C++

Fall 2023

Szymon Wojtulewicz

Na przykładzie swojego własnego kodu w C++ sprawdź działanie preprocesora C++

```
g++ -E -o src.E src.cpp
```

Do czego służą “puste” dyrektywy preprocesora w pliku wynikowym (pojedynczy znak #)?

https://gcc.gnu.org/onlinedocs/cpp/index.html#SEC_Contents

High-level preprocessor steps

- Initial processing
- Tokenization
- Header files
- Macros
- Conditionals

Initial processing

- The input file is read into memory and broken into lines.
- Trigraphs (don’t use them)
- Continued lines are merged into one long line
- All comments are replaced with single spaces

Tokenization

Header files

- `#include <file>` or `#include "file"`
- Scan the specified file as input before continuing with the rest of the current file

Macros

- Object-like

```
#define BUFFER_SIZE 1024
```

- Function-like

```
#define PLUS_ONE(x) ((x) + 1)
```

- Variadic

```
#define eprintf(...) fprintf (stderr, __VA_ARGS__)
```

Conditionals

```
#ifdef MACRO

controlled text

#endif
```

```
//src.cpp
#include "a.h"

#ifdef NULL
#define X 1
#else
#define X -1
#endif

#define SQUARE(x) ((x) * (x))

int main() {
   int x = X;
   int num = SQUARE(x++);
   return 0;
}

//a.h
void fn_a(){}
```

```
g++ -E -o src.E src.cpp
```

```
//src.cpp
#include "a.h"

#ifdef NULL
#define X 1
#else
#define X -1
#endif

#define SQUARE(x) ((x) * (x))

int main() {
   int x = X;
   int num = SQUARE(x++);
   return 0;
}

//a.h
void fn_a(){}
```

```
//src.E
# 0 "src.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "src.cpp"

# 1 "a.h" 1
void fn_a(){}
# 3 "src.cpp" 2
# 12 "src.cpp"
int main() {
   int x = -1;
   int num = ((x++) * (x++));
   return 0;
}
```

linenum filename flags

Flags

‘1’ This indicates the start of a new file.

‘2’ This indicates returning to a file (after having included another file).

‘3’ This indicates that the following text comes from a system header file, so certain warnings should be suppressed.

‘4’ This indicates that the following text should be treated as being wrapped in an implicit extern "C" block.

```
//src.E
# 0 "src.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "src.cpp"

# 1 "a.h" 1
void fn_a(){}
# 3 "src.cpp" 2
# 12 "src.cpp"
int main() {
    int x = -1;
    int num = ((x++) * (x++));
    return 0;
}
```

Czym jest kowariancja i kontrawariancja? Podaj przykłady kiedy C++ dopuszcza użycie, a kiedy nie - w szczególności dla std::function

Subtyping

If

S is a subtype of ***T*** (written as: ***S*** < ***T***)

then

any term of type ***S*** can safely be used in any context
where a term of type ***T*** is expected

or less formally

S is at least as useful as ***T***

Type constructor

Builds new types from old ones.

Written as ***I<T>***

Eg.:

- `unique_pointer<bool>`
- `vector<int>`
- `std::function<const int&()>`
- `*T and &T`
- `const T`

Variance

A ***type constructor*** can be:

- covariant
- contravariant
- variant if covariant or contravariant
- invariant if not

Variance

Assume ***S*** is a subtype of ***T***, and
a type constructor ***I<U>***

● covariant	***I<S>*** is a subtype of ***I<T>***
● contravariant	***I<T>*** is a subtype of ***I<S>***
● variant if covariant or contravariant	***I<S>*** and ***I<T>*** are comparable
● invariant if not	***I<S>*** and ***I<T>*** are not comparable Meaning neither ***I<T>*** < ***I<S>*** nor ***I<T>*** > ***I<S>***

```
struct Tool{
   virtual void noise() const = 0;
   virtual ~Tool() = default;
};
struct Screwdriver : public Tool{
   void noise() const override {
       std::cout<<"cyk\n";
   }
};
```

Covariant

```
*T

&T
```

Covariant

```
std::unique_ptr<Tool> u = std::make_unique<Screwdriver>();
std::pair<const int, Tool*> p = std::pair(int{0}, new Screwdriver{});
std::function<const Tool*()> f([] -> Screwdriver* {return new Screwdriver{};});
```

In c++ variance is often one level deep

```
std::shared_ptr<Screwdriver> sd = std::make_shared<Screwdriver>();
std::shared_ptr<Tool>* tool = &sd;
```

Contravariant

```
std::function<void(Screwdriver*)> f([](Tool* t) {});
std::function<void(Screwdriver*)> f2([](Tool const* t) {});
```

```
std::function<const Tool*(Screwdriver&)> f3{
   [](Tool const& t) -> Screwdriver*
   {return new Screwdriver{};}
};
```

Invariant

```
std::vector<T>
std::future<T>
```

```
std::function<Tool()> fp{[] -> Screwdriver {return Screwdriver{};}};
```

Sprawdź, jak Twój kompilator obsługuje tzw. dangling references.

W tym celu napisz klasę struct A, a następnie w funkcji zwróć referencję do lokalnej instancji klasy A. Związ wspomnianą referencję przy wywołaniu funkcji oraz sprawdź, czy bezpiecznym jest operowanie na niej (np. dostęp do pól czy metod składowych).

Kiedy wywoływany jest destruktor obiektu tak związanego? Sprawdź zachowanie w różnych kompilatorach.

```cpp
#include <iostream>
using std::cout, std::endl;
struct A {
   ~A() {cout << "A is dead" << endl;}
   void hello() {cout << "Hello" << endl;}
};

A& f() { A local{}; return local; }

int main() {
   A& ref = f();
   ref.hello();
   return 0;
}
```

Expected output

```cpp
#include <iostream>
using std::cout, std::endl;
struct A {
   ~A() {cout << "A is dead" << endl;}
   void hello() {cout << "Hello" << endl;}
};

A& f() { A local{}; return local; }

int main() {
   A& ref = f();
   ref.hello();
   return 0;
}
```

```
Hello
A is dead
```

Actual output

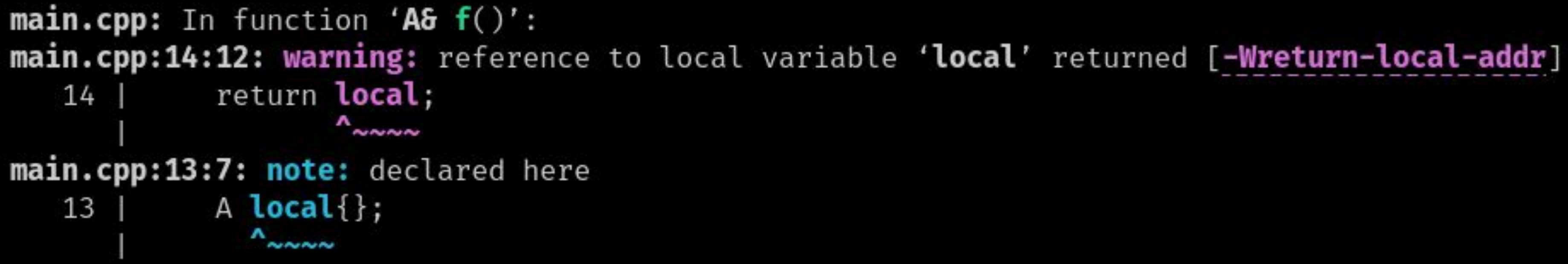
```cpp
#include <iostream>
using std::cout, std::endl;
struct A {
   ~A() {cout << "A is dead" << endl;}
   void hello() {cout << "Hello" << endl;}
};

A& f() { A local{}; return local; }

int main() {
   A& ref = f();
   ref.hello();
   return 0;
}
```

```
A is dead
Hello
```

g++



```
main.cpp: In function ‘A& f()’:
main.cpp:14:12: warning: reference to local variable ‘local’ returned [-Wreturn-local-addr]
   14 |     return local;
      |            ^~~~~
main.cpp:13:7: note: declared here
   13 |     A local{};
      |       ^~~~~
```

g++

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        call    f()
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        mov     rdi, rax
        call    A::hello()
        mov     eax, 0
        leave
        ret
```

clang

```
f():                                    # @f()
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        lea     rdi, [rbp - 1]
        call    A::~A() [base object destructor]
        lea     rax, [rbp - 1]
        add     rsp, 16
        pop     rbp
        ret
```

g++

```
f():
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 24
        mov     ebx, 0
        lea     rax, [rbp-17]
        mov     rdi, rax
        call    A::~A() [complete object destructor]
        mov     rax, rbx
        mov     rbx, QWORD PTR [rbp-8]
        leave
        ret
```

msvc

```
$T1 = -8                                                ; size = 4
_local$ = -1                                            ; size = 1
A & f(void) PROC                           ; f
       push     ebp
       mov      ebp, esp
       sub      esp, 8
       xor      eax, eax
       mov      BYTE PTR _local$[ebp], al
       lea      ecx, DWORD PTR _local$[ebp]
       mov      DWORD PTR $T1[ebp], ecx
       lea      ecx, DWORD PTR _local$[ebp]
       call     A::~A(void)                           ; A::~A
       mov      eax, DWORD PTR $T1[ebp]
       mov      esp, ebp
       pop      ebp
       ret      0
A & f(void) ENDP                           ; f
```

```cpp
#include <iostream>
using std::cout, std::endl;
struct A {
   int* number;
   A(): number{new int{10}} {}
   ~A() {delete number;}
   void hello() {cout << *(this->number) << endl;}
};

A& f() { A local{}; return local; }

int main() {
   A& ref = f();
   ref.hello();
   return 0;
}
```

Actual output

```
Segmentation fault (core dumped)
```

5.2

```cpp
#include <iostream>

template <std::size_t... Indices>
struct sequence {};

template <std::size_t First>
void print(sequence<First>) {
   std::cout << First << " ";
}

template <std::size_t First, std::size_t... Rest>
void print(sequence<First, Rest...>) {
   std::cout << First << " ";
   print(sequence<Rest...>());
}
```

```cpp
template <std::size_t N, std::size_t... Is>
struct make_sequence_impl {
   using type = typename make_sequence_impl<N-1, N-1, Is...>::type;
};

template <std::size_t... Is>
struct make_sequence_impl<0, Is...> {
   using type = sequence<Is...>;
};

template <std::size_t N>
using make_sequence = typename make_sequence_impl<N>::type;
```

```
int main() {
   print(make_sequence<4>());
   return 0;
}
```

```
int main() {
    print(make_sequence<4>());
    return 0;
}
```

```
0 1 2 3
```

Specialization templates

Primary template:

```
template <std::size_t N, std::size_t... Is>
struct make_sequence_impl {...};
```

Specialization template:

```
template <std::size_t... Is>
struct make_sequence_impl<0, Is...> {...};
```

1. Try to match the provided template arguments to the primary template
2. If there are any specializations available, check if any match
3. Choose the most *“specific”* match