Outline

**Visual Studio**: start project, debug program

**Program Structure**: iostream, std, main, sequential, return, comment

**Integer Variable**: declare, assign, initialize, arithmetic, input/output, round, unsigned, constant.

**Array Variable**: declaration, index, initialization, multi-dimensional array, sparse matrix

**Instruction**: arithmetic/increment, relational/logical, bitwise, if-else, switch, while-loop, for-loop, break

**Other Variables**: float, char, string, enum, vector, structure

**Pointer Variable**: idea, declare, extract address, access content, allocation, relation to array

**Function**: definition, declaration, assignment passing, local/global variable, overload

**Class**: definition, public/private member, constructor/destructor, declaration, memory
Each segment has an address.

```c
int main()
{
    int x;
    x = 3;
    return 0;
}
```

A declared variable is in a memory segment.

Variable name is associated with that segment.

* Addresses here are made-up.

Segment size depends on variable type.

*e.g., x is associated with the 2nd segment.*
Pointer Variable

A pointer variable is used to hold address.

It offers another method to access data.

\[ \text{int main()} \{
\text{int x;}
\text{x = 3;}
\text{return 0;}
\} \]

Method 1: access segment through “x”.

When x is called, data in its associated segment will be retrieved or updated.

* Addresses here are made-up.
A pointer variable is used to hold address.

It offers another method to access data.

Method 2: access segment through address.

When the address is called, data at that address will be retrieved or updated.

```c
int main()
{
    int x;
    x = 3;
    return 0;
}
```
Declare a pointer and extract address.

```c
int main()
{
    int x = 3;
    int *p;
    p = &x;
    return 0;
}
```

- Declare a pointer to hold the address of an integer variable!
  - Syntax: `var_type *pointer_name;`

- Extract address of a variable and assign it to the pointer.
  - Syntax: `&var_name;`

<table>
<thead>
<tr>
<th>Memory</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>001</td>
</tr>
<tr>
<td>3</td>
<td>002</td>
</tr>
<tr>
<td></td>
<td>003</td>
</tr>
<tr>
<td></td>
<td>004</td>
</tr>
</tbody>
</table>
Example

```c
int main()
{
    int x = 3;
    int *p;
    p = &x;
    return 0;
}
```

Associate “x” with a segment and store value 3 in it.
Example

```c
int main()
{
    int x = 3;
    int *p;
    p = &x;
    return 0;
}
```

At the moment, this pointer is not pointing to any address.
Example

int main()
{
    int x = 3;
    int *p;
    p = &x;
    return 0;
}
Access content of a pointer.

```cpp
int main()
{
  int x = 3;
  int *p = &x;
  cout << *p;  // Access data at an address held by a pointer via *pointer_name;
  cout << p;  // here, *p = 3
  return 0;
}
```
Attention: Accessed Content vs Pointer.

```cpp
int main()
{
    int x = 3;

    int *p = &x;

    cout << *p; // print content at address p, so “3”.

    cout << p; // print address held by p, so “2”.

    return 0;
}
```
Random allocation of memory segment.

```c
int main()
{
    int x = 3;

    int *p = &x;
    cout << *p << '\n';

    cout << p;

    return 0;
}
```

Run program for the 1st time.

Run program for the 2nd time.

Segment is randomly allocated, thus address is random.
Relation between pointer and array.

```cpp
int main()
{
    int x[3] = {1, 2, 3};
    int *p;
    p = x;
    cout << p[1];
    return 0;
}
```

When we declare an array, consecutive segments will be allocated for it.

* Actual addresses will increment by the size of a variable.
Relation between pointer and array.

```cpp
int main()
{
    int x[3] = {1, 2, 3};
    int *p;
    p = x;
    cout << p[1];
    return 0;
}
```

When we declare an array, consecutive segments will be allocated for it.

Array elements will be stored in order
1st element “1” goes to 1st segment
2nd element “2” goes to 2nd segment

* Actual addresses will increment by the size of a variable.
Relation between pointer and array.

```cpp
int main()
{
    int x[3] = {1,2,3};

    int *p;

    p = x;  // When assigning an array to a pointer, the pointer will hold the address of the first element (equivalently, address of the head of the array).

    cout << p[1];

    return 0;
}
```
Relation between pointer and array.

```cpp
int main()
{
    int x[3] = {1, 2, 3};
    int *p;
    p = x;
    cout << p[1]; // Q: what is the output?
    return 0;
}
```

Then, we can access array elements using pointer, as if it is an array name.

<table>
<thead>
<tr>
<th>x</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>007</td>
</tr>
<tr>
<td>2</td>
<td>008</td>
</tr>
<tr>
<td>3</td>
<td>009</td>
</tr>
<tr>
<td></td>
<td>010</td>
</tr>
</tbody>
</table>

Q: what is the output?
Relation between pointer and array.

```cpp
int main()
{
    int x[3] = {1, 2, 3};
    int *p;
    p = x;
    cout << p[1];  // Output 2, as if we are outputting x[1].
    return 0;
}
```

Then, we can access array elements using pointer, as if it is an array name.
Two equivalent ways of assigning an array to a pointer.

```c
int main()
{
    int x[3] = {0,1,2};

    int *p;
    p = &x[0];
    p = x;

    return 0;
}
```

Address of the 1st array element.

These two are equivalent.
Pointer has no limit on index, but out-of-scope elements are random.

```cpp
int main()
{
    int x[3] = { 0, 1, 2 };
    int* p = x;
    cout << p[2];         // output 2
    cout << '\n' << p[3]; // output random
    cout << '\n' << p[4]; // output random
    return 0;
}
```
Pointer has no limit on index, but out-of-scope elements are random.

```cpp
int main()
{
    int x[3] = { 0, 1, 2 };
    int* p = x;
    cout << p[2];
    cout << '
' << p[3];
    cout << '
' << p[4];
    return 0;
}
```
Outline

Visual Studio: start project, debug program

Program Structure: iostream, std, main, sequential, return, comment

Integer Variable: declare, assign, initialize, arithmetic, input/output, round, unsigned, constant.

Array Variable: declaration, index, initialization, multi-dimensional array, sparse matrix

Instruction: arithmetic/increment, relational/logical, bitwise, if-else, switch, while-loop, for-loop, break

Other Variables: float, char, string, enum, vector, structure

Pointer Variable: idea, declare, extract address, access content, allocation, relation to array

Function: definition, declaration, assignment passing, local/global variable, overload

Class: definition, public/private member, constructor/destructor, declaration, memory
A function is often used to perform a specific task, e.g.,
- add two numbers
- multiply two matrices
- print title of a course
- update a variable
- retrieve a variable

A function called “Sum”.
It takes two integers as input; returns their sum.

```
int main()
{
    int x=1, y=3, z;

    int Sum(int a, int b);

    z = Sum(x, y);

    cout << z;

    return 0;
}
```

z = 4 -- *do you know why?*
Define a function.

Define input and output.

```c
int Sum(int a, int b) {
    int c;
    c = a + b;
    return c;
}
```

Before using a self-defined function, we need to define it.
Define input and output of a function.

```c
int Sum(int a, int b) {
    int c;
    c = a + b;
    return c;
}
```

- Define output type.
- Output is an integer.
- Return integer c as output of the function.
Define input and output of a function.

Define function name.
Make it indicative!

\[
\text{int Sum}(\text{int } a, \text{ int } b) \{
\text{int } c;
\text{c} = a + b;
\text{return } c;
\}
\]

* These names are only needed in definition, not in application.
Define input and output of a function.

```c
int Sum(int a, int b) {
    int c;
    c = a + b;
    return c;
}
```

Define input list. (type + name)

We have two inputs here:
1st: an integer named “a”.
2nd: an integer named “b”.
Define body of a function.

```c
int Sum(int a, int b) {
    int c;
    c = a + b;
    return c;
}
```

Define body of a function in a similar way as main.

That’s right: main function is also a function!

Declare an integer c.

Assign a + b to c.

No need to declare a & b in the body, as they are already declared as function inputs.

Return c as output.
Main function is also a function!

Output an integer. → \texttt{int main()}

Define function name. → \{ \texttt{int x=1, y=2, z;} \}

Program starts with a function named “main”.

No input. → \texttt{z = x + y;}

Need to declare every variable to be used.

Return integer 0. → \texttt{cout << z;}

\texttt{return 0;}

0 = success
Void function.

Some functions may not need to return any value, e.g.,
- a function to print input
- a function to print warning

Define these functions as “void”.

Means no return is expected.

```plaintext
int main()
{
    int x=1;
    Print(x);
    return 0;
}
```

```plaintext
void Print(int a) {
    cout << a;
}
```

No return instruction.

Print 1 on screen.
Declare a user-defined function.

To use a user-defined function in main, declare it before main. There are two approaches.

Method 1: place the function definition before main.

```c
int Sum(int a, int b) {
    int c;
    c = a + b;
    return c;
}
```

Directly call “Sum”.

```c
int main()
{
    int x=1, y=2, z;
    z = Sum(x, y);
    return 0;
}
```

Definition of Sum function.

Main function.
Declare a user-defined function.

To use a user-defined function in main, declare it before main. There are two approaches.

Method 1: place the function definition before main.

Method 2: place def after main, but declare it in main before use.

```c
int main()
{
    int x=1, y=2, z;
    int Sum(int a, int b);
    z = Sum(x, y);
    return 0;
}

int Sum(int a, int b) {
    int c;
    c = a + b;
    return c;
}
```

Syntax: same as function definition.
Different forms of function input.

Inputs are any valid expression

1. integer variables
   \[ z = \text{Sum}(x, y); \]
   \[ z = \text{Sum}(1, 2); \]
   \[ z = \text{Sum}(\text{Sum}(x, 7), y); \]

2. integers

3. function output

\[
\text{int} \ \text{main}() \{
\text{int} \ x=1, \ y=2, \ z;
\text{int} \ \text{Sum} (\text{int} \ a, \text{int} \ b);
\}
\]

Input names in definition and calls can be different.

Q: \( z = ? \)
Argument passing to a function.

Arguments can be passed to the function in two ways:

**By Value:** function creates a new variable initialized by the input, and play with it thereafter. The original input is not impacted.

**By Reference:** directly pass input into the function and play with it. Input can be modified during call.

```cpp
int Increment(int a) {
    a += 1;
    return a;
}

int main() {
    int x=1;
    cout << Increment(x);
    return 0;
}
```
Argument passing to a function.

Arguments can be passed to the function in two ways:

**By Value:** function creates a new variable initialized by the input, and play with it thereafter. The original input is not impacted.

**By Reference:** directly pass input into the function and play with it. Input can be modified during call.

```cpp
int Increment(int a) {
    a += 1;
    return a;
}

int main()
{
    int x=1;
    cout << Increment(x);
    return 0;
}
```
Example of argument passing by value.

```c
int Increment(int a) {
    a += 1;
    return a;
}

int main()
{
    int x=1;
    cout << Increment(x);
    return 0;
}
```

If we define input as a regular variable, it will be passed by value.

Define a function that returns an incremented value.

Call the function.
Example of argument passing by value.

```cpp
int Increment(int a) {
    a += 1;  // x = 1
    return a;
}

int main() {
    int x = 1;
    cout << Increment(x);
    return 0;
}
```
Example of argument passing **by value**.

```cpp
int Increment(int a) {
    a += 1;
    return a;
}

int main() {
    int x = 1;
    cout << Increment(x);
    return 0;
}
```
Example of argument passing by value.

1. Start function.
2. Declare an integer a.
3. Initialize a by input x.

```cpp
int main()
{
    int x = 1;
    cout << Increment(x);
    return 0;
}

int Increment(int a)
{
    a += 1;
    return a;
}
```

Memory:
- x = 1
- x = 1, a = 1

* x and a are two different vars.
Example of argument passing by value.

```cpp
int Increment(int a) {
    a += 1;
    return a;
}

int main()
{
    int x=1;
    cout << Increment(x);
    return 0;
}
```

Memory:

- `x = 1`
- `x = 1, a = 1`
- `x = 1, a = 2`

* `x` is untouched in the function.

Increment `a` by 1.
Example of argument passing by value.

```cpp
int Increment(int a) {
    a += 1;
    return a;
}

int main()
{
    int x=1;
    cout << Increment(x);
    return 0;
}
```

Return value of `a`. Function will return 2.

Memory:
- `x = 1`
- `x = 1, a = 1`
- `x = 1, a = 2`
Example of argument passing by value.

```cpp
class Example {
public:
    int Increment(int a) {
        a += 1;
        return a;
    }

    int main() {
        int x = 1;
        cout << Increment(x); // Demo code.
        return 0;
    }
};

int main() {
    int x = 1;
    cout << Increment(x);
    return 0;
}
```

Print value 2.

x remains 1 during call.
Argument passing to a function.

Arguments can be passed to the function in two ways:

**By Value**: function creates a new variable initialized by the input, and play with it thereafter. The original input is not impacted.

**By Reference**: directly pass input into the function and play with it. Input can be modified during call.

```c
int Increment(int a) {
    a += 1;
    return a;
}

int main() {
    int x=1;
    cout << Increment(x);
    return 0;
}
```
Example of argument passing by reference.

Define a function that implements increment.

```c
int Increment(int &a) {
    a += 1;
    return a;
}
```

Syntax: (type &name).

Means input will be passed by reference and directly manipulated.

Call the function.

```c
int main()
{
    int x = 1;
    cout << Increment(x);
    return 0;
}
```
Example of argument passing by reference.

Declare an integer \( x = 1 \).

```cpp
int Increment(int &a) {
    a += 1;
    return a;
}

int main()
{
    int x = 1;
    cout << Increment(x);
    return 0;
}
```

Memory

\( x = 1 \)
Example of argument passing by reference.

```cpp
int Increment(int &a) {
    a += 1;
    return a;
}

int main()
{
    int x = 1;
    cout << Increment(x);
    return 0;
}
```

Call the function.
Example of argument passing by reference.

1. Start function.
   
   ```
   int Increment(int &a) {
     a += 1;
     return a;
   }
   ```

2. Claim “a” is input by reference.

3. “a” is equivalent to “x”
   
   ```
   int main() {
     int x = 1;
     cout << Increment(x);
     return 0;
   }
   ```

   Memory
   
   x = 1
   a = x = 1

   * a and x refer to the value at the same address in memory.
Example of argument passing by reference.

```cpp
int Increment(int &a) {
    a += 1;
    return a;
}

int main()
{
    int x = 1;
    cout << Increment(x);
    return 0;
}
```

Memory

```
x = 1
a = x = 1
a = x = 2
```

* a and x are both incremented.
Example of argument passing by reference.

```cpp
int Increment(int &a) {
    a += 1;
    return a;
}

int main()
{
    int x = 1;
    cout << Increment(x);
    return 0;
}
```

Return value of “a”. Function will return 2.
Example of argument passing by reference.

```cpp
int Increment(int &a) {
    a += 1;
    return a;
}

int main()
{
    int x = 1;
    cout << Increment(x);
    return 0;
}
```

Print value 2.
Passing array to a function.

We can input an array to a function through a pointer. Same as pass by reference. We can assign an array to a pointer and access array elements using the pointer, as if it is an array.

```c
void Increment(int *a) {
    a[0] += 1;
    a[1]++;
}

int main() {
    int x[2] = { 1, 2 };
    Increment(x);
    cout << x[0] << x[1];
    return 0;
}
```
Passing array to a function.

```c
void Increment(int *a) {
    a[0] += 1;
    a[1]++;
}

int main() {
    int x[2] = { 1, 2 };
    Increment(x);
    cout << x[0] << x[1];
    return 0;
}
```

Declare `x` as an array.
Passing array to a function.

```cpp
void Increment(int *a) {
    a[0] += 1;
    a[1]++;
}

int main()
{
    int x[2] = { 1, 2 };
    Increment(x);
    cout << x[0] << x[1];
    return 0;
}
```

Call function.
Passing array to a function.

1. Declare a pointer `a`.

```cpp
void Increment(int *a) {
    a[0] += 1;
    a[1]++;
}
```

2. Let “a” point to the same address as `x`.

```cpp
int main()
{
    int x[2] = { 1, 2 };
    Increment(x);
    cout << x[0] << x[1];
    return 0;
}
```
Passing array to a function.

Increment content by 1.

Note that this content is not only pointed by \texttt{a[0]}, but also pointed by \texttt{x[0]}.

So \texttt{x[0]} is also incremented.
Passing array to a function.

```c
void Increment(int *a) {
    a[0] += 1;
    a[1]++;
}

int main()
{
    int x[2] = { 1, 2 };
    Increment(x);
    cout << x[0] << x[1];
    return 0;
}
```

Increment content by 1.

Note that this content is not only pointed by a[1], but also pointed by x[1].

So x[1] is also incremented.
Passing array to a function.

```c
void Increment(int *a) {
    a[0] += 1;
    a[1]++;
}

int main() {
    int x[2] = { 1, 2 };
    Increment(x);
    cout << x[0] << x[1];
    return 0;
}
```

Print 2 and 3.
Passing array to a function.

Another way to pass array is to declare array as input.

Same as pass by reference.

```cpp
#include <iostream>

void Increment(int a[2]) {
    a[0] += 1;
    a[1]++;
}

int main()
{
    int x[2] = { 1, 2 };
    Increment(x);
    std::cout << x[0] << x[1];
    return 0;
}
```
Scope of a variable: local variable.

Normally, a variable declared in a function only exists in that function.

We call this a local variable.

```cpp
void Print() {
    int x = 3;
    cout << x;
}

int main() {
    int y = 1;
    Print();
    return 0;
}
```

x is declared inside Print(), so only exists inside Print() but not main().
Example of a local variable.

Normally, a variable declared in a function only exists in that function. We call this a local variable.

```c
int main()
{
    int y = 1;
    Print();
    return 0;
}

void Print()
{
    int x = 3;
    cout << x;
}
```

Declare `y`. 

Memory

```c
int y = 1
```
Example of a local variable.

Normally, a variable declared in a function only exists in that function.

We call this a local variable.

```c
int main()
{
    int y = 1;
    Print();
    return 0;
}
```

```c
void Print()
{
    int x = 3;
    cout << x;
}
```
Example of a local variable.

Normally, a variable declared in a function only exists in that function.

We call this a local variable.
Example of a local variable.

Normally, a variable declared in a function only exists in that function. We call this a local variable.

Memory
y = 1, y = 1, x = 3

As long as Print is running, x is in memory.
Example of a local variable.

Normally, a variable declared in a function only exists in that function. We call this a local variable.

```
when Print is done, x is removed from memory.
```

```
int x = 3;
int y = 1;
Print();
return 0;
```
Scope of a variable: local variable.

Normally, a variable declared in a function only exists in that function.

Normally, a function cannot directly access a variable declared in another.

```cpp
void Print() {
    int x = 3;
    cout << x + y;
}

int main() {
    int y = 1;
    Print();
    return 0;
}
```

This line is wrong!
y is declared in main, so it cannot be directly accessed by the Print function.

But remember we can still pass y to Print as argument.
Scope of a variable: global variable.

Normally, a variable declared in a function only exists in that function.

Normally, a function cannot directly access a variable declared in another.

A variable declared outside functions is a global variable. It can be directly accessed by other functions.

```c
int y = 1;

void Print() {
    int x = 3;
    cout << x + y;
}

int main() {
    Print();
    cout << y;
    return 0;
}
```

- y is declared outside Print() and main(). It is a global var.
- Print() can access y. Output 3 + 1 = 4.
- main() can access y. Output 1.
Local variable vs Global variable

Normally, a variable declared in a function only exists in that function.

Normally, a function cannot directly access a variable declared in another.

A variable declared outside functions is a global variable. It can be directly accessed by other functions.

If a global var and a local var have the same name, local var is accessed.
Local variable vs Global variable

Normally, a variable declared in a function only exists in that function.

Normally, a function cannot directly access a variable declared in another.

A variable declared outside functions is a global variable. It can be directly accessed by other functions.

If a global var and a local var have the same name, local var is accessed.

```cpp
int x = 1;               // A global variable named x.

void Print() {
    cout << x;           // No conflict. x is global. Output 1.
}

int main()
{
    int x = 5;           // A local variable also named x.
    cout << x;           // Conflict! Is this x the global one or the local one? Output 1 or 5?
    Print();
    return 0;
}
```

Local is protected. So output 5!