CSU NUPAC Tutorials
2019

Gabriel Palacino
A few comments

Working on a remote machine

• This is the safest way to work
  ‣ Your data are backed up
  ‣ Software is installed and maintained

• Not graphically pretty but very efficient
  ‣ Do not connect using a graphic server, e.g., `ssh -Y(-X)`, if you are not on the same network!
  ‣ If you connect using a graphic server, use text editors on the terminal, e.g., `emacs -nw`

Assignments

• Will be posted at the end of the slides once a week. Due date will also be posted.

• For questions, feel free to email me
  ‣ You can also start a new thread in the group mailing list: `csu-fresno-summer-2019@cern.ch`
  ‣ Usually, this helps the level of the group become more homogeneous, which means we could cover more topics

As usual I’m borrowing a lot of the material from Sam Meehan (University of Washington)
References and Pointers

- Sometimes variables (and even structures) are cumbersome
- Pointers and references allow
  - Efficiency - deal with an address instead of the whole thing
  - Flexibility - can touch the memory itself / break the “one output” rule of C++ functions
- The cost: you may be confused about what your code is doing

Question: What is in your fridge?

Guy1: Oh, well let me open it up and show you. Just don’t touch anything.

Guy2: So, you’ll have to go to my house, at this address, and then look in my kitchen, it’s right there. Oh, and could you put these pickles in there (hands jar of pickles).

(All analogies are imperfect … )
Reference Conceptually

- Conceptual hurdle: “memory address is just a number”
- Gain access to that number by “&”
  - “Address of” or “reference” operator
Reference Conceptually

- Conceptual hurdle: “memory address is just a number”
- Gain access to that number by “&”
  - “Address of” or “reference” operator
Reference Conceptually

- Conceptual hurdle: “memory address is just a number”
- Gain access to that number by “&”
  - “Address of” or “reference” operator
- Now “foo” is referring to the address of `myvar`
De-Referencing

- What if we want to know what was at that address?
- We apply the ‘*’ operator
  - “Thing at the address of” or “Dereference” operator
Copying Memory Address

- What if we want to know what was at that address?
- We apply the “*” operator
  - “Thing at the address of” or “Dereference” operator
How to evaluate this

- Can decode all of this into English via
  - (*)“the thing at ..” and (&)“the address of ...”
- baz = *foo = *(&myvar)
  - The thing at the address of myvar (“=1776”)
- Common conception: “oh this isn’t bad”
  - When in trouble, slow down and decode

<table>
<thead>
<tr>
<th>1775</th>
<th>myvar - 1776</th>
<th>1777</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>25</td>
<td>??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23</th>
<th>foo - 24</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>1776</td>
<td>??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>111</th>
<th>bar - 112</th>
<th>113</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>25</td>
<td>??</td>
</tr>
</tbody>
</table>
Pointers - making it concrete

- Pointers are a type modifier
  - Allows you to store the address of a specific type of object
  - Not important during storing → crucial during dereferencing
- “foo” is pointing to 1776 in memory - what if we dereference it?

Different things take different amounts of space!
Pointers - making it concrete

- Pointers are a type modifier
  - Allows you to store the address of a specific type of object
  - Not important during storing → crucial during dereferencing
- "foo" is pointing to 1776 in memory - what if we dereference it?
  - It will only get the correct information in memory if it knows what to get

<table>
<thead>
<tr>
<th>Type names*</th>
<th>Notes on size / precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Exactly one byte in size. At least 8 bits.</td>
</tr>
<tr>
<td>char16_t</td>
<td>Not smaller than char. At least 16 bits.</td>
</tr>
<tr>
<td>char32_t</td>
<td>Not smaller than char16_t. At least 32 bits.</td>
</tr>
<tr>
<td>wchar_t</td>
<td>Can represent the largest supported character set.</td>
</tr>
<tr>
<td>signed char</td>
<td>Same size as char. At least 8 bits.</td>
</tr>
<tr>
<td>signed short int</td>
<td>Not smaller than char. At least 16 bits.</td>
</tr>
<tr>
<td>signed int</td>
<td>Not smaller than short. At least 16 bits.</td>
</tr>
<tr>
<td>signed long int</td>
<td>Not smaller than int. At least 32 bits.</td>
</tr>
<tr>
<td>signed long long int</td>
<td>Not smaller than long. At least 64 bits.</td>
</tr>
</tbody>
</table>

GOOD

Thinks that "the thing at" foo is an int (32 bits)

Thinks that "the thing at" foo is a signed long long int (64 bits)
Pointers - making it concrete

- Pointers are a type modifier
  - Allows you to store the address of a specific type of object
  - Not important during storing → crucial during dereferencing

- “foo” is pointing to 1776 in memory - what if we dereference it?
  - It will only get the correct information in memory if it knows what to get

<table>
<thead>
<tr>
<th>Type names</th>
<th>Notes on size / precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Exactly one byte in size. At least 8 bits.</td>
</tr>
<tr>
<td>char16_t</td>
<td>Not smaller than char. At least 16 bits.</td>
</tr>
<tr>
<td>char32_t</td>
<td>Not smaller than char16_t. At least 32 bits.</td>
</tr>
<tr>
<td>wchar_t</td>
<td>Can represent the largest supported character set.</td>
</tr>
<tr>
<td>signed char</td>
<td>Same size as char. At least 8 bits.</td>
</tr>
<tr>
<td>signed short</td>
<td>Not smaller than char. At least 16 bits.</td>
</tr>
<tr>
<td>signed int</td>
<td>Not smaller than short. At least 16 bits.</td>
</tr>
<tr>
<td>signed long</td>
<td>Not smaller than int. At least 32 bits.</td>
</tr>
<tr>
<td>signed long long int</td>
<td>Not smaller than long. At least 64 bits.</td>
</tr>
</tbody>
</table>

GOOD

BAD

Thinks that “the thing at” foo is an int (32 bits)

Thinks that “the thing at” foo is a signed long long int (64 bits)
Pointers - making it concrete

- Remedy this by telling it what type of thing it is pointing to
  - This is done by making a compound type
  - "*" is NOT the dereference operator here
Pointers - making it concrete

- Remedy this by telling it what type of thing it is pointing to
  - This is done by making a compound type
  - "*" is NOT the dereference operator here

```cpp
#include <iostream>
using namespace std;

int main()
{
    int firstvalue, secondvalue;
    int *mypointer = NULL;

    mypointer = &firstvalue;
    *mypointer = 10;
    mypointer = &secondvalue;
    *mypointer = 20;

    cout << "firstvalue is " << firstvalue << "\n";
    cout << "secondvalue is " << secondvalue << "\n";
    return 0;
}
```
Pointers - making it concrete

- Remedy this by telling it what **type** of thing it is pointing to
  - This is done by making a compound type
  - "*" is **NOT** the dereference operator here

I am so ready to point to an integer, just tell me where it is!
Pointers - making it concrete

- Remedy this by telling it what type of thing it is pointing to
  - This is done by making a compound type
  - "*" is NOT the dereference operator here
Pointers - making it concrete

- Remedy this by telling it what **type** of thing it is pointing to
  - This is done by making a compound type
  - "*" is NOT the dereference operator here
Pointers to complicated things

- Reminder: structs (and classes) are like variables on steroids
- We can treat them just like we treat normal types when dealing with pointers

```cpp
myDate(Date) 984
```

```cpp
struct Date {
  int day, month, year;
};

int main() {
  Date myDate;
  myDate.day = 2;
  myDate.month = 2;
  myDate.year = 1987;
}
```

```cpp
Date myBirthday;
myBirthday = myDate;
```

```cpp
cout << "Date of my birthday: " << endl;
  << myBirthday.day << "-" << myBirthday.month << "-" << myBirthday.year;
return 0;
```
Pointers to complicated things

- Reminder: structs (and classes) are like variables on steroids
- We can treat them just like we treat normal types when dealing with pointers

\[ \text{myBirthday} = \text{myDate} \]

\[ \text{myBirthday} = \text{myDate}(2, 2, 1987) \]
Pointers to complicated things

- Reminder: structs (and classes) are like variables on steroids
- We can treat them just like we treat normal types when dealing with pointers

I am so ready to point to a date, just tell me where it is!
Pointer #shortcut

- Tiresome to have to go through 2-step process to get at the members of a pointer to a struct
  - *myBirthday : gives us the stuff at the place in memory
  - (*myBirthday).day : gives us the specific member “day”

- Bottom line : “->” is like “.” for pointers
  - (*myBirthday).day = myBirthday->day

This is the most common way - keep your head up when we move to classes ... and then Root
Revisiting Arrays

- "Arrays are like vectors in linear algebra"
  - But we can do better than that
- Arrays are actually pointers to the first spot in a group of allocated memory
  - "[n]" says we move our pointer “n” spots to the right
Revisiting Arrays

- “Arrays are like vectors in linear algebra”
  - But we can do better than that
- Arrays are actually pointers to the first spot in a group of allocated memory
  - “[n]” says we move our pointer “n” spots to the right
Revisiting Arrays

- “Arrays are like vectors in linear algebra”
  - But we can do better than that
- Arrays are actually pointers to the first spot in a group of allocated memory
  - “[n]” says we move our pointer “n” spots to the right
Revisiting Arrays

- “Arrays are like vectors in linear algebra”
  - But we can do better than that
- Arrays are actually pointers to the first spot in a group of allocated memory
  - “[n]” says we move our pointer “n” spots to the right

```
// more pointers
#include <iostream>
using namespace std;

int main()
{
    int numbers[6];
    int **p;
    p = &numbers[2];
    *p = 20;
    p = numbers + 3;
    *p = 48;
    p = &numbers;
    *(p++) = 56;
    return 0;
}
```

What the heck is going on here?
Functions (aka “black boxes”)

- Black boxes are double-edged swords
  - Bad if you don’t know what is going on inside of it and/or it's not well documented
  - Good if you can effectively use it to simplify/factorize code and facilitate writing

- Difficult and subjective points
  - When do I put something in a function?
  - How do I best put this in a function?
  - Is this the most efficient way to create this function?
  - ... this comes with experience
Functions (aka “black boxes”)

- Four main components to a function
  - **Type**: this is the type of thing that it will return after being run inside main()
  - **Name**: this is the identifying name (like a variable)
  - **Arguments**: this is the input, as many as you like, typically comes with type
  - **Return**: this is what it spits out at the end

```
functionName(type arg1, type arg2, …)
  arg1(type)
  arg2(type)
  Stuff happens
  return functionReturnValue;
```

Function:
Do stuff to “input” and generate some “output”
Functions (aka “black boxes”)

- Similar to a struct in the layout
  - [1] Declare the function → prototype specifies the use of the blackbox
  - [2] Define the function → implement the “stuff”
- In this case they are done together
  - Definition must be done before you use it

```
int addition(int a, int b)
{
    int r;
    r = (a + b) / 2;
    return r;
}
```

Functions

- Similar to a struct in the layout
  - [1] Declare the function ➔ Typically done in header
  - [2] Define the function ➔ Typically done in source
- Now we see the power of the header
  - Prototypes in header = user interface
  - `main()` can come before function definitions
- What is this code missing?

```cpp
// function prototypes
#include <iostream>
using namespace std;

int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);
```
Functions

- Similar to a struct in the layout
  - [1] Declare the function → Typically done in header
  - [2] Define the function → Typically done in source

- Now we see the power of the header
  - Prototypes in header = user interface
  - main() can come before function definitions

- What is this code missing? **COMMENTS**
Functions - Arguments

- There exist multiple ways to pass arguments to functions - we will cover two
  - **by value**: Arguments passed to the function are *copied* into new chunks in memory
    - Less complicated: only way to influence outside world is by creating the return value
      - “What happens in the function, stays in the function” ... (except the return value)
    - Less flexible: you can only return one thing from a function
    - Inefficient/slow: copies of (sometimes large) objects in memory are made

```c
int addition(int afunc, int bfunc)
{
    afunc = afunc + bfunc;
    return rfunc;
}
```

**Output**

```
Stuff happens
```

```c
Output
```
Functions - Arguments

- There exist multiple ways to pass arguments to functions - we will cover two

- **by value**: Arguments passed to the function are copied into new chunks in memory
  - Less complicated: only way to influence outside world is by creating the return value
    - “What happens in the function, stays in the function” … (except the return value)
  - Less flexible: you can only return one thing from a function
  - Inefficient/slow: copies of (sometimes large) objects in memory are made

```c
int addition(int afunc, int bfunc)
{
    afunc = afunc + bfunc;
    return rfunc;
}
```

```
5 6
```

```
Stuff happens
```

```
Output
```
Functions - Arguments

- There exist multiple ways to pass arguments to functions - we will cover two

  - **by reference**: Location of argument ("lightweight baby") is passed to the function
    - More complicated: what happens in your function can influence the outside world
    - More flexible: you can “return” more than one thing from a function
    - Efficient/Fast: only need to pass along the address (int) not a whole object

```
int addition(int& afunc, int& bfunc)

afunc(int) bfunc(int)

"&ainput" "&binput"

Stuff happens

afunc = afunc+bfunc;

rfunc(int)

4

Output
```

```
return rfunc;

5
6
```

```
ainput(int)

5
```

```
binput(int)

6
```
There exist multiple ways to pass arguments to functions - we will cover two:

- **by reference**: Location of argument ("lightweight baby") is passed to the function
  - More complicated: what happens in your function can influence the outside world
  - More flexible: you can “return” more than one thing from a function
  - Efficient/Fast: only need to pass along the address (int) not a whole object

```cpp
int addition(int& afunc, int& bfunc)
{
    afunc = afunc + bfunc;
    return afunc;
}
```

```cpp
int afunc(int)
{
    "&ainput"
    return afunc;
}
```

```cpp
int bfunc(int)
{
    "&ainput"
    return bfunc;
}
```

So we can play with the external memory from within the function!!!!!!
What does this look like in code?

- This stresses the fact that when dealing with functions, read them carefully!
  - And print many things to the screen “cout” often
- Will these give the same printout in the terminal? Why or why not?
What does this look like in code?

- This stresses the fact that when dealing with functions, read them carefully!
  - And print many things to the screen “cout” often
- Will these give the same printout in the terminal?
Not covered here

- Reminder: learning C++ takes experimentation and playing outside of what we do here. Exercise your googling prowess to look into these things
  - This is a set of things that are important but that I deeply secondary
- [1] Pointers to pointers to pointers to...
- [2] Void/NULL pointers
  - “With great power comes great responsibility”
- [3] Strings as pointers to arrays of characters
- [4] Pointer arithmetic and manipulation
  - We did a little bit of it...
  - If you want to work on hardware/electronics, this is essential
- [5] Recursive functions
- [6] Pointers to functions
Now let’s do some live programming