the size of any *data type*, *variable*, or *constant* in *bytes*.

consider the following code fragment:

```
char myCharArray[] = { 'a','b','c','d' };
int myIntArray[] = { 12, 16, 13, 18, 21,
23 };
double myDoubleArray[] = { 1.2, 0.16, -13, -
1.8, 2.1, -2.3, 0.9, 12.89, -9.8 };

sizeof(myCharArray)
sizeof(myCharArray) / sizeof(myCharArray[0])

sizeof(myIntArray)
sizeof(myIntArray) / sizeof(myIntArray[0])
...
```

Pointer assignment

A pointer can be assigned to another pointer if both pointers are of the *same type*.

A cast operator (will be discussed in Section 14.8) can be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment.

Pointer to `void` (`void*`) is a generic pointer capable of representing *any pointer type*.

Any pointer to a *fundamental type* or *class type* can be assigned to a pointer of type `void*` without casting, but not vice versa.

A `void*` pointer cannot be dereferenced, because the compiler does not know the type of the data the pointer refers to (i.e. the number of bytes is unknown).

HW assignment

1) posted on our web-site, from previous lecture
2) Exercise 8.12, see the draft posted on our web-site.

Suggested exercises
(*not for grade, but the questions related to these will appear on a quiz or a test*):
1) Chapter 8,
   Summary and all Self-Review Exercises
2) Chapter 8, Exercise: 8.13, 8.14