COMP 3522

Object Oriented Programming in C++

Week 2, Day 2

Agenda

1. Functions, references, local variables, and return values
2. Right-left rule
3. string and stringstream
4. new and delete keywords
5. Structs & Unions
6. C++ Vectors

COMP 3522

FUNCTIONS AND REFERENCES

Functions and references

- A function **cannot** return a reference to a local object.
- This is a question of **scope**.

```
int& f()
{
   int n = 1;
   return n;
}
```

Functions and references

- Everything in C++ has scope (of course):
  1. Global
  2. Local.

```
int& f()
{
   int n = 1;
   return n;
}
```

Will this work?

```
int& f()
{
    int n = 1;
    return n;
}
```

Will this?

```
int * g()
{
    int a[10];
    a[0] = 1;
    return a;
}
```

What can we do?

- Use a global object
- Pass a reference

- Can we really use a global object?
- We can, but we shouldn’t
- Avoid globals
- Use references

How exactly do functions work in C++?

```
int function_1(int n)
{
    return n;
}
```

- Receives a copy of an integer
- Returns a copy of the copy

How exactly do functions work in C++?

```
int function_2(int& n)
{
    return n;
}
```

- Gets the original integer
- Returns a copy of the original

How exactly do functions work in C++?

```
int& function_3(int n)
{
    return n;
}
```

INVALID (will compile, but don’t do this!)

How exactly do functions work in C++?

```
int& function_4(int& n)
{
    return n;
}
```

- Gets the original integer
- Returns the original

RIGHT-LEFT RULE

What’s this monstrosity?

```
int a [100];
int (&ref) [100] = a;
```

What’s this monstrosity?

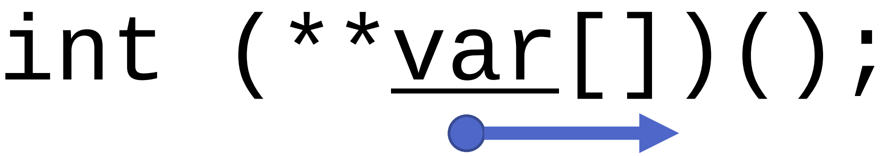
```
void swap(int*&p, int *&q)
{
    int * temp = p;
    p = q;
    q = temp;
}
```

Right left rule breakdown

```
int (**var[])();
```

- Start from variable name (var)

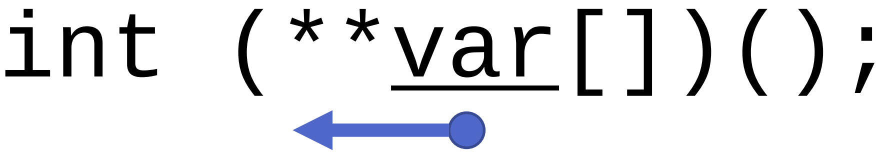
Right left rule breakdown



```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right bracket)

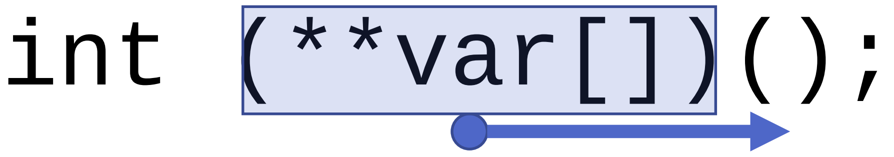
Right left rule breakdown



```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right bracket)
- Then go left of ppf. Read what we see until we see left bracket (

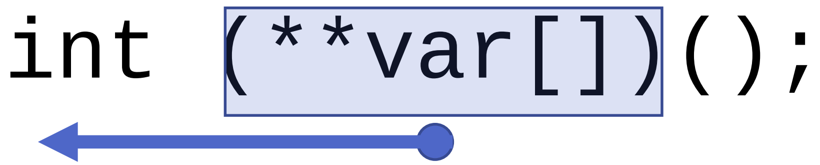
Right left rule breakdown



```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right bracket)
- Then go left of ppf. Read what we see until we see left bracket (
- Exit bracket and go right. Read what we see until we see bracket or semicolon

Right left rule breakdown



```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right bracket)
- Then go left of ppf. Read what we see until we see left bracket (
- Exit bracket and go right. Read what we see until we see bracket or semicolon
- Go left of where we exited bracket. Read what we see until we see bracket or finish
- Repeat steps as needed

Right left rule breakdown

```
int (**var[])();
```

- Start from variable name (var)
- Keep going right. Read what we see until we see right bracket)
- Then go left of ppf. Read what we see until we see left bracket (
- Exit bracket and go right. Read what we see until we see bracket or semicolon
- Go left of where we exited bracket. Read what we see until we see bracket or finish
- Repeat steps as needed

"var is an array of pointers to pointers to a function that returns int"

What’s happening here?

```
int ***ppp;
int (**ppa)[];
int (**ppf)();
int *(*pap)[];
int (*paa)[][];
int (*paf)[]();
```

Let’s examine the RL rule

```
int ***ppp;
```

More RL rule

```
int (**ppa)[];
```

More RL rule

```
int (**ppf)();
```

More RL rule

```
int *(*pap)[];
```

More RL rule

```
int (*paa)[][];
```

More RL rule

```
int (*paf)[]();
```

More RL rule (step by step)

```
int * (* (*fp1) (int) ) [10];
```

Start from the variable name (fp1)

Nothing to right but) so go left to find * (is a pointer)

Jump out of parentheses and encounter (int) (to a function that takes an int as argument)

Go left, find *(and returns a pointer)

Jump put of parentheses, go right and hit [10] (to an array of 10)

Go left find * (pointers to)

Go left again, find int (ints)

More RL rule (step by step)

```
int *( *( *arr[5])()))();
```

Start from the variable name (arr)

Go right, find array subscript (is an array of 5)

Go left, find * (pointers)

Jump out of parentheses, go right to find () (to functions)

Go left, encounter * (that return pointers)

Jump out, go right, find () (to functions)

Go left, find * (that return pointers)

Continue left, find int (to ints).

Even more!

```
int *ptr_to_int;
int *func_returning_ptr_to_int();
int (*ptr_to_func_returning_int)();
int (*array_of_ptr_to_func_returning_int[])();
```

```
CRAZINESS:
int (*(*ptr_to_an_array_of_ptr_to_func_returning_int)[])();
```

Final note: if possible, please don’t write messy initializations like this

string AND stringstream

C++’s std::string class (lower case s)

```
#include <string>

string s1; // Creates a string object!
string s2{"Hello"}; // This does too
string s3{"world!"}; // So does this
cout << s1 << " " << s2 << " " << s3 << endl;
```

*** In C++, the string object needn’t terminate with \0**

The std::string class

- Member functions include:
    - size() returns the number of characters
    - length() returns the number of characters (same thing!)
    - c_str() returns a non-modifiable standard C char array

```
string line;
cin >> line;
const char * c_line = line.c_str();
```

http://en.cppreference.com/w/cpp/string/basic_string

More about the std::string

- We can use **relational operators** (>, <, >=, ==, etc.) to perform lexicographical comparisons (unlike Java which required compareTo or an overridden equals method)
- We can use **square brackets** [ ] to access chars in a std::string
- We can also use the **at(size_type pos)** member function to acquire a reference to the char at the specified index

```
string s = "hello";
cout << s[0]; //prints h
cout << s.at(1); //prints e
```

Classes in C (a short aside, more next week!)

```
string first; // calls default constructor

string second = first; // calls copy constructor

first = second; // calls assignment operator
```

The getline function

- Defined in <string>
- Reads a line of characters from an input stream and puts the characters in the specified string (tosses the newline!)
- Returns the original input stream

```
string input;
getline(cin, input); // returns cin
```

getline (even more information!)

```
getline(inputstream, line, delimiter)
```

Keeps extracting characters until:

1. EOF (sets EOF bit)
2. Delimiter or newline is extracted (and tossed!)
3. So many characters have been extracted that it exceeds the number storable in line (sets the failbit)

getline (failures)

```
getline(cin, input);
```

User Input	string input
Hello\nworld\n	Hello
\nWorld\n	EMPTY
Hello*	Hello (eofbit set)
Hello\n*	Hello
*	No change, eofbit and failbit are set

A C++ standard idiom

- To process a stream line by line, try:

```
string line;
while (getline (cin, line))
{
    /* process your line */
}
```

The istringstream class

- Great for reading and manipulating strings
- Defined in <sstream>
- Actual type is **basic_istringstream<char>**

```
#include <sstream>
string input{“   123abc”};
istringstream iss{input};
int n;
iss >> n;
cout << n << endl;
```

More istringstream

```
istringstream iss;
int n;
iss.str(“   123abc”);
iss >> n;
cout << n << endl;
```

More istringstream

```
istringstream iss;
int n;
string aString
iss.str(“   123abc”);
iss >> n >> aString;
cout << n << endl;
cout << aString << endl;
```

More istringstream

```
istringstream iss;
int n;
string aString
iss.str(“   123a b c”);
iss >> n >> aString;
cout << n << endl;
cout << aString << endl;
```

More istringstream

```
istringstream iss;
iss.str(“   123a b c”);
while(!iss.eof())
{
    string newString;
    iss >> newString;
    cout << newString << endl;
}
```

Even more istringstream

```
string line;
int n, sum{0};
istringstream iss;
while (getline(cin, line)) {
    iss.clear();
    iss.str(line);
    if (iss >> n) {
        sum += n;
    }
}
```

One more istringstream example

```
string line;
int n, sum{0};
istringstream iss;
while (getline(cin, line)) {
    istringstream iss{line};
    if (iss >> n) {
        sum += n;
    }
}
```

istringstream.cpp

The ostringstream class

- Similar!

```
#include <sstream>
ostringstream oss;
int n {3512}, m{2526};
oss << n << “+” << m << “=“ << n + m;
string output = oss.str();
```

ACTIVITY

Modify your program from earlier to determine which 10 words occur most frequently in the Gutenberg file we downloaded. A word is any sequence of char delineated by whitespace. List the top 10 words and their frequency in a file called TopWords.txt

new AND delete

Dynamic memory management

- Refers to **manual** memory management

- Allows us to obtain more memory when required and release it when not necessary

- C++ does not inherently have any technique to allocate memory dynamically for dynamic memory – we have to use library functions

Recall dynamic memory management in C

- There are 4 library functions defined under **stdlib.h** for dynamic memory allocation in C:

1. **malloc()** Allocates requested size of bytes and returns a pointer first byte of allocated space
2. **calloc()** Allocates space for an array elements, initializes to zero and then returns a pointer to memory
3. **realloc()** Changes the size of previously allocated space
4. **free()** Deallocates the previously allocated space

In C++, it's much easier

- We have two operators in C++ for allocating memory dynamically:
  1. **new**
  2. **new[]**

- The new operator returns a pointer to the memory that was just allocated

The new operator

```
int * my_pointer = nullptr;
my_pointer = new int { 3522 };
```

We say that my_pointer refers to a **data object** (not the same an an instance of a class)

We can also do this

```
int * my_pointer = new int;
*my_pointer = 3522;
```

What about new[]

```
int * my_pointer;
my_pointer = new int [5];

for (int i = 0; i < 5; ++i) {
    my_pointer[i] = i;
}
```

What’s the difference?

```
int i; int iArray[10];
```

- Memory is automatically allocated and deallocated
- If local array, it’s deallocated when function returns/completes

```
int *i = new int; int *iArray = new int[10]
```

- Programmers’ responsibility to deallocate memory when no longer needed
- Memory leaks occur if memory not deallocated. Exists even after function returns/completes

What if there’s not enough memory?

- The **new** operator will throw an exception

- We can avoid dealing with exceptions by using the **nothrow** object:
  1. Indicates that it will not throw an exception on failure, but return a *null pointer* instead
  2. Include the <**new**> header
  3. Pass **nothrow** as an argument for new

```
int * my_pointer = new (nothrow) int [5];
if (my_pointer == nullptr) {
     // DO SOMETHING
}
```

The delete keyword

- We must remember to free the allocated memory
- If we don't, we get a **memory leak**
- There is no garbage collector in C++
- We must remember to deallocate the memory

- We can do this with the **delete** and **delete[]** operators

The delete keyword

```
int *i = new int;
int *iArray = new int[10]

...//some code

delete i; //free allocated memory
delete[] iArray; // freed block of allocated memory
```

ACTIVITY

1. What is inside each cell in a dynamically created array of a fundamental type?
2. What is inside each cell in a dynamically created array of string?
3. How big is the largest array of long you can dynamically allocate on YOUR laptop? Write a program that asks the user to enter a long. Pass the long to a function that dynamically allocates an array of int, and returns a pointer to it to the main function. In the main function, delete[ ] the array to free up all that memory, and ask the user to enter a larger number. Keep going until you determine the limit.

STRUCTS AND UNIONS

User-Defined Types: C++ has structs too!

```
struct type_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;

} object_names;
```

Where:

`type_name` is the name for the struct

`object_names` is an optional list of declared objects

User-Defined Types: C++ has structs too!

```
struct product{
     int weight;
     double price;
};

struct product{
     int weight;
     double price;
} apple, banana, melon;
```

The union

- The union allows a **single portion of memory** to be accessed as different types
- Its declaration and use is similar to a struct's:

```
union unionName{
    char characters[4];
    int integer;
} my_union;
```

- This union is 4 bytes in size. We can access those 4 bytes in 2 different ways: **my_union.characters** is an array of char, and **my_union.integer** is (wait for it) an int!

The union

- Can declare union with member of different byte size

```
union unionName{
        char c;
        int i;
        float f
} my_union;
```

- All members will occupy same space in memory, but size of this union type with be the largest member

The union

```
union unionName{
       char characters;
       int integer;
   } my_union;
   my_union.integer = 40;

   cout << my_union.integer << endl;
   cout << my_union.characters << endl;

   my_union.integer = 67;
   cout << my_union.integer << endl;
   cout << my_union.characters << endl;
```

union.cpp

The union

- Imagine creating a struct that can transform into 3 types of numbers but doesn't take up byte size of all 3 types
  - Although it can potentially be 3 types of numbers, it can only be 1 type at a time
- Need a way to toggle between type of value being used

```
union integralUnion{
    short short_value;
    int int_value;
    long long_value;
};
```

```
struct integrals {
    int type;
    integralUnion iu;
};
```

union2.cpp

THE C++ VECTOR

The C++ vector (think ArrayList)

- In **<vector>**
- A sequence container that **can change size** (like Java's ArrayList)
- Part of the STL (which we will cover in a few weeks)
- But for now it's very useful, even without knowing how to use its iterators
- http://www.cplusplus.com/reference/vector/vector/
- http://en.cppreference.com/w/cpp/container/vector

The C++ vector (think ArrayList)

- There are some very useful member functions:
  - **push_back(const T& value)** appends the given value to the end
  - **size()** //returns number of elements in vector
  - **operator[size_type pos]** returns a reference to the element at pos
  - **at(size_type pos)** returns a reference to the element at pos. Differs from operator[] by doing bounds check and throws exception
- We can use the for-each loop with the vector (it's called the **ranged-for** in C++)

The C++ vector (think ArrayList)

```
Vector <int> intVector;
intVector.push_back(5);
intVector.push_back(10);
intVector.push_back(15);
```

```
for(int i=0; i<intVector.size(); i++)
{
    cout << intVector[i]
}
```

The C++ vector (think ArrayList)

```
Vector <int> intVector;
intVector.push_back(5);
intVector.push_back(10);
intVector.push_back(15);
```

```
for(int i=0; i<intVector.size(); i++)
{
    cout << intVector[i]
}
```

```
for(int value: intVector)
{
    cout << value
}
```

vector.cpp

ACTIVITY

1. Modify your top 10 word counter from before:
    1. Define a struct called **word** that contains a string that represents a word (always in lower case!) and an int to represent its frequency
    2. Use an **std::vector<word>** to store the words and their frequencies
    3. Is your program faster or slower than before, assuming no other changes are made?