VE280 Recitation Class Notes

VE280 SU19 TA Group

UM-SJTU Joint Institute

June 4, 2019

Slides credit: Yue Yao

Table Of Contents I

RC Week 4

Building a C++ program

The problem of *building* complex programs

Building is different from *compiling*

- Compiling refers to the process of translating code to binaries.
- Building is **piecing together** from its components.
- A program might depend on other package.
- A program might use a pre-compiled library.
- A program might involve more than one source files.
- A program might need to be built for different platform.
- Sometimes you not only needs to build just one executable, but also documentations / test suites / libraries for the sake of other programs.

How complicated is Linux kernel version 3.2?

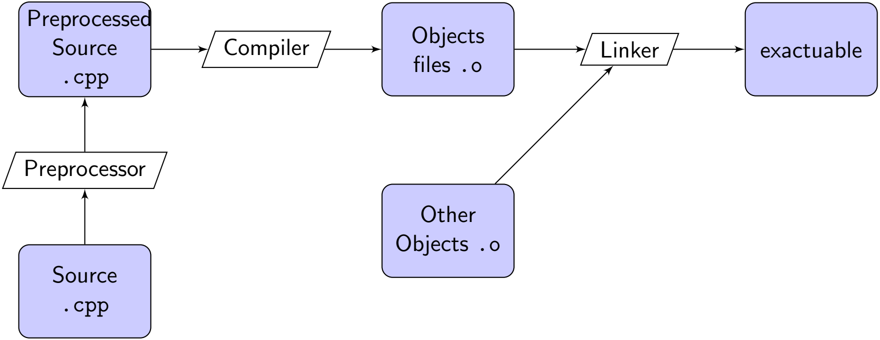
- 37,626 The number of files
- 15,004,006 The number of lines of code

Building a multi-file C++ program

The golden rule

Each source file (`.cpp`, `.c`) compiles independently.

The *building* process



The g++ tool chain

g++ as a all-in-one tool

- Preprocessor, compiler and linker used to be separate.
- Now g++ combines them into one.
- By default g++ takes source files and generate executable.
- Using different switches you can perform individual step.

Options for g++

- **-o out** Name the output file as `out`. Outputs `a.out` if not present.
- **-std=** Specify C++ standard. Recommend `-std=c++11`.
- **-Wall** Report all warnings.
- **-O{0123}** Optimization level. `-O2` is the recommended for release.
- **-c** Only compiles the file (Can not take multiple arguments).
- **-E** Only pre-processes the file (Can not take multiple arguments).

An example

This example contains a “main” source file accompanied with multiple other source files. All files are compiled separately into object files. We link some of them together and see what happens.

Keep in mind variables/function must be first declared before used.

−− > code/rc4build/main.cpp

```
#include <iostream>
using namespace std;
extern int number[], size;
int reduce(int n[], int s);
int main() {
   for (int i=0; i<size; i++) cout << number[i] << " ";
   cout << "\nReduced to " << reduce(number,size) << endl;
}
```

An example

$-->$ code/rc4build/odd.cpp

```
int number[] = {1, 3, 5, 7, 9};
int size = sizeof(number) / sizeof(*number);
```

$-->$ code/rc4build/even.cpp

```
int number[] = {2, 4, 6, 8, 10};
int size = sizeof(number) / sizeof(*number);
```

$-->$ code/rc4build/sum.cpp

```
int reduce(int number[], int size) {
    int sum = 0;
    while (--size) sum += number[size];
    return sum;
}
```

An example

$-->$ code/rc4build/prod.cpp

```
int reduce(int number[], int size) {
    int prod = 1;
    while (--size) prod *= number[size];
    return prod;
}
```

The following file is a C source file. This file is given just for you to know you can do pretty weired things if you know the deal.

$-->$ code/rc4build/sum_large.c

```
int _Z6reducePii(int* number, int size) {
    int sum = 0;
    while (--size)
        if (number[size] > 3) sum += number[size];
    return sum;
}
```

An example

We compile the source files one by one.

```
$ g++ -o main.o -c main.cpp
$ g++ -o odd.o -c odd.cpp
$ g++ -o even.o -c even.cpp
$ g++ -o sum.o -c sum.cpp
$ g++ -o prod.o -c prod.cpp
```

Next one is compiled through gcc

```
$ gcc -o sum_large.o -c sum_large.c
```

Next step we are going to link (some of) them and execute it.
Linking in g++ is easy. If you supply `.o` files, g++ will know that is should link them instead of compiling them.
Pay extra attention to compiler errors (actually linker errors), they are the most interesting part.

An example

Now first standard examples

```
$ g++ -o main main.o even.o sum.o && ./main
$ g++ -o main main.o even.o prod.o && ./main
$ g++ -o main main.o odd.o prod.o && ./main
```

Now what if we link both `even.o` and `odd.o`

```
$ g++ -o main main.o odd.o even.o prod.o && ./main
```

Now what if we link both `prod.o` and `sum.o`

```
$ g++ -o main main.o odd.o sum.o prod.o && ./main
```

Now what if we leave out both `even.o` and `odd.o`

```
$ g++ -o main main.o prod.o && ./main
```

Now what if we leave out the `main.o`

```
$ g++ -o main even.o prod.o && ./main
```

An example

Surprises

Now we introduce something crazy. The name of the function in `sum_large.c` is really strange. But we just ignores that link its object file any way.

```
$ g++ -o main main.o even.o sum_large.o && ./main
```

Well it worked. The question is how on earth can this work. In fact g++ is doing some crazy renaming when compiling your source code. The reason why they did this is understandable when you think about it in the later period of the course.

Understanding linking actually allows you to do some crazy things. Try compiling the following file (with only one line of code) on your machine.

```
int main[-1u] = {1};
```

It tooks quite long to finish. How large is the executable?

Headers and inclusion

`#include<>` : Why we need them?

- Things must be declared before used.
- Each source file compiles independently. Needs a method to "export" functions defined in one file to other files.
- Avoid repeating declarations.

Preprocessing

- Preprocessing is purely **textual**.
- `#include` simply copy the content.
- *Conditional compilation directives* simply deletes unused branch. (`#ifdef`, `#ifndef`, `#else`, ...)

Header guards

problem

Whenever there is dependence of source files, there will be dependence of headers.

Consider the following `a.cpp`, `a.h`, `b.h` and `c.h`. Keep in mind that *everything in C++ is allowed to have at most 1 definition during compilation.*

$-- >$ a.cpp

```
#include "a.h"
#include "b.h"
int main() {...}
```

$-- >$ point.h

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

$-- >$ a.h

```
#include "point.h"
int area(Point a, Point b);
```

$-- >$ b.h

```
#include "point.h"
void circle(Point o, int r);
```

Header guards

Solution

The idea is to use a unique macro to guard a header.

- Define that unique macro when the header is first included.
- Check if the macro is defined in future inclusion.

Now `point.h` becomes:

```
#ifndef _POINT_H_
#define _POINT_H_
struct Point {int x, y;}
#endif
```

The macro could be something else. Just don't use something common.

Build systems

The need for a build system

- Build process is complicated, avoid type every command.
- Project have dependence, need to manage dependence
- Compile minimum amount of code possible upon update.
- Many other reasons, abstract out actual compiler, compile for different platform / target.

Choices of build systems

GNU/make: Our choice of make system. It has a very long history.

CMake: A modern make system used by CLion and many other projects. Very flexible and reliable. It is also a cross platform solution.

Makefile and it's syntax

The *Makefile*

- The executable for GNU/make is simply `make`
- `make` requires a file that describes the building process. Such file is named `Makefile`.
- `Makefile` is made up of *targets*. A target can depend on other target, or some file.

Syntax

The following syntax defines a target. Note the `tab` key.

```
TargetName :  Dep1 Dep2 file1.o file2.o
	Command1-to-run
	Command2-to-run
```

Makefile : Example

This is a `Makefile` for our previous example. $-->$ `code/rc4build/Makefile`

```
all : sum_even
sum_even : objects
        g++ -o run main.o even.o sum.o
prod_odd : objects
        g++ -o run main.o prod.o odd.o
clean :
        rm -f *.o && rm -f ./run
onestep : main.cpp even.cpp sum.cpp
        g++ -o run main.cpp even.cpp sum.cpp
objects : sum.cpp prod.cpp even.cpp odd.cpp main.cpp
        g++ -c sum.cpp && g++ -c prod.cpp
        g++ -c even.cpp && g++ -c odd.cpp
        g++ -c main.cpp
```

CMake

CMake stands for Cross-platform Make. You're required to use CMake if you're using CLion.

The build process has one step if you use a Makefile, namely typing make at the command line. For CMake, there are two steps[1]:

1. Setup your build environment.
2. Perform the actual build in the selected build system.

Commands:

```
mkdir build && cd build
cmake ..
make
```

[1] https://prateekvjoshi.com/2014/02/01/cmake-vs-make/

CMakeList.txt

Example CMakeList.txt

```
cmake_minimum_required(VERSION 3.8)
project(surfaces)
set(CMAKE_CXX_STANDARD 11)
find_package(OpenCV REQUIRED)
set(CMAKE_CXX_FLAGS_RELEASE
 ↪  "${CMAKE_CXX_FLAGS_RELEASE} -march=native
 ↪  -ffast-math")
add_executable(main main.cpp utils.cpp)
target_link_libraries(main ${OpenCV_LIBS})
```

RC Week 4

Review of C++

Review of C++

- Standardized C++ and Undefined Behaviors.
- Declaration versus Definition.
- lval versus rval.
- References.
- Function argument passing.
- const modifier.
- Function pointers.

Standardized C++

Once upon the time, programming languages are just conventions, design choices made by the language creator.

The standardize process

- Establishes program syntax, what are acceptable and what are syntax errors?.
- Language semantics, what's the "meaning" of an expression / language construct.
- Behavior, what are the expected behavior and what are undefined and left to the choice of compilers ...
- Standard library, what to include and what's the implementation constraint.

The latest standard is C++17 (3/21/2017). Major standards are C++98, C++03, C++11, C++14. C++ after C++11 is generally considered "modern C++".

Online reference for `std::to_string()`

The following information comes from
`http://www.cplusplus.com/reference/string/to_string/`

Figure: Online reference for `C++11` library function `to_string`

- Notice the `C++11` sign.
- Notice the overloads supported by this function.

Undefined Behaviors

One outcome from the standardize process is that, almost every true-or-false question about the C++ program can be answered with one of the following decisively. It is either YES, NO, or more importantly **undefined behavior** (*UB* for short).
Undefined behaviors are program whose output depends on a specific platform, or a specific implementation of the compiler.
You should always remember the following:

- 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.
- Anyone who test you with UB, is both stupid and ignorant.

There is a reason why UB exists. It's not that the committee doesn't know how to eliminate them, but they leave room for pretty impressive *compiler optimizations*.

Undefined Behaviors

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

- The compiler may refuse to compile.
- The compiler still compiles, but throw you an warning
- The compiler compiles silently.
- Your program crashes when executed.
- Your program malfunctions when executed.
- The compiler deletes all your photos.
- 72 fairies come out of your screen and dance around you.
- **Your program works perfectly.**

It's your job to avoid UB. We may refuse to answer the "why my program works locally but crashes on OJ" type of question.

Undefined Behaviors

Common cases

- Integer overflow (No, it's not guaranteed to be negative!)

```
int x = INT_MAX; x++;
```

- Dereferencing `nullptr` (No, it's not guaranteed to be crash!)

```
int* x = nullptr; *x = 2;
```

- Array out-of-bound (Even taking address is UB!)

```
int x[10] = {0};  x[10] = 1; int* x = &(x[11]);
```

- Dangling references (You could still get correct value)

```
int* x = int[10]; x[3] = 5; delete[] x; cout << x[3];
int* f(int t) {return &t;} int* x = f(10); cout << *x;
```

Declaration versus *Definition*

Consider writing a declaration for the following add function.

```
int add(int x, int y) {return x + y;}
```

No doubt above is a function definition. We first write

```
int add(int x, int y);
```

Well, that's a right answer. But we could also do

```
int add(int elephant, int haskell);
```

Suprised? Well it makes sense since changing formal arguments doesn't change the function at all! $f(x) = x$ and $f(z) = z$ are the same function. But we could push this even further!

```
int add(int, int);
```

This will work as well.

lval and *rval*

Compare the following 2 expression, suppose arr is an array of integers

1. `arr[10]`
2. `arr[10] + arr[1]`

They both have the type `int` of course.

- `int *p = &(arr[10]);` makes sense.
- `int *p = &(arr[10] + arr[1]);` gives you compile error.

Further more

- `arr[10] = 10;` makes sense.
- `arr[10] + arr[1] = 20;` doesn't

Clearly the two expression are "different" in some sense. How? Think about memory! The first kind is called *left values* and the second is called *right values*. (Those are not technical definitions.)

References

What we discuss here applies only to non-const references!
Lvals always corresponds to a fixed memory region. This gives rises to a special construct called *references*.

```
int a = 1, b[10] = {2};
int& ra = a; int& rb3 = b[3];
a = 10; /* ra reads 10 */ ra = 20; // a reads 20
```

Think about references as aliases. Essentially, you are giving the memory region associated with `a` an extra name `ra` (memory region given by `b[3]` an extra name `rb3`).
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.

Function argument passing

Syntacticly there exists 2 ways of argument passing:

Pass-By-Value

```
int f(int x) { return (x = 2);}
```

Pass-By-Reference

```
int g(int& x) { return (x = 2);}
```

We give the following code to demonstrate their difference:

```
int y = 10; f(y); cout << y; // returns 2, outputs 10
int z = 10; g(z); cout << z; // returns 2, outputs 2
```

From a language point of view, reference parameter allows the function to change the input parameter.

Function argument passing

This memory point of view discussion give rise to some argument:

- Reference introduce an extra layer of indirect access to the original memory object, which drags down the performance.
- Pass-by-value needs to copy the argument, which can be slow.

In light of these observation, we suggest the following:

- **Small types better passed by value** (`int`, `float`, `char*`...). The cost of indirect access is much more than copying them.
- **Complicated structure better passed through reference.** (especially large ones, or class object)

On the other hand, references allows the function to change the parameter, and sometimes would like to enforce invariance of arguments. We recommend add const modifier.

const modifier

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

```
const int a = 10; a = 2; // Compile error
struct P {int x = 1, y = 2;};
const P p; p.y = 3; // Compile error
```

Remember this immutability is enforced by the compiler at compile time. This has a very strong implication. The compiler does NOT forbid you from doing strange things intentionally.

```
const int a = 10;
int *p = const_cast<int*>(&a); // C++11 style cast
*p = 20; cout << a; // Will this output 20?
```

Well this is actually UB. `const` is not a guarantee of immutability, it is an **intention**. It asks the compiler to look out for you, if you know you shouldn't change something.

const and pointers

We now combine the previous discussions.

`const int *p` — `*p` is of type `const int`, thus changing `*p` is illegal. However this declaration does **not** say anything about `p`, thus changing `p` is possible. This is called *pointer-to-const*.

`int *const p` — Equivalent to `int *(const p)`.

`int *(const p)` — This declaration essentially says if you dereference `const p`, you will get `int`. Since `int` is not quantified by `const`, you can change `*p`. However, the pointer itself, is modified by `const`, so you can change `p`. This is called *const-pointer*.

Naturally you could have `const int *(const p)`. This declaration basically says both the pointer `p` itself and the dereferenced object (`const int`) cannot be changed.

const and reference

Recall that **references cannot be rebind once initialized**. The following definitions are equivalent. They are all *const references*.

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

The second one makes the most sense, although the first one is the most commonly used.
The second one essentially says,`iref` is a reference, or an alias to a memory region, that is protected by the `const` modifier.

const reference and argument passing

There is something special about const references:

Const reference are allowed to be bind to right values,
while normal references are not allowed to.

Normally if a const reference is bind to a right value, the const reference is no difference to a simple const.

```
int a=5; const int& r=a+1; const int c=a+1;
```

In above example practically there is no difference between `r` and `c`. But when you pass arguments through const references, things become a little bit different.

```
int foo(const ReallySuperLargeStruct& s);
```

- We are passing by reference, this avoids copying.
- `const` enforces immutability.
- `rvals` can be passed directly into it (unlike pointers).

const propagation and type coercion

const modifier introduce incompatibility

The subtitle is summarized into the following 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.

Example:

```
int foo(int& x); int bar(int* px); int cfoo(const int& x);
void baz() {
    const int *q = nullptr; int *p = q; //Compile error
    const int& r = 10;  foo(r); //Compile error
    cfoo(*p); cfoo(*q); cfoo(r); // All OK
}
```

Functions Pointers: Why Pointers?

The Von Neumann View of functions

Functions are code, and code when compiled are simply binary number, i.e. data. The action of calling a function is simply pumping these binary numbers into the CPU (after you take VE370 you would find it's actually the other way around).

- Functions are just a bunch of numbers in the memory
- We could refer to the function by refering to the numbers
- These numbers has an *address* (think of arrays)
- We could use that address to refer to the function

Variable that stores the address of functions are called *function pointers*. By passing them around we could pass functions into functions, return them from functions, and assign them to variables.

Functions Pointers: Type

But there is one question, what is the type of them?
Well by our previous understanding dereferencing a function pointer should give us a function, just like dereferencing `int*` gives us `int`. We would like to do a comparison:

Function decl. foo

- `void foo();`
- `int foo(int x, int y);`
- `int foo(int, int);`
- `int *foo(int, char*);`
- `char* foo(int[], int);`

Function pointer bar

- `void (*bar)();`
- `int (*bar)(int, int);`
- `int (*bar)(int, int);`
- `int *(*bar)(int, char*);`
- `char *(*bar)(int[], int);`.

Note `int *bar(int);` does **NOT** declare a function pointer. The grouping is `int *(bar(int))`, which is declaring a function. This is related to the operator precedence of C++.

Functions Pointers: Usage

Assignment from functions

In fact, the identifier (name) of the functions are actually values of function pointers.

```
int max(int x, int y) { return x > y ? x : y;}
int (*cmp)(int, int) = max;
```

Invoking a function pointer

You can invoke a function pointer by applying operator () to it.

```
int m = cmp(10, 20); // No need to dereference it
```

Invariance under *

Dereferencing a function pointer still gives back a function pointer.

```
int m = cmp(10, 20); // 20
int n = (*cmp)(10, 20); // 20
int p = (*******cmp)(10, 20) // 20
```

Example

The following code implements a simple calculator. Notice how function pointer helps to clarify the code.

$--->$ code/rc4fptr/fptr.cpp

```
#include <iostream>
int add      (int x, int y) {return x + y;}
int subtract(int x, int y) {return x - y;}
int (*fun[])(int, int) = {add, subtract};
using namespace std;
int main() {
    int op = 1, x = 0, y = 0;
    cout << "Select your operation (1,2): "; cin >> op;
    cout << "Numbers: "; cin >> x >> y;
    cout << "ANS = " << fun[op-1](x, y) << endl;
}
```