CS100
Introduction to Programming

Lecture 14. C++ Warm Up 2

Declaring references

- References are a new data type in C++
  - char c;   // a character
  - char *p = &c; // a pointer to a character
  - char& r = c;   // a reference to a character
- Local or global variables
  - *Type& reference_name* = variable;
  - References must be initialized

References

- A reference is an alias (of an existing variable).

```
int  X = 47;
int& Y = X; // Y is a reference to X

// X and Y now refer to the same variable
cout << "Y = " << Y; // prints Y = 47
Y = 18;
cout << "X = " << X; // prints X = 18
```

Rules of References

- References must be initialized when defined
- Initialization establishes a binding
  - In declaration

```
int x = 3;
int& y = x;
const int& z = x;
```

  - As a function argument

```
void f(int &x);
f(y); // initialized when function is called
```

Rules of References

- Bindings don't change at run time (unlike pointers)
- Assignment changes the object referred-to

```
int& y = x;
y = 12;        // change value of x
```

- The target of a reference must have an identity!

```
void func(int &x);
func(3);       // Error!
func(i);       // Correct
func(i * 3);         // Error!
```

Pointers vs. References

- References

a) can’t be null
b) are dependent on an existing variable, (they are an alias for an variable)
c) can’t bind to another variable after initialization

- Pointers

a) can be set to `nullptr`
b) Pointer is independent of existing objects
c) can point to another variable

Restrictions

- A reference must bind to an existing object, but the reference itself is not an object.
  - A pointer must also point to an existing object.
  - A pointer is also an object itself.
- No references to references
- No pointers to references

```
int& *p ;    // illegal
```

  — Reference to pointer is OK

```
void f(int*& p);
```

- No arrays of references

Reference in range-for

- Change lowercase letters to uppercase in a string

```
std::string str = "AbcaBC";
for (char &c : str)
  c = std::toupper(c);
```

- If c is not a reference, it will become **copies** of the characters in str.
  - Modifying c has no effect on the contents of str.

Return references

- Functions can return references

  (But they should refer to non-local variables ……)

```
const int SIZE = 32;
double myarray[SIZE];
double& subscript(int i){
    return myarray[i];
}
```

Return references

- Functions can return references

  (But they should refer to non-local variables ......)

```
int main(){
  for (int i = 0; i < SIZE; i++){
     myarray[i] = i * 0.5;
  }

  double value = subscript(12);
  subscript(3) = 12.345;
}
```

Pass by reference-to-const

- Avoiding copy:

```
void print_thing(BigType something) { /* ... */ }
void print_thing_2(BigType& something) { /* ... */ }


print_thing(x); // BigType something = x; (a copy)
print_thing_2(x); // BigType& something = x; (no copy)
```

- However, this parameter declaration refuses const arguments.
- Worse still, what if the function modifies `something` in its body?

Pass by reference-to-const

- To avoid copying and ensure the parameter does not get changed:

```
void print_thing_good(const BigType& something)
  { /* ... */ }
```

- This also accepts const and/or non-entity values (eg. literals, temporary objects...)

The C++ Standard Template Libraries

- In 1990, Alex Stepanov and Meng Lee of HP Laboratories extended C++ with a library of class and function templates which has come to be known as the STL
- In 1994, STL was adopted as part of ANSI/ISO Standard C++

The C++ Standard Template Libraries

- STL had three basic components:
  - Containers
    - Generic class templates to store data
  - Algorithms
    - Generic function templates to operate on containers
  - Iterators
    - Generalized ‘smart’ pointers that facilitate use of containers
    - They provide an interface that is needed for STL algorithms to operate on STL containers
  - **String abstraction was added during standardization**

Why use STL?

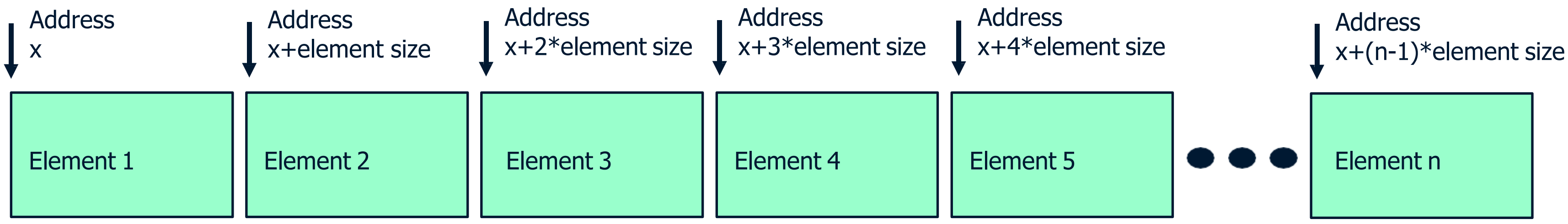
- STL
  - offers an assortment of containers
  - releases containers' time/storage complexity
  - containers grow/shrink in size automatically
  - provides built-in algorithms to process containers
  - provides iterators that make the containers and algorithms flexible and efficient.
  - is extensible which means that users can add new containers and new algorithms such that
    - algorithms can process STL containers as well as user defined containers
  - User defined algorithms can process STL containers as well as user defined containers

Standard Template Library

- Uses template mechanism for generic …
  - … containers (classes)
    - Data structures that hold **anything**
    - **Ex.:** `vector` (today!), `list`, `map`, `set`
  - … algorithms (functions)
    - handle common tasks (searching, sorting, comparing, etc.)
    - **Ex.:** `find`, `merge`, `reverse`, `sort`, `count`, `random shuffle`, `remove`, `nth-element`, `rotate`, …

Vector

- An alternative to the built in array
  - Use it instead!
- A vector is self grown (dynamic in size)
- Contiguous placement in memory



Using Vector

- `#include <vector>`

- `using std::vector;`
  - Enables using by **`vector`** without **`std::`**

- Two ways to use the vector type:
  - Array style
  - STL style (modern C++ style, recommended)

Declaring a new vector

- Syntax:
  - `std::vector<of what>`

- For example (`using std::vector;`):
  - `vector<int>` - vector of integers
  - `vector<string>` - vector of strings
  - `vector<int*>` - vector of pointers to integers
  - `vector<Shape>` - vector of Shape objects, where Shape is a user defined struct or class

Using a Vector – Array Style

- Similar to using C-style arrays
- Use the operator[] with an index to access an element.

```
void simple_example()
{
    const int N = 10;
    vector<int> ivec(N);
    for (int i = 0; i < N; ++i)
        cin >> ivec[i];
}
```

- STL style is more recommended!

Using a vector – STL style

- We declare an empty vector

```
vector<string> svec;
```

- Insert elements into the vector using the method push_back

```
string word;
while ( cin >> word ) //# words "unlimited"
{
    svec.push_back(word);
}
```

Insertion

```
void push_back(const T& x);
```

- Inserts an element with value x at the end of the container
- Elements must be of the same type as declared. For a `vector<int>`, type `T` is `int`.
  - Example:

    ```
    vector<string> svec;
    svec.push_back(str);
    ```

Size

```
std::size_t size() const;
```

- Returns the length of the container (how many items it contains)
- C arrays require manually recording the size. Not anymore for C++ vectors!
  - Example

    ```
    size_t size = svec.size();
    ```

Going Through a Vector – Array Style

- Still your familiar C-style for-loop and operator[].

```
for (size_t i = 0; i < ivec.size(); ++i) {
  cout << ivec[i] << endl;
}
```

Going Through a Vector – STL Style

- Use a range-based for-loop!

```
vector<int> ivec = {1, 3, 5};
for (int i : ivec) {
    cout << i << endl;
}
```

Going Through a Vector – STL Style

- What if we want to modify each element?

```
vector<int> ivec = {1, 3, 5};
for (int i : ivec) {
    i *= 10;
}
// No effect! ivec is still {1, 3, 5}!
```

- Because `i` accesses each element by **value**.

References and Range-based For

- What if we want to modify each element?
- Use a **reference** in range-based for

```
vector<int> ivec = {1, 3, 5};
for (int& i : ivec) {
    i *= 10;
}
// ivec becomes {10, 30, 50}!
```

References and Range-based For

- When modification is not needed
  - For built-in types, passing by value is fine
  - For other types, **reference-to-const** is better

```
for (BigType something : bigvec) {
    // Actually copies every element!
}
for (const BigType& something : bigvec) {
    // No copying, better performance
}
```

More about Vectors

- More operations:
  - `bool empty() const;`
  - `void clear();`

  `...`
- Iterators (special "Generalized pointers" for STL):
  - `iterator begin();`
  - `iterator end();`
  - `iterator erase(iterator it);`

  `...`
- Algorithms(in header `<algorithm>`):
  - `void std::sort(iterator first, iterator last);`
  - `iterator std::find(iterator first, iterator last, const T& value);`

  `...`

Let's revisit STL in later lectures!

Putting it all together

```
int main() {
    int input;
    vector<int> ivec;

    // input
    while (cin >> input)
        ivec.push_back(input);

    // modify
    for (int& i : ivec)
        i = i > 0 ? i : -i;

    // sort (in header <algorithm>)
    std::sort(ivec.begin(), ivec.end());

    // output
    for (int i : ivec)
        cout << i << " ";
    cout << endl;

    return 0;
}
```

new and **delete**

- Better ways of allocating/deallocating dynamic memory in C++ (alternates to malloc/free).
- Type *ptr = new Type;
- Type *arr = new Type[n];
- delete ptr;
- delete []arr;

new and **delete**

- To avoid memory leak, match any new with a delete, any new [] with a delete[]!
- delete a pointer created by malloc:
  – Undefined Behavior
- free a pointer created by new:
  – Undefined Behavior
- delete (without []) a pointer created by new []:
  – Undefined Behavior