

Astroinformatics II (Semester 2 2024)

Introduction to C/C++ (I)

Nina Hernitschek

Centro de Astronomía CITEVA
Universidad de Antofagasta

October 29, 2024

Motivation

We have seen how to write simple programs in C++.

We will extend this to **more complex programs**.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Motivation

We have seen how to write simple programs in C++.

We will extend this to **more complex programs**.

But before doing so, we will take a look at how to **systematically find errors** in C++ code.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Semantic Errors

Often it happens that we write a program, it compiles fine, but if we run it, it is not working correctly.

So despite all the **syntax** is correct (checked by the compiler), we must have a **semantic** error somewhere.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Semantic Errors

Often it happens that we write a program, it compiles fine, but if we run it, it is not working correctly.

So despite all the **syntax** is correct (checked by the compiler), we must have a **semantic** error somewhere.



How to find it?

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

The Cause of Bugs

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

All **software bugs** stem from a simple premise: Something that you thought was correct, in fact isn't.

It might have been an error in the way you wrote a concept, equation or generally algorithm into code. It also might be an error in that concept, equation or algorithm itself.

The Cause of Bugs

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

All **software bugs** stem from a simple premise: Something that you thought was correct, in fact isn't.

It might have been an error in the way you wrote a concept, equation or generally algorithm into code. It also might be an error in that concept, equation or algorithm itself.

Actually figuring out where that error is can be challenging. In this lesson, we'll outline the general process of debugging a program.

A General Approach to Debugging

Once we have identified that our code is not working as intended, **debugging** the problem generally consists of six steps:

1. Find the root cause of the problem (usually the line of code that is not working).
2. Ensure you understand why the issue is occurring.
3. Determine how you'll fix the issue.
4. Repair the issue causing the problem.
5. Retest to ensure the problem has been fixed.
6. Retest to ensure no new problems have emerged.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

A Debugging Example

Now let's apply this process to a simple **example** with code:

```
1  #include <iostream>
2  using namespace std;
3
4  // this function is supposed to perform addition
5  int add(int x, int y)
6  {
7      return x - y;
8  }
9
10 int main()
11 {
12     // should produce 7, but produces 1
13     cout << "4 + 3 = " << add(4, 3) << '\n';
14     return 0;
15 }
```

The wrong answer gets printed to the screen via line 13. That gives us a starting point for our investigation.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

A Debugging Example

Now let's apply this process to a simple **example** with code:

```
1  #include <iostream>
2  using namespace std;
3
4  // this function is supposed to perform addition
5  int add(int x, int y)
6  {
7      return x - y;
8  }
9
10 int main()
11 {
12     // should produce 7, but produces 1
13     cout << "4 + 3 = " << add(4, 3) << '\n';
14     return 0;
15 }
```

1. Find the root cause: On line 13, we can see that we're passing in correct literals for arguments (4 and 3). As the inputs to add are correct, but the output isn't, add must be producing the wrong value. The only statement in function add is the return statement, we've found the problem line.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

A Debugging Example

Now let's apply this process to a simple **example** with code:

```
1  #include <iostream>
2  using namespace std;
3
4  // this function is supposed to perform addition
5  int add(int x, int y)
6  {
7      return x - y;
8  }
9
10 int main()
11 {
12     // should produce 7, but produces 1
13     cout << "4 + 3 = " << add(4, 3) << '\n';
14     return 0;
15 }
```

2. Understand the problem: In this case, it's obvious why the wrong value is being generated: we're using the wrong operator.
3. Determine a fix: We'll simply change operator - to operator +.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

A Debugging Example

Now let's apply this process to a simple **example** with code:

```
1  #include <iostream>
2  using namespace std;
3
4  // this function is supposed to perform addition
5  int add(int x, int y)
6  {
7      return x + y;
8  }
9
10 int main()
11 {
12     // should produce 7, but produces 1
13     cout << "4 + 3 = " << add(4, 3) << '\n';
14     return 0;
15 }
```

4. Repair the issue: This is actually changing operator - to operator + and ensuring the program recompiles.

5. Retest: After implementing the change, rerunning the program will indicate that our program now produces the correct value of 7.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

A Debugging Example

Now let's apply this process to a simple **example** with code:

```
1  #include <iostream>
2  using namespace std;
3
4  // this function is supposed to perform addition
5  int add(int x, int y)
6  {
7      return x + y;
8  }
9
10 int main()
11 {
12     // should produce 7, but produces 1
13     cout << "4 + 3 = " << add(4, 3) << '\n';
14     return 0;
15 }
```

This example is trivial, but **illustrates** the basic process you'll go through when diagnosing any program.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

In the previous example, it was quite easy to spot the error. Usually it is, however, more difficult.

We will see here some **tactics** for making better guesses what and where the problem is to by collecting information to help find errors in our code.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

Debugging tactic #1: Commenting out your code

If your program is showing erroneous behavior, one way to reduce the amount of code you have to search through is to comment some code out and see if the issue persists. If the issue remains unchanged, the commented out code probably wasn't responsible.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

Debugging tactic #1: Commenting out your code

Consider the following code **example**:

```
1  int main()
2  {
3      getNames(); // ask user to enter a bunch of names
4      doMaintenance(); // do some random stuff
5      sortNames(); // sort them in alphabetical order
6      printNames(); // print the sorted list of names
7
8      return 0;
9  }
```

The program is supposed to print the names the user enters in alphabetical order, but we find it is printing them in reverse alphabetical order. Where's the problem?

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

Debugging tactic #1: Commenting out your code

```
1  int main()
2  {
3      getNames(); // ask user to enter a bunch of names
4      //doMaintenance(); // do some random stuff
5      sortNames(); // sort them in alphabetical order
6      printNames(); // print the sorted list of names
7
8      return 0;
9  }
```

We suppose that `doMaintenance()` has nothing to do with the problem. We start with commenting it out.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

Debugging tactic #1: Commenting out your code

```
1  int main()
2  {
3      getNames(); // ask user to enter a bunch of names
4      //doMaintenance(); // do some random stuff
5      sortNames(); // sort them in alphabetical order
6      printNames(); // print the sorted list of names
7
8      return 0;
9  }
```

We suppose that `doMaintenance()` has nothing to do with the problem. We start with commenting it out.

Warning: Don't forget which functions you've commented out so you can uncomment them later!

After making many debugging-related changes, it's very easy to miss undoing one or two. If that happens, you'll end up fixing one bug but introducing others.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

There are three likely outcomes:

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

There are three likely outcomes:

If commenting out `doMaintenance` makes the problem disappear, then `doMaintenance` must be causing the problem, and we should focus our attention there.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

There are three likely outcomes:

If commenting out `doMaintenance` makes the problem disappear, then `doMaintenance` must be causing the problem, and we should focus our attention there.

If the problem is unchanged (which is likely), we can reasonably assume that `doMaintenance` is not causing the problem. We can exclude the entire function from our search for now. We don't know yet where the problem is but have reduced the amount of code we have to look through.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

There are three likely outcomes:

If commenting out `doMaintenance` makes the problem disappear, then `doMaintenance` must be causing the problem, and we should focus our attention there.

If the problem is unchanged (which is likely), we can reasonably assume that `doMaintenance` is not causing the problem. We can exclude the entire function from our search for now. We don't know yet where the problem is but have reduced the amount of code we have to look through.

If commenting out `doMaintenance` changes the problem to some other related problem (e.g. the program stops printing names), then it's likely that `doMaintenance` is doing something other code depends on. We then probably can't tell whether the issue is in `doMaintenance` or elsewhere, so we can uncomment `doMaintenance` and try some other approach.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

There are three likely outcomes:

If commenting out `doMaintenance` makes the problem disappear, then `doMaintenance` must be causing the problem, and we should focus our attention there.

If the problem is unchanged (which is likely), we can reasonably assume that `doMaintenance` is not causing the problem. We can exclude the entire function from our search for now. We don't know yet where the problem is but have reduced the amount of code we have to look through.

If commenting out `doMaintenance` changes the problem to some other related problem (e.g. the program stops printing names), then it's likely that `doMaintenance` is doing something other code depends on. We then probably can't tell whether the issue is in `doMaintenance` or elsewhere, so we can uncomment `doMaintenance` and try some other approach.

Having a good **version control system** is extremely useful here, as you can diff your code against the main branch to see all the debugging-related changes you've made (and ensure that they are reverted before you commit your change).

Motivation

Debugging

File I/O

Complex Data Structures

Functions

References and Pointers

Summary & Outlook

Basic Debugging Tactics

Debugging tactic #2: Validating your code flow

Another problem common in more complex programs is that the program is calling a function too many or too few times (including not at all).

In such cases, it can be helpful to place statements at the top of your functions to print the function's name. That way, when the program runs, you can see which functions are getting called.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

Debugging tactic #2: Validating your code flow

Another problem common in more complex programs is that the program is calling a function too many or too few times (including not at all).

In such cases, it can be helpful to place statements at the top of your functions to print the function's name. That way, when the program runs, you can see which functions are getting called.

Good to know:

When printing information for debugging purposes, use `std::cerr` instead of `std::cout`.

One reason for this is that `std::cout` may be buffered, which means there may be a pause between when you ask `std::cout` to output information and when it actually does. If you output using `std::cout` and then your program crashes immediately afterward, `std::cout` may or may not have actually output yet. This can mislead you about where the issue is. In contrast, `std::cerr` is unbuffered, which means anything you send to it will output immediately. This helps ensure all debug output appears as soon as possible (at the cost of some performance, which we usually don't care about when debugging).

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Basic Debugging Tactics

Debugging tactic #3: Printing values

With some types of bugs, the program may be calculating or passing the wrong value.

We can also output the value of variables (including parameters) or expressions to ensure that they are correct.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Using a Debugger

We started exploring how to **manually debug** problems. There are, however, some negative side effects of using statements to print debug text:

- Debug statements clutter your code.
- Debug statements clutter the output of your program.
- Debug statements require modification of your code to both add and to remove, which can introduce new bugs.
- Debug statements must be removed after you're done with them, which makes them non-reusable.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Using a Debugger

We started exploring how to **manually debug** problems. There are, however, some negative side effects of using statements to print debug text:

- Debug statements clutter your code.
- Debug statements clutter the output of your program.
- Debug statements require modification of your code to both add and to remove, which can introduce new bugs.
- Debug statements must be removed after you're done with them, which makes them non-reusable.



We can mitigate some of these issues if we use a **debugger** instead.

Motivation

Debugging

File I/O

Complex Data Structures

Functions

References and Pointers

Summary & Outlook

Using a Debugger

A **debugger** is a computer program that allows to control how another program executes code. While doing so, the program state can be examined.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Using a Debugger

A **debugger** is a computer program that allows to control how another program executes code. While doing so, the program state can be examined.

For example, a debugger can be used to execute a program line by line, examining the value of variables along the way. By comparing the actual value of variables to what is expected, or watching the path of execution through the code, the debugger can help immensely in tracking down semantic (logic) errors.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Using a Debugger

A **debugger** is a computer program that allows to control how another program executes code. While doing so, the program state can be examined.

For example, a debugger can be used to execute a program line by line, examining the value of variables along the way. By comparing the actual value of variables to what is expected, or watching the path of execution through the code, the debugger can help immensely in tracking down semantic (logic) errors.

The power behind the debugger is twofold: the ability to precisely **control execution** of the program, and the ability to **view** (and modify, if desired) the program's state.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Using a Debugger

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

A **debugger** is a computer program that allows to control how another program executes code. While doing so, the program state can be examined.

For example, a debugger can be used to execute a program line by line, examining the value of variables along the way. By comparing the actual value of variables to what is expected, or watching the path of execution through the code, the debugger can help immensely in tracking down semantic (logic) errors.

The power behind the debugger is twofold: the ability to precisely **control execution** of the program, and the ability to **view** (and modify, if desired) the program's state.

It is recommended to use a **IDE (Integrated Development Environment)** that can handle debugging more comfortable.

Example: VS Code, Code::Blocks.

GNU Debugger (GDB)

Initially, debuggers (such as GDB) were separate programs with command-line interfaces.

These days, usually an IDEs with an integrated debugger is used, making debugging with a graphical user interface possible in the same environment that you use to write your code.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

GNU Debugger (GDB)

Initially, debuggers (such as GDB) were separate programs with command-line interfaces.

These days, usually an IDEs with an integrated debugger is used, making debugging with a graphical user interface possible in the same environment that you use to write your code.

While integrated debuggers are highly convenient and recommended for beginners, command line debuggers are well supported and still commonly used in environments that do not support graphical interfaces (e.g. embedded systems, computing clusters without GUI access).

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Using an IDE

To use VS Code with C++, you must do the following:

1. Ensure GCC is installed

Although you'll use VS Code to edit your source code, you'll compile the source code on Linux using the g++ compiler. You'll also use GDB to debug. These tools are not installed by default on Ubuntu, so you have to install them. Fortunately, that's easy.

First, check to see whether g++ is already installed. To verify whether it is, open a terminal window and enter the following command:

```
g++ -v
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Using an IDE

To use VS Code with C++, you must do the following:

1. Ensure GCC is installed

Although you'll use VS Code to edit your source code, you'll compile the source code on Linux using the g++ compiler. You'll also use GDB to debug. These tools are not installed by default on Ubuntu, so you have to install them. Fortunately, that's easy.

First, check to see whether g++ is already installed. To verify whether it is, open a terminal window and enter the following command:

```
g++ -v
```

2. Install Visual Studio Code

With Linux, you can use a package manager like Discover for installing VS Code (often also found as "Code").

It is also available for download at

<https://code.visualstudio.com/download>

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

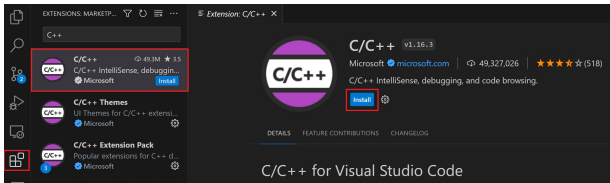
Summary &
Outlook

Using a IDE

3. Install the C++ extension

Once you have VS Code installed, install the C/C++ extension by searching for 'c++' in the Extensions view (shortcut: **Ctrl** **Shift** **X**).

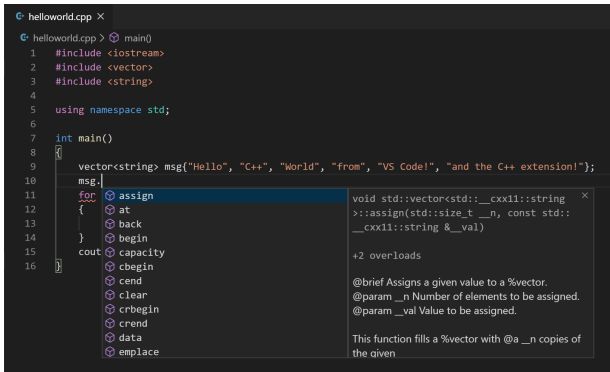
The C++ extension accesses the C++ compiler you have installed on your machine to build your program.



Using a IDE

4. Using VS Code

You can then use VS Code to edit C/C++ code:



```
helloworld.cpp X
helloworld.cpp > main()
1  #include <iostream>
2  #include <vector>
3  #include <string>
4
5  using namespace std;
6
7  int main()
8  {
9      vector<string> msg{"Hello", "C++", "World", "from", "VS Code!", "and the C++ extension!"};
10
11     msg.
12     {
13         for assign
14         {
15             at
16             back
17             begin
18             capacity
19             cbegin
20             cend
21             clear
22             crbegin
23             crend
24             data
25             emplace
26         }
27     }
28     cout
29     cbegin
30     cend
31     clear
32     crbegin
33     crend
34     data
35     emplace
36 }
```

void std::vector<std::__cxx11::string>::assign(std::size_t __n, const std::__cxx11::string &__val)

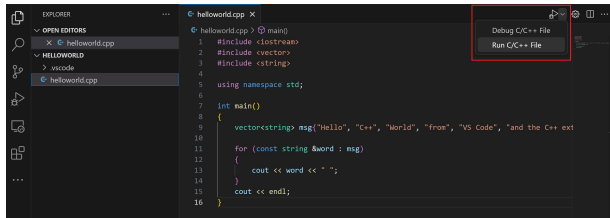
+2 overloads

@brief Assigns a given value to a %vector.
@param __n Number of elements to be assigned.
@param __val Value to be assigned.

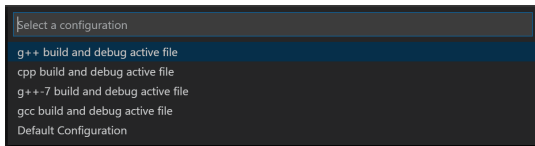
This function fills a %vector with @a __n copies of the given

Using a IDE

To run code, open the file so that it is the active file.
Then press the play button in the top right corner of the editor.



Choose **g++ build and debug active file** from the list of detected compilers on your system. You'll be asked to choose a compiler the first time you run the code. This compiler will be set as the "default" compiler.



Using a IDE

After the build succeeds, your program's output will appear in the integrated terminal.

```
Hello C++ World from VS Code and the C++ extension!
```

```
C:\projects\helloworld>
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

General Debugging Commands

Breakpoints

A breakpoint is a special marker that tells the debugger to stop execution of the program at the breakpoint when running in debug mode.

When you set a breakpoint, you will see a new type of icon appear. Like many IDEs, VS Code uses a red circle.



When debugging, the program runs until execution reaches the breakpoint. Then it returns control to you so you can debug starting at that point.

General Debugging Commands

Step over

Like step into, the step over command executes the next statement in the normal execution path of the program. However, whereas step into enters function calls to execute them line by line, step over will execute an entire function and return control to you after the function has been executed.

Step into

The step into command executes the next statement in the normal execution path of the program, and then pauses program execution so we can examine the program's state using the debugger. If the statement being executed contains a function call, step into causes the program to jump to the top of the function being called, where it will pause.

Step out

Step out executes all remaining code in the function currently being executed, and then returns control to you when the function has returned.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

General Debugging Commands

Step back

Some debuggers (such as Visual Studio Enterprise Edition and) have introduced a stepping capability generally referred to as step back or reverse debugging.

The goal of a step back is to undo the last step, so you can return the program to a prior state. This can be useful if you overstep, or if you want to re-examine a statement that just executed.

Implementing step back requires a great deal of sophistication on the part of the debugger (because it has to keep track of a separate program state for each step). Because of the complexity, this capability isn't standardized yet, and varies by debugger.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Debugging in VS Code

In VS Code, set a breakpoint by clicking on the editor margin or using **F9** on the current line.



```
helloworld.cpp x
helloworld.cpp > main()
1  #include <iostream>
2  #include <vector>
3  #include <string>
4
5  using namespace std;
6
7  int main()
8  {
9      vector<string> msg{"Hello", "C++", "World", "from", "VS Code", "and the C++ extension!"};
10     for (const string &word : msg)
11     {
12         cout << word << " ";
13     }
14     cout << endl;
15 }
```

The image shows a screenshot of the Visual Studio Code editor. The active file is 'helloworld.cpp'. The code is a C++ program that prints a vector of strings. A red square breakpoint icon is set on the left margin of line 12, which contains the statement 'cout << word << " ";'. The editor has a dark theme.

Debugging in VS Code

To start the debugging, from the drop-down next to the play button, select **Debug C/C++ File**.



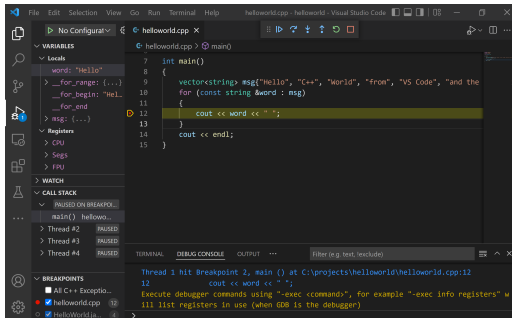
Choose **C/C++: g++ build and debug active file** from the list of detected compilers on your system.

The play button has two modes: **Run C/C++ File** and **Debug C/C++ File**. It will default to the last-used mode. If you see the debug icon in the play button, you can just select the play button to debug, instead of selecting the drop-down menu item.

Debugging in VS Code

We notice several changes in the user interface:

- The **Integrated Terminal** appears at the bottom of the editor. In the **Debug Console** tab, you see output indicating the debugger is running.
- The editor highlights line 12 with the previously set breakpoint.



The **Run and Debug** view on the left shows debugging information. At the top of the code editor, a debugging control panel appears. You can move this around the screen by grabbing the dots on the left side.

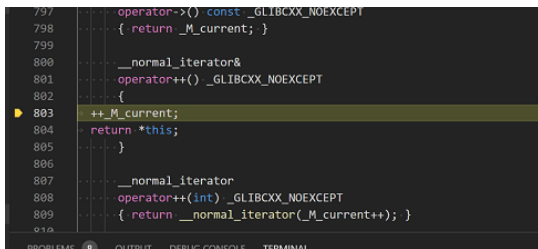
Debugging in VS Code

Now you're ready to start stepping through the code.

Click or press the **Step over** icon in the debugging control panel.

This will advance program execution. Notice the change in the Variables window on the side.

If you like, you can keep pressing **Step over** until all the words in the vector have been printed to the console. But if you are curious, try pressing the Step Into button to step through source code in the C++ standard library.

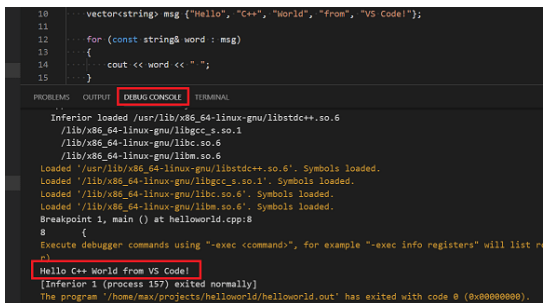


```
797 operator->() const _GLIBCXX_NOEXCEPT
798 { return _M_current; }
799
800 __normal_iterator&
801 operator++() _GLIBCXX_NOEXCEPT
802 {
803     ++_M_current;
804     return *this;
805 }
806
807 __normal_iterator
808 operator++(int) _GLIBCXX_NOEXCEPT
809 { return __normal_iterator(_M_current++); }
910
```

The screenshot shows a VS Code editor window with a dark theme. The code is C++ and appears to be from the standard library. Line 803, which contains the statement `++_M_current;`, is highlighted with a green background. The editor has a scrollbar on the right, and the bottom status bar shows 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'.

Debugging in VS Code

When the code has completed, you can see the output in the Debug Console tab of the integrated terminal, along with some other diagnostic information that is output by GDB.



The screenshot shows a C++ program in a code editor and its execution output in the Debug Console. The code defines a vector of strings and prints them. The Debug Console shows the loading of various system libraries, the execution of the program, and the final output: "Hello C++ World from VS Code!".

```
10 vector<string> msg {"Hello", "C++", "World", "from", "VS Code!"};
11
12 for (const string& word : msg)
13 {
14     cout << word << " ";
15 }
```

PROBLEMS OUTPUT **DEBUG CONSOLE** TERMINAL

Inferior loaded /usr/lib/x86_64-linux-gnu/libstdc++.so.6
/lib/x86_64-linux-gnu/libgcc_s.so.1
/lib/x86_64-linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libm.so.6
Loaded '/usr/lib/x86_64-linux-gnu/libstdc++.so.6'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libgcc_s.so.1'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libc.so.6'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libm.so.6'. Symbols loaded.
Breakpoint 1, main () at helloworld.cpp:8
8 {
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers.
Hello C++ World from VS Code!
[Inferior 1 (process 157) exited normally]
The program '/home/max/projects/helloworld/helloworld.out' has exited with code 0 (0x00000000).

To keep track of the value of a variable as your program executes, set a **watch** on the variable.

I/O in C++

Input and output functionality is not defined as part of the core C++ language, but rather is provided through the C++ standard library (and thus resides in the `std` namespace).

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

I/O in C++

Input and output functionality is not defined as part of the core C++ language, but rather is provided through the C++ standard library (and thus resides in the `std` namespace).

In the previous lecture and tutorial, we included the `iostream` library header and made use of the `cin` and `cout` objects to do simple I/O. Here, we will not take a closer look at the `iostream` library.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

I/O in C++

Including the `iostream` library gives you access to a whole hierarchy of classes responsible for providing I/O functionality (including one class that is actually named `iostream`).

At its most basic, I/O in C++ is implemented with **streams**. Abstractly, a stream is a sequence of bytes that can be accessed sequentially. Over time, a stream may produce or consume potentially unlimited amounts of data.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

I/O in C++

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Including the `iostream` library gives you access to a whole hierarchy of classes responsible for providing I/O functionality (including one class that is actually named `iostream`).

At its most basic, I/O in C++ is implemented with **streams**. Abstractly, a stream is a sequence of bytes that can be accessed sequentially. Over time, a stream may produce or consume potentially unlimited amounts of data.

Input streams are used to hold input from a data producer, such as a keyboard, a file, or a network. For example, the user may press a key while the program is currently not expecting any input. Rather than ignore the users keypress, the data is put into an input stream, where it will wait until the program is ready for it.

I/O in C++

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Including the `iostream` library gives you access to a whole hierarchy of classes responsible for providing I/O functionality (including one class that is actually named `iostream`).

At its most basic, I/O in C++ is implemented with **streams**. Abstractly, a stream is a sequence of bytes that can be accessed sequentially. Over time, a stream may produce or consume potentially unlimited amounts of data.

Input streams are used to hold input from a data producer, such as a keyboard, a file, or a network. For example, the user may press a key while the program is currently not expecting any input. Rather than ignore the users keypress, the data is put into an input stream, where it will wait until the program is ready for it.

Output streams are used to hold output for a particular data consumer, such as a monitor, a file, or a printer. When writing data to an output device, the device may not be ready to accept that data yet – for example, the printer may still be warming up when the program writes data to its output stream. The data will sit in the output stream until the printer begins consuming it.

I/O in C++

A **standard stream** is a pre-connected stream provided to a computer program by its environment. C++ comes with four predefined standard stream objects that have already been set up for your use.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

I/O in C++

A **standard stream** is a pre-connected stream provided to a computer program by its environment. C++ comes with four predefined standard stream objects that have already been set up for your use.

We have already used the first three of them:

`cin`: an `istream` object tied to the standard input (typically the keyboard)

`cout`: an `ostream` object tied to the standard output (typically the monitor)

`cerr`: an `ostream` object tied to the standard error (typically the monitor), providing unbuffered output

`clog`: an `ostream` object tied to the standard error (typically the monitor), providing buffered output

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

File I/O in C++ works very similarly to normal I/O (with a few minor added complexities).

There are 3 basic file I/O classes in C++: `ifstream` (derived from `istream`), `ofstream` (derived from `ostream`), and `fstream` (derived from `iostream`). These classes do file input, output, and input/output respectively.

To use the file I/O classes, it is necessary to include the `fstream` header.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

To do **file output** in the following example, we're going to use the `ofstream` class. This usage is straightforward:

```
1  #include <fstream>
2  #include <iostream>
3
4  int main()
5  {
6      // ofstream is used for writing files; we'll make a file called Sample.txt
7      std::ofstream outf{"Sample.txt"};
8
9      // If we couldn't open the output file stream for writing
10     if (!outf)
11     {
12         // Print an error and exit
13         std::cerr << "Error: Sample.txt could not be opened for writing!\n";
14         return 1;
15     }
16
17     // We'll write two lines into this file
18     outf << "This is line 1\n";
19     outf << "This is line 2\n";
20
21     return 0;
22
23     // When outf goes out of scope, the ofstream destructor will close the file
24 }
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

To do **file input**, we use the `ifstream` class.

In this **example**, we take the file we wrote in the last example and read it back in from disk. Note that `ifstream` returns a 0 if we've reached the end of the file (EOF). We'll use this fact to determine how much to read.

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      // We use ifstream to read from a file called Sample.txt
8      std::ifstream inf{"Sample.txt"};
9
10     // If we couldn't open the output file stream for reading, print an error and exit
11     if (!inf)
12     {
13         std::cerr << "Error: Sample.txt could not be opened for reading!\n";
14         return 1;
15     }
16
17     // While there's still stuff left to read
18     std::string strInput{};
19     while (inf >> strInput)
20         std::cout << strInput << '\n';
21
22     return 0;
23
24     // When inf goes out of scope, the ifstream destructor will close the file
25 }
```

File I/O in C++

The result produced from the above code looks like the following:

```
This  
is  
line  
1  
This  
is  
line  
2
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

The result produced from the above code looks like the following:

```
This  
is  
line  
1  
This  
is  
line  
2
```

This is not what we had intended.

Remember that the extraction operator breaks on whitespace. In order to read in entire lines, we'll have to use the `getline()` function.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      // We use ifstream to read from a file called Sample.txt
8      std::ifstream inf{"Sample.txt"};
9
10     // If we couldn't open the input file stream for reading, print an error and exit
11     if (!inf)
12     {
13         std::cerr << "Error: Sample.txt could not be opened for reading!\n";
14         return 1;
15     }
16
17     // While there's still stuff left to read
18     std::string strInput{};
19     while (std::getline(inf, strInput))
20         std::cout << strInput << '\n';
21
22     return 0;
23
24     // When inf goes out of scope, the ifstream destructor will close the file
25 }
```

This produces the result:

```
This is line 1
This is line 2
```

File I/O in C++

When we run the above example, we see that in case the file already exists, it is **overwritten**. In some cases, this might be the desired behavior of a program.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

When we run the above example, we see that in case the file already exists, it is **overwritten**. In some cases, this might be the desired behavior of a program.

However, what if instead we want to **append data** on the end of a file?

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

When we run the above example, we see that in case the file already exists, it is **overwritten**. In some cases, this might be the desired behavior of a program.

However, what if instead we want to **append data** on the end of a file?

The file stream constructor can take an optional second parameter that allows you to specify information about how the file should be opened. This parameter is called **mode**, and the valid flags that it accepts are from the `ios` class.

ios file mode	meaning
app	Opens the file in append mode
ate	Opens the file in binary mode (instead of text mode)
binary	Opens the file in binary mode (instead of text mode)
in	Opens the file in read mode (default for <code>ifstream</code>)
out	Opens the file in write mode (default for <code>ofstream</code>)
trunc	Erases the file if it already exists

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

File I/O in C++

example: Let's write a program that appends two more lines to the `Sample.txt` file we previously created:

```
1  #include <iostream>
2  #include <fstream>
3
4  int main()
5  {
6      // We'll pass the ios::app flag to tell the ofstream to append rather than rewrite
7      // the file. We do not need to pass in std::ios::out because ofstream defaults
8      // to std::ios::out
9      std::ofstream outf{"Sample.txt", std::ios::app};
10
11     // If we couldn't open the output file stream for writing, print an error and exit
12     if (!outf)
13     {
14         std::cerr << "Error: Sample.txt could not be opened for writing!\n";
15         return 1;
16     }
17
18     outf << "This is line 3\n";
19     outf << "This is line 4\n";
20
21     return 0;
22
23     // When outf goes out of scope, the ofstream destructor will close the file
24 }
```

File I/O in C++

Now if we take a look at `Sample.txt` (using one of the above sample programs that prints its contents, or loading it in a text editor), we will see the following:

```
This is line 1
This is line 2
This is line 3
This is line 4
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

Generally, **arrays** are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array in C++, we define the type, specify the name of the array followed by square brackets and specify the number of elements it should store.

example:

```
string astro_objects[4];
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

Generally, **arrays** are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array in C++, we define the type, specify the name of the array followed by square brackets and specify the number of elements it should store.

example:

```
string astro_objects[4];
```

We insert values into it like in the following **example:**

```
string astro_objects[4] = {"star", "planet", "asteroid", "galaxy"};
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

Generally, **arrays** are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array in C++, we define the type, specify the name of the array followed by square brackets and specify the number of elements it should store.

example:

```
string astro_objects[4];
```

We insert values into it like in the following **example:**

```
string astro_objects[4] = {"star", "planet", "asteroid", "galaxy"};
```

Another **example** shows how to create an array of three integers:

```
int myNum[3] = {10, 20, 30};
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

To **access** the elements of an array, we refer to the index number inside square brackets [].

This **example** statement accesses the value of the first element in cars:

```
string astro_objects[4] = {"star", "planet", "asteroid", "galaxy"};
cout << astro_objects[0];
// Outputs star
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

You can **loop through the array elements** with the for loop.

The following **example** outputs all elements in the cars array:

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};

for (int i = 0; i < 5; i++) {
    cout << cars[i] << "\n";
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

You can **loop through the array elements** with the for loop.

The following **example** outputs all elements in the cars array:

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};

for (int i = 0; i < 5; i++) {
    cout << cars[i] << "\n";
}
```

This **example** outputs the index of each element together with its value:

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};

for (int i = 0; i < 5; i++) {
    cout << i << " = " << cars[i] << "\n";
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

To get the **size** of an array, you can use the `sizeof()` operator.

example:

```
int myNumbers[5] = {10, 20, 30, 40, 50};  
cout << sizeof(myNumbers);
```

Result: **20**

Why did the result show 20 instead of 5, when the array contains 5 elements?

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

To get the **size** of an array, you can use the `sizeof()` operator.

example:

```
int myNumbers[5] = {10, 20, 30, 40, 50};  
cout << sizeof(myNumbers);
```

Result: **20**

Why did the result show 20 instead of 5, when the array contains 5 elements?

The reason is that the `sizeof()` operator returns the size of a type in bytes.

An `int` type is usually 4 bytes, so from the example above, 4×5 (4 bytes \times 5 elements) = 20 bytes.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Arrays in C++

To find out how many elements an array has, you have to divide the size of the array by the size of the data type it holds.

example:

```
int myNumbers[5] = {10, 20, 30, 40, 50};  
int getArrayLength = sizeof(myNumbers) / sizeof(int);  
cout << getArrayLength;
```

Result: **5**

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Multi-Dimensional Arrays in C++

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Based on what we saw so far, we can also construct **multi-dimensional arrays**. A multi-dimensional array is an array of arrays.

To declare a multi-dimensional array, define the variable type, specify the name of the array followed by square brackets which specify how many elements the main array has, followed by another set of square brackets which indicates how many elements the sub-arrays have. See the following **example**:

```
string letters[2][4];
```

In a multi-dimensional array, each element in an array literal is another array literal.

We **insert** values like the following:

```
string letters[2][4] = {  
    { "A", "B", "C", "D" },  
    { "E", "F", "G", "H" }  
};
```

Multi-Dimensional Arrays in C++

Each set of square brackets in an array declaration adds another dimension to an array. An array like the one above is said to have two dimensions.

Arrays can have any number of dimensions. The more dimensions an array has, the more complex the code becomes. The following array has three dimensions:

```
string letters[2][2][2] = {  
    {  
        { "A", "B" },  
        { "C", "D" }  
    },  
    {  
        { "E", "F" },  
        { "G", "H" }  
    }  
};
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Multi-Dimensional Arrays in C++

To **access** an element of a multi-dimensional array, specify an index number in each of the array's dimensions.

In the following **example**, the statement accesses the value of the element in the first row (0) and third column (2) of the `letters` array.

```
string letters[2][4] = {  
    { "A", "B", "C", "D" },  
    { "E", "F", "G", "H" }  
};  
  
cout << letters[0][2]; // Outputs "C"
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Multi-Dimensional Arrays in C++

To **access** an element of a multi-dimensional array, specify an index number in each of the array's dimensions.

In the following **example**, the statement accesses the value of the element in the first row (0) and third column (2) of the `letters` array.

```
string letters[2][4] = {  
    { "A", "B", "C", "D" },  
    { "E", "F", "G", "H" }  
};  
  
cout << letters[0][2]; // Outputs "C"
```

Remember: Array indices start with 0: [0] is the first element. [1] is the second element, etc.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Multi-Dimensional Arrays in C++

To **change** the value of an element in a multi-dimensional array, refer to the index number of the element in each of the dimensions:

example:

```
string letters[2][4] = {  
    { "A", "B", "C", "D" },  
    { "E", "F", "G", "H" }  
};  
  
cout << letters[0][0]; // outputs "A"  
  
letters[0][0] = "Z";  
  
cout << letters[0][0]; // now outputs "Z" instead of "A"
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Multi-Dimensional Arrays in C++

To **loop** through a multi-dimensional array, you need one loop for each of the array's dimensions.

The following **example** outputs all elements in the letters array.

```
string letters[2][4] = {
    { "A", "B", "C", "D" },
    { "E", "F", "G", "H" }
};

for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 4; j++) {
        cout << letters[i][j] << "\n";
    }
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

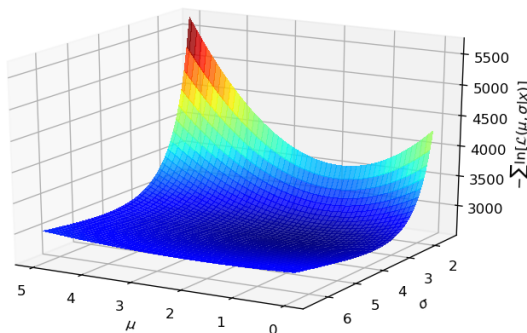
Multi-Dimensional Arrays in C++

Why are we using multi-dimensional arrays?

Multi-dimensional arrays are great at **representing grids**.

Imagine for example a 3D array that stores the value z computed for values x and y , or the likelihood value computed for some variables x_i .

Maximum Likelihood Estimation



Motivation

Debugging

File I/O

Complex Data Structures

Functions

References and Pointers

Summary & Outlook

C++ Structures

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.

Unlike an array, a structure can contain many different data types (int, string, bool, etc.).

C++ Structures

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.

Unlike an array, a structure can contain many different data types (int, string, bool, etc.).

To create a structure, use the `struct` keyword and declare each of its members inside curly braces.

After the declaration, specify the name of the structure variable (myStructure in the **example** below):

```
struct {                // Structure declaration
    int myNum;           // Member (int variable)
    string myString;     // Member (string variable)
} myStructure;          // Structure variable
```

C++ Structures

To **access** members of a structure, use the dot syntax.

example: Assign data to members of a structure and print it

```
// Create a structure variable called myStructure
struct {
    int myNum;
    string myString;
} myStructure;

// Assign values to members of myStructure
myStructure.myNum = 1;
myStructure.myString = "Hello World!";

// Print members of myStructure
cout << myStructure.myNum << "\n";
cout << myStructure.myString << "\n";
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

C++ provides the pre-defined function `main()`, which is used to start the execution of code. But you can also create your own functions.

To create (often referred to as declare) a function, specify the name of the function, followed by parentheses `()`:

The **syntax** is:

```
void myFunction() {  
    // code to be executed  
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ provides the pre-defined function `main()`, which is used to start the execution of code. But you can also create your own functions.

To create (often referred to as declare) a function, specify the name of the function, followed by parentheses `()`:

The **syntax** is:

```
void myFunction() {  
    // code to be executed  
}
```

example explained:

`myFunction()` is the name of the function.

`void` means that the function does not have a return value.

C++ Functions

Functions must be declared before the `main()` function.

If a user-defined function, such as `myFunction()` is declared after the `main()` function, an error will occur.

example:

```
int main() {  
    myFunction();  
    return 0;  
}  
  
void myFunction() {  
    cout << "code running";  
}  
  
// Error
```

pause However, it is possible to separate the declaration and the definition of the function for code optimization.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

You will often see C++ programs that have function declaration above `main()`, and function definition below `main()`. This will make the code better organized and easier to read.

example:

```
// Function declaration
void myFunction();

// The main method
int main() {
    myFunction(); // call the function
    return 0;
}

// Function definition
void myFunction() {
    cout << "code running";
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

To **call** a function, write the function's name followed by two parentheses () and a semicolon ;

In the following **example**, myFunction() is used to print a text when it is called.

```
//Inside main, call myFunction():

// Create a function
void myFunction() {
    cout << "code running";
}

int main() {
    myFunction(); // call the function
    return 0;
}

// Outputs "I just got executed!"
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma, using the following **syntax**:

```
void functionName(parameter1, parameter2, parameter3) {  
    // code to be executed  
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

The following **example** has a function that takes a string called `fname` as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
void myFunction(string fname) {  
    cout << fname << " code running\n";  
}  
  
int main() {  
    myFunction("quasar");  
    myFunction("star");  
    myFunction("asteroid");  
    return 0;  
}  
  
// quasar code running  
// star code running  
// asteroid code running
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

The following **example** has a function that takes a string called `fname` as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
void myFunction(string fname) {  
    cout << fname << " code running\n";  
}  
  
int main() {  
    myFunction("quasar");  
    myFunction("star");  
    myFunction("asteroid");  
    return 0;  
}  
  
// quasar code running  
// star code running  
// asteroid code running
```

When a parameter is passed to the function, it is called an argument. So, from the example above: `fname` is a parameter, while `quasar`, `star` and `asteroid` are arguments.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

Functions can have multiple parameters. See the following **example**:

```
void myFunction(string fname, int age) {
    cout << fname << " code running. " << age << " files. \n";
}

int main() {
    myFunction("quasar", 30);
    myFunction("star", 4);
    myFunction("asteroid", 92);
    return 0;
}

// quasar code running. 30 files.
// star code running. 4 files.
// asteroid code running. 92 files.
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

Functions can have multiple parameters. See the following **example**:

```
void myFunction(string fname, int age) {  
    cout << fname << " code running. " << age << " files. \n";  
}  
  
int main() {  
    myFunction("quasar", 30);  
    myFunction("star", 4);  
    myFunction("asteroid", 92);  
    return 0;  
}  
  
// quasar code running. 30 files.  
// star code running. 4 files.  
// asteroid code running. 92 files.
```

Note: When working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

C++ functions have **return values**.

The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function.

example:

```
int myFunction(int x) {  
    return 5 + x;  
}  
  
int main() {  
    cout << myFunction(3);  
    return 0;  
}  
  
// Outputs 8 (5 + 3)
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

This **example** returns the sum of a function with two parameters:

```
int myFunction(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    cout << myFunction(5, 3);  
    return 0;  
}  
  
// Outputs 8 (5 + 3)
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

You can also store the result in a variable.

example:

```
int myFunction(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int z = myFunction(5, 3);  
    cout << z;  
    return 0;  
}  
  
// Outputs 8 (5 + 3)
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Functions

So far, we used normal variables when passing parameters to a function. You can also pass a **reference** to the function. This is called **pass by reference** and can be useful when you need to change the value of the arguments. See the following **example**:

```
void swapNums(int &x, int &y) {
    int z = x;
    x = y;
    y = z;
}

int main() {
    int firstNum = 10;
    int secondNum = 20;

    cout << "Before swap: " << firstNum << " " << secondNum << "\n";

    // Call the function
    swapNums(firstNum, secondNum);

    cout << "After swap: " << firstNum << " " << secondNum << "\n";

    return 0;
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Variable Scope

In C++, variables are only accessible inside the region they are created. This is called **scope**.

A variable created inside a function belongs to the **local scope** of that function, and can only be used inside that function.

example:

```
void myFunction() {  
    // Local variable that belongs to myFunction  
    int x = 5;  
  
    // Print the variable x  
    cout << x;  
}  
  
int main() {  
    myFunction();  
    return 0;  
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ Variable Scope

A variable created outside of a function is called a global variable and belongs to the **global scope**.

Global variables are available from within any scope, global and local.

example:

```
// A variable created outside of a function is global

// Global variable x
int x = 5;

void myFunction() {
    // We can use x here
    cout << x << "\n";
}

int main() {
    myFunction();

    // We can also use x here
    cout << x;
    return 0;
}
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

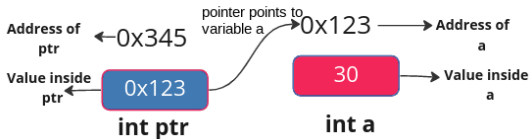
References
and Pointers

Summary &
Outlook

C++ References and Pointers

When a variable is created in C++, a **memory address** is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.

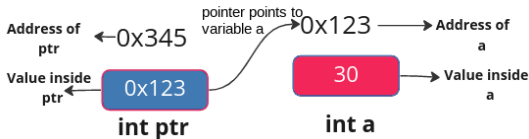
Pointer in C



C++ References and Pointers

When a variable is created in C++, a **memory address** is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.

Pointer in C



Why is it useful to know the memory address?

References and pointers give you the ability to manipulate the data in the computer's memory. This can improve the code performance.

These two features are one of the things that make C++ stand out from other programming languages, like Python and Java.

Motivation

Debugging

File I/O

Complex Data Structures

Functions

References and Pointers

Summary & Outlook

C++ References and Pointers

On the surface, both references and pointers are very similar as both are used to have one variable provide access to another.

Pointers: A pointer is a variable that holds the memory address of another variable. A pointer needs to be dereferenced to access the memory location it points to.

References: A reference is an alias (another name) for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object. A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value).

example:

```
int i = 3;

// A pointer to variable i or "stores the address of i"
int *ptr = &i;

// A reference (or alias) for i.
int &ref = i;
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ References and Pointers

Despite their similarities, there are certain **differences** between pointers and references:

1. Initialization

A pointer can be initialized in this way:

```
int a = 10;
int *p = &a;
// OR
int *p;
p = &a;
```

We can declare and initialize pointer at same step or in multiple line.
In contrast, for references:

```
int a = 10;
int &p = a; // It is correct
// but
int &p;
// The following is incorrect as we should declare
// and initialize references at single step
p = a;
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ References and Pointers

2. Reassignment

A pointer can be re-assigned. This property is useful to implement data structures like linked lists, trees, etc. See the following **example**:

```
int a = 5;
int b = 6;
int *p;
p = &a;
p = &b;
```

A reference cannot be re-assigned, and must be assigned at initialization.

```
int a = 5;
int b = 6;
int &p = a;
// The following will throw an error
// of "multiple declaration is not allowed"
int &p = b;

// However it is valid statement,
int &q = p;
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

C++ References and Pointers

3. Memory Address

A pointer has its own memory address and size on the stack, whereas a reference shares the same memory address with the original variable and takes up no space on the stack.

```
int &p = a;  
cout << &p << endl << &a;
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Application of C++ References and Pointers

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

There are different **use cases** for pointers and references.

As this concept might look rather unusual, we will briefly discuss about the common use cases.

1. Initialization of a string literal

In C, string is not part of the primitive data type. It is only considered as a sequential of characters or array of characters that ends with null terminating character ('\\0').

There are different ways of declaring a string data type in C. One of them is through string literal.

```
char *str = "This is a string literal!";
```

String literal is another way of initializing strings with the help of a pointer. It points to the first character in the string.

Application of C++ References and Pointers

2. Pass by reference

Arguments can be passed to a function either by value or by reference depending on the choice of outcome.

Pass by value creates a local copy of the variable in the called function. Any update made on it would not affect the original in the calling function.

Pass by reference means passing the address of variable to a function. When passed by reference, no duplicate of the variable is created rather it is pointing directly to the original copy and any changes made would reflect in the original copy.

The following **example** demonstrates the difference:

```
int x = 10;
int y = 20;

// Pass by reference
swap(&x, &y);

// Value of x and y were succesfully swapped
printf("%d, %d\n", x, y); // 20, 10
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Application of C++ References and Pointers

3. Dereferencing

The asterisk (*) symbol is used to declare a pointer and also used to dereference pointer. Dereference is a way of accessing the actual value stored in the variable referenced by a pointer.

The following **example** will help you understanding this principle:

```
int x = 10;
int *ptr = &x;

// Address of x
printf("%p\n", ptr); // 0x1f3546772

// Access the value stores in the address ptr is pointing to
printf("%d, %d\n", x, *ptr); // 10, 10

// Update the value stores in ptr to 20
*ptr = 20;

printf("%d, %d\n", x, *ptr); // 20, 20

*ptr == x // true
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

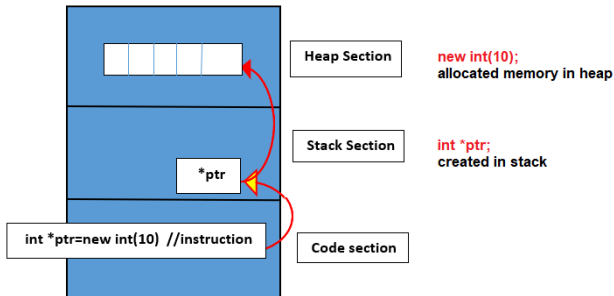
Summary &
Outlook

Application of C++ References and Pointers

4. Dynamic memory allocation

Due to the nature of the execution context and stack section of memory, having access to the local variable of a called function after it has finished executing is impossible.

To make it possible, we will need to dynamically allocate memory in the heap section of memory for the local variable. Functions such as `malloc()`, `realloc()` and `free()` are used for dynamic memory allocation.



Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Application of C++ References and Pointers

4. Dynamic memory allocation

This is an **example** of a program that dynamically allocates 100 bytes of memory in the heap.

```
char *ptr;  
ptr = malloc(sizeof(char) * 100);
```

*ptr points to the first character of the 100 characters in the heap, which can be utilised to get access to the rest of the characters using pointer arithmetic or indexing. It is crucial to note that, it is our responsibility to free or deallocate any memory space allocated in the heap after we are done using it.

```
free(ptr);  
ptr = NULL; // To prevent dangling pointer
```

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Summary: C/C++

With this session, we have already learned many concepts that enable us to write more efficient programs with C/C++.

As this was, of course, only an introduction to C/C++, if you will use this programming language on a regular base, I can recommend that you take a look at additional material, such as the following:

<https://github.com/fffaraz/awesome-cpp>

<https://en.cppreference.com/w/cpp/links/libs>

<http://numerical.recipes/>

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook

Useful Additional Material on C/C++

Libraries

Like Python, for C/C++ libraries provide useful functionality for common problems especially in science.

Standard Template Library (STL): The STL provides a wide range of containers and algorithms for working with data, making it easier to write efficient and effective code.

MPI (Message Passing Interface): MPI is crucial for parallel programming in C++. Libraries like OpenMPI or MPICH provide implementations. Use Cases: Essential for distributed computing tasks that require communication between processes.

Matplotlib C++: A C++ wrapper for the popular Python library Matplotlib, allowing you to create scientific plots and visualize data easily.

mlpack: A free, open-source and header-only software library for machine learning and artificial intelligence.

GNU Scientific Library (GSL): A software library for numerical computations in applied mathematics and science.

Motivation

Debugging

File I/O

Complex Data Structures

Functions

References and Pointers

Summary & Outlook

An Outlook: Object-Oriented Programming

So far, we have used **functional programming** (which involves functions).

In the next session, we will extend this to the concept of **object-oriented programming**.

Motivation

Debugging

File I/O

Complex Data
Structures

Functions

References
and Pointers

Summary &
Outlook