Lecture 4: C++ Basics

Value Category

Naïve Value Categories

L/R-value refers to memory location which identifies an object. A simplified definition for beginners are as follows.

- L-value may appear as **either left hand or right hand** side of an assignment operator(=).
  - In memory point of view, an l-value, also called a *locator value*, represents an object that occupies some identifiable location in memory (i.e. has an address).
  - Any non-constant variable is lval.
- R-value may appear as **only right hand** side of an assignment operator(=).
  - Exclusively defined against L-values.
  - Any constant is an rval.

Consider the following code.

```
int main(){
    int arr[5] = {0, 1, 2, 3, 4};
    int *ptr1 = &arr[0];
    int *ptr2 = &(arr[0]+arr[1]);
    cout << ptr1 << endl;
    cout << ptr2 << endl;
    arr[0] = 5;
    arr[0] + arr[1] = 5;
}
```

The compiler will raise errors.

```
...\main.cpp:7:32: error: lvalue required as unary '&' operand
     int *ptr2 = &(arr[0]+arr[1]);
                                ^
...\main.cpp:11:21: error: lvalue required as left operand of assignment
     arr[0] + arr[1] = 5;
```

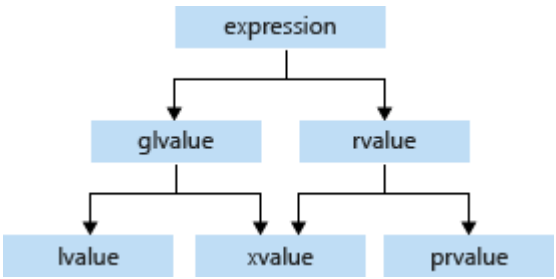
Deep Dive into Value Categories (*Optional*):

Consider the following code.

```cpp
int main(){
    vector<int> v = {0, 1, 2, 3, 4};
    auto output_iterator = back_inserter(v);
    *output_iterator = 5;
    cout << *output_iterator << endl;
}
```

- What's the output of this function? Or will this program compile?
- Is `*output_iterator` an l-value or r-value? Why?

Read L-values and R-values and take EECS 483, Compiler Structure, if interested.



Function Declaration and Definition

Consider the following codes.

```cpp
// Function Declaration
int getArea(int length, int width);

int main()
{
    cout << getArea(2, 5) << endl;
}

// Function Definition
int getArea(int length, int width)
{
    return length*width;
}
```

Function Declaration

Function declaration (prototype) shows how the function is called. It must appear in the code before function can be called.

A Function declaration `int getArea(int length, int width);` tells you about:

- Type signature:
  - Return type: `int`;
  - # arguments: 2;
  - Types of arguments: `int`*2;
- Name of the function: `getArea`;

- Formal Parameter Names (*): `length` and `width`.

However, formal parameter names are not necessary. Try to replace:

```
int getArea(int length, int width);
```

with:

```
int getArea(int l, int w);
```

or even:

```
int getArea(int, int);
```

The program still works. Yet, it is considered good coding style to keep the formal parameter names, so that your potential collaborators can understand what the function is for.

Function Definition

Function definition describes how a function performs its tasks. It can appear in the code before or after function can be called.

```
int getArea(int length, int width)  // ----> Function Header
{                                   // -+
    return length*width;            //  |--> Function Body
}                                   // -+
```

Argument Passing Mechanism

Consider the following example.

```
void pass_by_val(int x){
    x = 2;
}

void pass_by_ref(int &x){
    x = 2;
}

void mixed(int x, int &y){
    x = 3;
    y = 3;
}

int main(){
    int y = 1;
    int z = 2;
    pass_by_val(y);
    pass_by_ref(z);
    cout << y << " " << z << endl;
```

```
    mixed(y, z);
    cout << y << " " << z << endl;
}
```

The output of the above code is

```
1 2
1 3
```

This example demonstrates the 2 argument passing mechanism in C++:

- Pass by Value;
- Pass by Reference.

The difference of above mechanisms can be interpreted from the following aspects:

- Language point of view: reference parameter allows the function to change the input parameter.
- Memory point of view:
  - Pass-by-reference introduce an extra layer of indirect access to the original memory object. In fact, many compilers implement references with pointers.
  - Pass-by-value needs to copy the argument.
  - Can both expensive, in terms of memory and time.

Choosing argument passing methods wisely:

- Pass atomic types by value (`int`, `float`, `char*`...).
  - `char` is 1B, `int32` is 2B, `int64` is 4B, and a pointer is 8B (x64 System).
- Pass large compound objects by reference (`struct`, `class`...).
  - For `std` containers, passing by value costs 3 pointers, but passing by reference costs only 1;
  - What about structures/classes you created?

Arrays and Pointer

Your familiarity of arrays and pointers is assumed in this course. Test yourself with the following examples.

Pointers

What is the output of the following example?

```
int main(){
    int x = 1;
    int y = 2;
    int *p = &x;
    *p = ++y;
    p = &y;
    y = x++;
    cout << x << " " << y << " " << *p << endl;
}
```

Arrays

What is the output of the following example?

```
void increment(int arr[], int size){
    for (int i = 0; i < size; ++i){
        (*(arr + i))++; // correct
        //*(arr + i)++; // wrong
    }
}

int main(){
    int arr[5] = {0, 1, 2, 3, 4};
    increment(arr, 5);
    for (int i = 0; i < 5; ++i){
        cout << *(arr + i);
    }
    cout << endl;
}
```

Two important things to keep in mind here:

- Arrays are naturally passed by reference;
- Conversion formula between arrays and pointers: `*(arr + i) = arr[i]`

References and Pointers

L-vals always corresponds to a fixed memory region. This gives rises to a special construct called references.

Your familiarity of **non-constant references** is assumed in this course. Test yourself with the following example. What is the output?

```
int main(){
    int a = 1;
    int b = 5;
    int *p1 = &a;
    int *p2 = &b;
    int &x = a;
    int &y = b;
    p2 = &a;
    x = b;
    a++;
    b--;
    cout << x << " " << y << " " << *p1 << " " << *p2 << endl;
}
```

Non-constant references must be initialized by a variable of the same type, and cannot be rebounded. Try resist the temptation to think reference as an alias of variables, but remember they are alias for the memory region. References must be bind to a memory region when created: there is no way to re-bind of an existing reference.

Structures

Your familiarity of structures is assumed in this course.

```cpp
struct Student{
    // represents a JI student.
    string name;
    string major;
    long long stud_id;
    bool graduated;
};

int main(){
    // Initialize a structrue
    struct Student s = {"martin", "undeclared", 517370910114, false};
    struct Student *ptr = &s;

    // Use . and -> notation to access and update
    cout << s.name << " " << ptr->stud_id << endl;
    s.major = "ece";
    ptr->graduated = true;
}
```

Two facts to take away here:

- `struct` is in fact totally the same as `class`, instead the default is `public`.
- To save memory, place larger attributes first. You will understand and get used to this when you take VE/EECS 281 and VE/EECS 370.

Undefined Behaviors

Undefined behaviors (UB) are program whose output depends on a specific platform, or a specific implementation of the compiler. Therefore, the outcome/answer of a process may be YES, NO...or UB.

Why UBs are still there throughout the years? It's not that the committee doesn't know how to eliminate them, but they leave room for compiler optimizations.

For a programmer, keep in mind that:

- It's an absolute waste of time trying to figure out what will happen given an code that contains UB.
- It's dangerous and to write code that contains UB, and it is your job to avoid them.

Any (zero or more) of the following may happen if you trigger any of UBs:

- Compiler refuse to compile;
- Compiler compiles but warns (Again, warnings are bugs);
- The compiler compiles silently, but...
    - Program crashes when executed;
    - Your program works out, but the output is wrong;
    - The compiler deletes your files in the systems;
    - Your program works perfectly on your computer but gets rejected by the JOJ.

Common examples of UBs:

- Integer overflow:

```
int x = INT_MAX;
x++; // UB
```

  Many compilers would work out `x = -2147483648`, but that is not guaranteed.

- Dereferencing `nullptr`:

```
int* x = nullptr;
*x = 1; // UB
```

  Most compilers compile it, but the program crashes. Yet, it is not guaranteed that the program crashes.

- Array out-of-bound:

  For most platforms, the following works perfectly, but is bad and not guaranteed to.

```
int main(){
    int x[5] = {0};
    x[5] = 1; // UB
    cout << x[5] << endl;
}
```

  Merely taking the address is UB.

```
int main(){
    int x[5] = {0};
    int *p = &(x[5]); // UB
    *p = 1;
    cout << *p << endl;
}
```

- Dangling references

  You could still get correct value, or not.

```
int main(){
    auto x = new int[10];
    x[3] = 1;
    delete[] x;
    cout << x[3]; // UB
}
```

There are more UBs out there. Please be aware and remember them when you encounter any.

Lecture 5: `const` Qualifier

Constant Modifier

Immutability of `const`

Whenever a type something is `const` modified, it is declared as immutable.

```
struct Point{
    int x;
    int y;
};

int main(){
    const Point p = {1, 2};
    p.y = 3; // compiler complains
    cout << "(" << p.x << "," << p.y << ")" << endl;
}
```

Remember this immutability is enforced by the compiler at compile time. This means that `const` in fact does not guarantee immutability, it is an intention to. The compiler does not forbid you from changing the value intentionally. Consider the following program:

```
int main(){
    const int a = 10;
    auto *p = const_cast<int*>(&a);
    *p = 20;
    cout << a << endl;
}
```

What's the output? This is actually an UB. It depends on your compiler and platform whether the output is 10 or 20. Therefore, be careful of undefined behaviors from type casting, especially when `const` is involved.

Const Global

Say when we declare a string for jAccount username, and want to ensure that the max size of the string is 32.

```
int main(){
    char jAccount[32];
    cin >> jAccount;
    for (int i = 0; i < 32; ++i){
        if (jAccount[i] == '\0'){
            cout << i << endl;
            break;
        }
    }
}
```

This is bad, because the number 32 here is of bad readability, and when you want to change 32 to 64, you have to go over the entire program, which, chances are that, leads to bugs if you missed some or accidentally changed 32 of other meanings.

This is where we need constant global variables.

```
int MAX_SIZE = 32;

int main(){
    char jAccount[MAX_SIZE];
    cin >> jAccount;
    for (int i = 0; i < MAX_SIZE; ++i){
        if (jAccount[i] == '\0'){
            cout << i << endl;
            break;
        }
    }
}
```

Note that const globals must be initialized, and cannot be modified after. For good coding style, use UPPERCASE for const globals.

Const References

Const Reference vs Non-const Reference

There are many ways to define a constant reference, which are all identical.

- `const int& iref`
- `(const int)& iref`
- `int& (const iref)`
- `const int& (const iref)`

There is something special about const references:

- Const reference are allowed to be bind to right values;
- Normal references are not allowed to.

Consider the following program. Which lines cannot compile?

```
1  int main(){
2      // Which lines cannot compile?
3      int a = 1;
4      const int& b = a;
5      const int c = a;
6      int &d = a;
7      const int& e = a+1;
8      const int f = a+1;
9      int &g = a+1;       // x
10     b = 5;              // x
11     c = 5;              // x
12     d = 5;
13 }
```

Normally, if a const reference is bind to a right value, the const reference is no difference to a simple const. In above example, you may consider line 4 and 5 identical.

Argument Passing

Then why do we need const references? See the following example.

```
1  class Large{
2      // I am really large.
3  };
4
5  int utility(const Large &l){
6      // ...
7  }
```

Reasons to use a constant reference:

- Passing by reference -> avoids copying;
- `const` -> avoids changing the structure;
- const reference -> rvals can be passed in.
    - That's why we don't use a const pointer.

Const Pointers

There are many ways to define a `const` reference, which are NOT identical. Personally my trick of identification is to read from the right hand side.

- Pointer to Constant (PC): `const int *ptr`;
- Constant Pointer (CP): `int *const ptr` or `int *(const ptr);`
- Constant Pointer to Constant (CPC): `const int *const ptr` or `const int *(const ptr)`.

Type	Can change the value of pointer?	Can change the object that the pointer points to?
Pointer to Constant	Yes	No
Constant Pointer	No	Yes
Constant Pointer to Constant	No	No

See the following example. Which lines cannot compile?

```
int main(){
    // Which lines cannot compile?
    int a = 1;
    int b = 2;
    const int *ptr1 = &a;
    int *const ptr2 = &a;
    const int *const ptr3 = &a;
    *ptr1 = 3;      // x
    ptr1 = &b;
    *ptr2 = 3;
    ptr2 = &b;      // x
    *ptr3 = 3;      // x
    ptr3 = &b;      // x
}
```

Const and Types

Type Definition

When some compound types have long names, you probably don't want to type them all. This is when you need `typedef`.

The general rule is `typedef real_name alias_name`.

For example, you probably want:

```
typedef std::unordered_map<std::string, std::priority_queue<int, std::vector<int>,
std::greater<int> > > string_map_to_PQ;
```

Mind that you can define a type based on a defined type. See following example. Which lines cannot compile?

```
typedef const int_ptr_t Type1;
typedef const_int_t* Type2;
typedef const Type2 Type3;

int main(){
    // Which lines cannot compile?
    int a = 1;
```

```
    int b = 2;
    Type1 ptr4 = &a;
    Type2 ptr5 = &a;
    Type3 ptr6 = &a;
    *ptr4 = 3;
    ptr4 = &b;      // x
    *ptr5 = 3;      // x
    ptr5 = &b;
    *ptr6 = 3;      // x
    ptr6 = &b;      // x
}
```

Type Coercion

The following are the **Const Prolongation Rules**.

- `const type&` to `type&` is incompatible.
- `const type*` to `type*` is incompatible.
- `type&` to `const type&` is compatible.
- `type*` to `const type*` is compatible.

In one word, only from non-const to const is allowed.

Consider the following example:

```
void reference_me(int &x){}
void point_me(int *px){}
void const_reference_me(const int &x){}

void main() {
    int x = 1;
    const int *a = &x;
    const int &b = 2;
    int *c = &x;
    int &d = x;

    // Which lines cannot compile?
    int *p = a;             // x
    point_me(a);            // x
    point_me(c);
    reference_me(b);        // x
    reference_me(d);
    const_reference_me(*a);
    const_reference_me(b);
    const_reference_me(*c);
    const_reference_me(d);
}
```

Credit

SU2019 & SU2020 VE280 Teaching Groups.

VE 280 Lecture 4-5