GARMIN

# The Preprocessor

Everything you need to know, and more!

Brian Ruth(he/him) brian.ruth@garmin.com

# What happens when you write this?

```
#include “my_header.h”
```

GARMIN®

# Including stuff

my_header.h

```
typedef struct {
 int x;
 int y;
} point;

typedef enum {
 OR_LEFT,
 OR_RIGHT,
 OR_CENTER
} orientation;

point get_layout_left(orientation);
```

main.c

```
#include “my_header.h”

int main()
{
  const point p = get_layout_left(OR_LEFT);
  return p.x;
}
```

GARMIN.

# Including stuff

my_header.h

```
typedef struct {
 int x;
 int y;
} point;

typedef enum {
 OR_LEFT,
 OR_RIGHT,
 OR_CENTER
} orientation;

point get_layout_left(orientation);
```

main.c

```
#include “my_header.h”

int main()
{
  const point p = get_layout_left(OR_LEFT);
  return p.x;
}
```

main.c

```
typedef struct {
 int x;
 int y;
} point;

typedef enum {
 OR_LEFT,
 OR_RIGHT,
 OR_CENTER
} orientation;

point get_layout_left(orientation);

int main()
{
  const point p = get_layout_left(OR_LEFT);
  return p.x;
}
```

GARMIN.

# Including stuff

# Translation Unit

main.c

```
typedef struct {
 int x;
 int y;
} point;

typedef enum {
 OR_LEFT,
 OR_RIGHT,
 OR_CENTER
} orientation;

point get_layout_left(orientation);

int main()
{
  const point p = get_layout_left(OR_LEFT);
  return p.x;
}
```

GARMIN

# Translation Phases

- Phase 1 Source Character Translation
- Phase 2 Line Continuation Processing
- Phase 3 Comments/Whitespace/Pre-Preprocessor
- Phase 4 Preprocessor
- Phase 5 Execution Character Translation
- Phase 6 String Literal Concatenation
- Phase 7 Compilation
- Phase 8 Linking
- Phase 9 Profit!

GARMIN.

# Translation Phases

- Phase 1 Source Character Translation
- Phase 2 Line Continuation Processing
- Phase 3 Comments/Whitespace/Pre-Preprocessor
- **Phase 4 Preprocessor**
- Phase 5 Execution Character Translation
- Phase 6 String Literal Concatenation
- Phase 7 Compilation
- Phase 8 Linking
- Phase 9 Profit!

GARMIN®

# About this talk

GARMIN®

# The Preprocessor

Replaces the source code **text** before compilation by expanding directives. This can be manipulated to generate code in some very interesting and complicated ways.

Sometimes this text informs the compiler and/or linker to process the code differently than they normally would.

GARMIN®

# C and C++ Preprocessing

Though C++ inherited the pre-processor in C, they have since diverged, so not all C pre-processor code will be correctly interpreted by C++ and vice-versa.

# The Preprocessor is still here

It is still the most portable way to generate code in C and C++ and is used (and abused)

extensively to this day and shows no signs of going away anytime soon.

GARMIN.

Remember, everything in the preprocessor happens
BEFORE the compiler gets involved

GARMIN®

# A Preprocessor Directive

#<directive> <stuff>

- #
  - introduces a <directive>
  - # can be followed by 0 or more spaces
  - # followed by a newline is a no-op
  - # must be the first non-whitespace character
- <directive>
  - we'll see these later
- <stuff>
  - depends on the <directive>, more on this later
  - cannot introduce another directive either via # or via macro expansion

GARMIN.

# What’s a Directive?

A directive can be one of the following:

- Include
  - #include
- Conditional
  - #if, #ifdef, #ifndef, #else, #elif, #endif, #elifdef, #elifndef
- Replace
  - #define, #undef, #, ##
- Error
  - #error
- Compiler Extension
  - #pragma, _Pragma
- File and Line Information
  - #line

GARMIN®

# What’s a Directive?

A directive can be one of the following:

- Include
  - #include
- Conditional
  - #if, #ifdef, #ifndef, #else, #elif, #endif, #elifdef, #elifndef
- Replace
  - #define, #undef, #, ##
- Error
  - #error
- Compiler Extension
  - #pragma, _Pragma
- File and Line Information
  - #line
- Modules
  - import, export, module*

* module, import and export are special pre-processing directives under certain conditions, but that's another talk

GARMIN.

# including another file

#include file_to_include

- file_to_include
  - **evaluated by the preprocessor macro expansion**
  - **post-expansion must be in one of the following forms**
    - <path/to/file_name>
      - **path/to/file_name is searched for relative to system header directories (implementation defined)**
      - **if not found in system header directories, an error *may* be generated, or it will fall back to the quoted search**
    - "path/to/file_name/file_name"
      - **path/to/file_name is searched for first relative to the same directory as the current file being translated, then relative to non-system header directories, and finally relative to system header directories (implementation defined)**
  - **if the specified file is not found, an error will be generated**

GARMIN.

# #include

| When to use | C Alternative | C++ Alternative |
| --- | --- | --- |
| When you don't have module support | None | C++20 modules |

GARMIN.

# View preprocessor output

You can view the output after the preprocessor runs

| Compiler | Complier Flag |
| --- | --- |
| GCC/Clang | -E -o output.i |
| MSVC | /P /Fioutput.i |

GARMIN®

# Object like macros

```
#define PI 3.14159
#define RADIUS 2
#define CIRCUMFERENCE (PI*RADIUS)

int main()
{
  return (int)CIRCUMFERENCE;
}
```

GARMIN®

# Object like macros

#define <identifier> <replacement>

#undef <identifier>

- **define**
  - Introduces an identifier to the preprocessor
- **<identifier>**
  - must be a valid name according to variable naming rules
  - must be unique in this translation unit
- **<replacement>**
  - cannot introduce a new preprocessor directive
  - pretty much anything else is allowed, but no guarantee it will compile
- **undef**
  - Removes an identifier from the preprocessor

# Object like macros

```
#define PI 3.14159
#define RADIUS 2
#define CIRCUMFERENCE (PI*RADIUS)

int main()
{
  return (int)CIRCUMFERENCE;
}
```

```
#define PI 3.14159
#define RADIUS 2



int main()
{
  return (int)(PI * RADIUS);
}
```

GARMIN

# Object like macros

```
#define PI 3.14159
#define RADIUS 2



int main()
{
  return (int)(PI * RADIUS);
}
```

```
int main()
{
  return (int)(3.14159 * 2);
}
```

GARMIN®

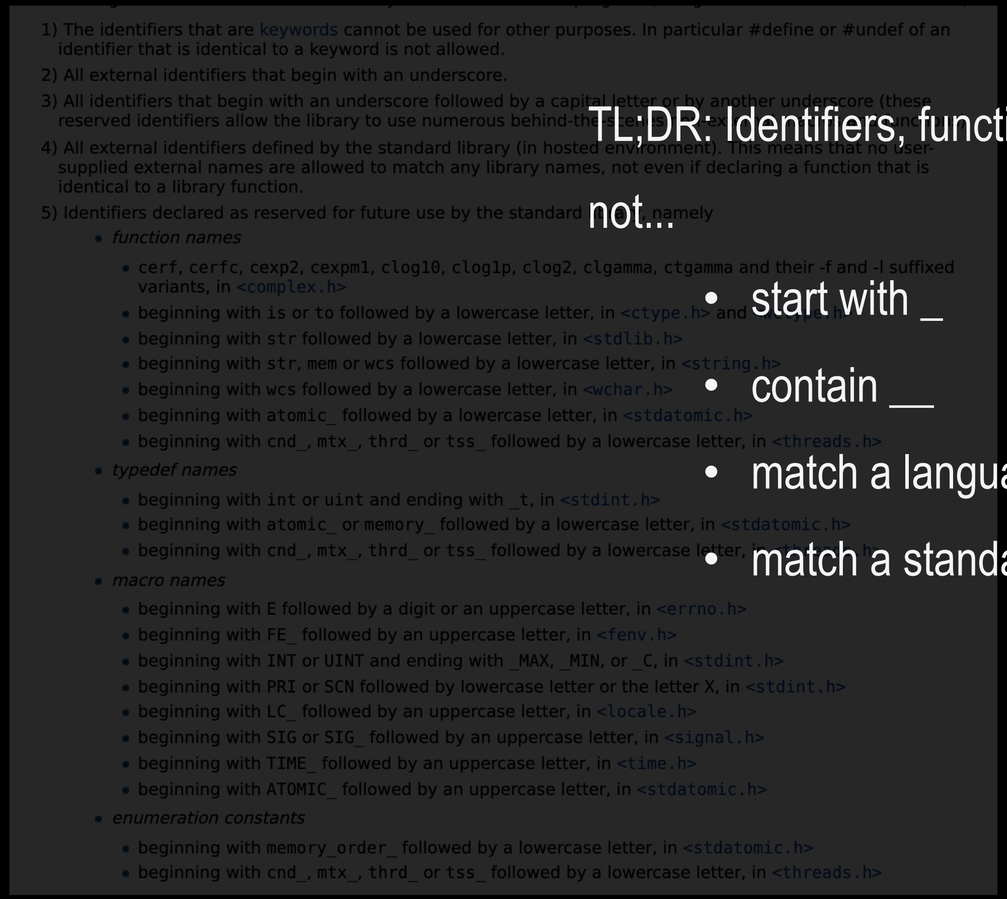
# Reserved Preprocessor Identifiers

1) The identifiers that are keywords cannot be used for other purposes. In particular #define or #undef of an identifier that is identical to a keyword is not allowed.
2) All external identifiers that begin with an underscore.
3) All identifiers that begin with an underscore followed by a capital letter or by another underscore (these reserved identifiers allow the library to use numerous behind-the-scenes non-external macros and functions)
4) All external identifiers defined by the standard library (in hosted environment). This means that no user-supplied external names are allowed to match any library names, not even if declaring a function that is identical to a library function.
5) Identifiers declared as reserved for future use by the standard library, namely
   - *function names*
     - `cerf`, `cerfc`, `cexp2`, `cexpm1`, `clog10`, `clog1p`, `clog2`, `clgamma`, `ctgamma` and their -f and -l suffixed variants, in `<complex.h>`
     - beginning with `is` or `to` followed by a lowercase letter, in `<ctype.h>` and `<wctype.h>`
     - beginning with `str` followed by a lowercase letter, in `<stdlib.h>`
     - beginning with `str`, `mem` or `wcs` followed by a lowercase letter, in `<string.h>`
     - beginning with `wcs` followed by a lowercase letter, in `<wchar.h>`
     - beginning with `atomic_` followed by a lowercase letter, in `<stdatomic.h>`
     - beginning with `cnd_`, `mtx_`, `thrd_` or `tss_` followed by a lowercase letter, in `<threads.h>`
   - *typedef names*
     - beginning with `int` or `uint` and ending with `_t`, in `<stdint.h>`
     - beginning with `atomic_` or `memory_` followed by a lowercase letter, in `<stdatomic.h>`
     - beginning with `cnd_`, `mtx_`, `thrd_` or `tss_` followed by a lowercase letter, in `<threads.h>`
   - *macro names*
     - beginning with E followed by a digit or an uppercase letter, in `<errno.h>`
     - beginning with `FE_` followed by an uppercase letter, in `<fenv.h>`
     - beginning with `INT` or `UINT` and ending with `_MAX`, `_MIN`, or `_C`, in `<stdint.h>`
     - beginning with `PRI` or `SCN` followed by lowercase letter or the letter X, in `<stdint.h>`
     - beginning with `LC_` followed by an uppercase letter, in `<locale.h>`
     - beginning with `SIG` or `SIG_` followed by an uppercase letter, in `<signal.h>`
     - beginning with `TIME_` followed by an uppercase letter, in `<time.h>`
     - beginning with `ATOMIC_` followed by an uppercase letter, in `<stdatomic.h>`
   - *enumeration constants*
     - beginning with `memory_order_` followed by a lowercase letter, in `<stdatomic.h>`
     - beginning with `cnd_`, `mtx_`, `thrd_` or `tss_` followed by a lowercase letter, in `<threads.h>`

https://en.cppreference.com/w/cpp/language/identifiers

GARMIN

# Reserved Preprocessor Identifiers



1) The identifiers that are keywords cannot be used for other purposes. In particular #define or #undef of an identifier that is identical to a keyword is not allowed.
2) All external identifiers that begin with an underscore.
3) All identifiers that begin with an underscore followed by a capital letter or by another underscore (these reserved identifiers allow the library to use numerous behind-the-[illegible]ex[illegible]n).
4) All external identifiers defined by the standard library (in hosted environment). This means that no user-supplied external names are allowed to match any library names, not even if declaring a function that is identical to a library function.
5) Identifiers declared as reserved for future use by the standard, namely
   - *function names*
     - `cerf`, `cerfc`, `cexp2`, `cexpm1`, `clog10`, `clog1p`, `clog2`, `clgamma`, `ctgamma` and their -f and -l suffixed variants, in `<complex.h>`
     - beginning with `is` or `to` followed by a lowercase letter, in `<ctype.h>` and [illegible]
     - beginning with `str` followed by a lowercase letter, in `<stdlib.h>`
     - beginning with `str`, `mem` or `wcs` followed by a lowercase letter, in `<string.h>`
     - beginning with `wcs` followed by a lowercase letter, in `<wchar.h>`
     - beginning with `atomic_` followed by a lowercase letter, in `<stdatomic.h>`
     - beginning with `cnd_`, `mtx_`, `thrd_` or `tss_` followed by a lowercase letter, in `<threads.h>`
   - *typedef names*
     - beginning with `int` or `uint` and ending with `_t`, in `<stdint.h>`
     - beginning with `atomic_` or `memory_` followed by a lowercase letter, in `<stdatomic.h>`
     - beginning with `cnd_`, `mtx_`, `thrd_` or `tss_` followed by a lowercase letter, in [illegible]
   - *macro names*
     - beginning with `E` followed by a digit or an uppercase letter, in `<errno.h>`
     - beginning with `FE_` followed by an uppercase letter, in `<fenv.h>`
     - beginning with `INT` or `UINT` and ending with `_MAX`, `_MIN`, or `_C`, in `<stdint.h>`
     - beginning with `PRI` or `SCN` followed by lowercase letter or the letter `X`, in `<stdint.h>`
     - beginning with `LC_` followed by an uppercase letter, in `<locale.h>`
     - beginning with `SIG` or `SIG_` followed by an uppercase letter, in `<signal.h>`
     - beginning with `TIME_` followed by an uppercase letter, in `<time.h>`
     - beginning with `ATOMIC_` followed by an uppercase letter, in `<stdatomic.h>`
   - *enumeration constants*
     - beginning with `memory_order_` followed by a lowercase letter, in `<stdatomic.h>`
     - beginning with `cnd_`, `mtx_`, `thrd_` or `tss_` followed by a lowercase letter, in `<threads.h>`

TL;DR: Identifiers, function or variable names you create should not...

- start with _
- contain __
- match a language keyword (for, if, public, int)
- match a standardized name (__FILE__, memcpy, vector)

h[illegible]ps://en.cppreference.com/w/cpp/language/identifiers

GARMIN.

# Object like macros

| When to use | C Alternative | C++ Alternative |
| --- | --- | --- |
| • When you need a value for use during the pre-processing phase<br>• When you need to support multiple compiler capabilities<br>• Command line flags | For constant expressions use enums<br>`enum {`<br>`  ARRAY_SIZE = 44`<br>`}`<br>`char arr[ARRAY_SIZE];` | use constexpr or static const values |

GARMIN.

# Allow command line flags

```
g++ test.cpp -DDEBUG_BUILD=true
```

```
void debug_print([[maybe_unused]] char* str){
  if constexpr (DEBUG_BUILD) {
   std::cout << str;
  }
}
```

- in addition to being used in code, these may be needed for some build systems in order to determine which files to include in the build

Directives do not have scope, the file is processed top down. So, moving directives can change generated code.

GARMIN.

# Macro scoping issues

```
#include <stdio.h>

void blorp()
{
    #define MY_VAL 7

    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
}

void other_fun(void);

int main()
{
    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
    blorp();
    other_fun();
    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
    return 0;
}

void other_fun(void)
{
    #define MY_VAL 9 // Generates a warning, maybe
    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
}
```

Output:

main: 7
blorp: 7
other_fun: 9
main: 7

GARMIN.

# Macro scoping issues

```
#include <stdio.h>

void blorp()
{
    #define MY_VAL 7

    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
}

void other_fun(void)
{
    #define MY_VAL 9 // Generates a warning, maybe
    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
}

int main()
{
    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
    blorp();
    other_fun();
    printf(“%s: %d\r\n", __FUNCTION__, MY_VAL);
    return 0;
}
```

Output:

**main: 9**
blorp: 7
other_fun: 9
**main: 9**

GARMIN.

# Conditionals

**main.cpp**

```
#include "pub_config.h"
#if defined(PUB_SUPPORTED) && PUB_SUPPORTED
#include "pub.h"
#endif

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

**pub_config.h**

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...
#endif
```

**pub.h**

```
#ifndef PUB_H
#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT
...
int asset_get_count();
...
#endif
```

GARMIN®

# Conditionals

```
#if <expr>
//code
#elif <expr>
//code
#else
//code
#endif
```

- **<expr>**
  - <expr> can optionally be surrounded by parenthesis
  - defined <identifier>, defined( <identifier> )
  - __has_include( file_name) (C++17)
  - __has_cpp_attribute( attribute_token) (C++20)
  - numeric constant expression with short-circuiting
    - 0 -> false and  != 0 -> true
    - If an identifier in the constant expression is not defined, it is treated as 0
- **#elif**
  - optional, can be one or more, must be after #if and before #else (if there are any) or #endif
- **#else**
  - optional, one per #if, must be after all #elif and before #endif
- **#ifdef <value>, #elifdef <value> (C++23)**
  - same as #if defined <value> or #if defined(<value>) / #elif defined <value> or #elif defined(<value>
  - cannot use parenthesis around <value>
- **#ifndef <value>, #elifndef <value> (C++23)**
  - same as #if !defined(<value>) or #if defined(<value>) == 0
  - cannot use parenthesis around <value>

GARMIN.

# Conditionals

**main.cpp**

```
#include "pub_config.h"
#if defined(PUB_SUPPORTED) && PUB_SUPPORTED
#include "pub.h"
#endif

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

**pub_config.h**

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...
#endif
```

**pub.h**

```
#ifndef PUB_H
#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT
...
int asset_get_count();
...
#endif
```

GARMIN®

# Conditionals

main.cpp

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...
#endif

#if defined(PUB_SUPPORTED) && PUB_SUPPORTED
#include "pub.h"
#endif

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

pub_config.h

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...
#endif
```

pub.h

```
#ifndef PUB_H
#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT
...
int asset_get_count();
...
#endif
```

GARMIN®

# Conditionals

main.cpp

```
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...

#if 1 && 1
#include "pub.h"
#endif

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

pub_config.h

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...
#endif
```

pub.h

```
#ifndef PUB_H
#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT
...
int asset_get_count();
...
#endif
```

GARMIN.

# Conditionals

**main.cpp**

```
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...

#include "pub.h"

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

**pub_config.h**

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...
#endif
```

**pub.h**

```
#ifndef PUB_H
#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT
...
int asset_get_count();
...
#endif
```

GARMIN®

# Conditionals

main.cpp

```
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...

#ifndef PUB_H
#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT
...
int asset_get_count();
...
#endif

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

pub_config.h

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED TRUE
...
#endif
```

GARMIN.

# Conditionals

**main.cpp**

```
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...

#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT
...
int asset_get_count();
...

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

**pub_config.h**

```
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED TRUE
...
#endif
```

GARMIN®

# Conditionals

main.cpp

```
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...

#define PUB_H
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED TRUE
...
#endif
#define ASSET_SUPPORT
...
int asset_get_count();
...

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

GARMIN®

# Conditionals

main.cpp

```
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED 1
...

#define PUB_H
#define ASSET_SUPPORT
...
int asset_get_count();
...

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

GARMIN.

# Conditionals

main.cpp

```
#define ASSET_SUPPORT
...
int asset_get_count();
...

int main()
{
 #ifdef ASSET_SUPPORT
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

GARMIN.

# Conditionals

main.cpp

```
...
int asset_get_count();
...

int main()
{
 #if 1
 return asset_get_count();
 #else
 return 0;
 #endif
}
```

GARMIN

# Conditionals

main.cpp

```
...
int asset_get_count();
...

int main()
{
 return asset_get_count();
}
```

GARMIN®

# Conditionals

| When to use | C Alternative | C++ Alternative |
| --- | --- | --- |
| • Include guards<br>• C linkage specification<br>`#ifdef __cplusplus`<br>`extern "C" {`<br>`#endif`<br>• Very sparingly, to exclude code<br>• Check for header or feature availability flags | Attempt to restructure code to limit inline code exclusion | • use constexpr if when all code is available<br>• module imports |

GARMIN.

# What about `if constexpr`?

# Code exclusion when not all paths compile

```
static constexpr bool BARK_USED = true;

#if BARK_USED
void proc_bark(void* buf){ ... }
#else
void proc_activity(void* buf) {...}
#endif
void handle_buffer(void* buf){
  if constexpr (BARK_USED) {
   proc_bark(buf);
  } else {
   proc_activity(buf);
  }
}
```

GARMIN.

# Code exclusion when not all paths compile

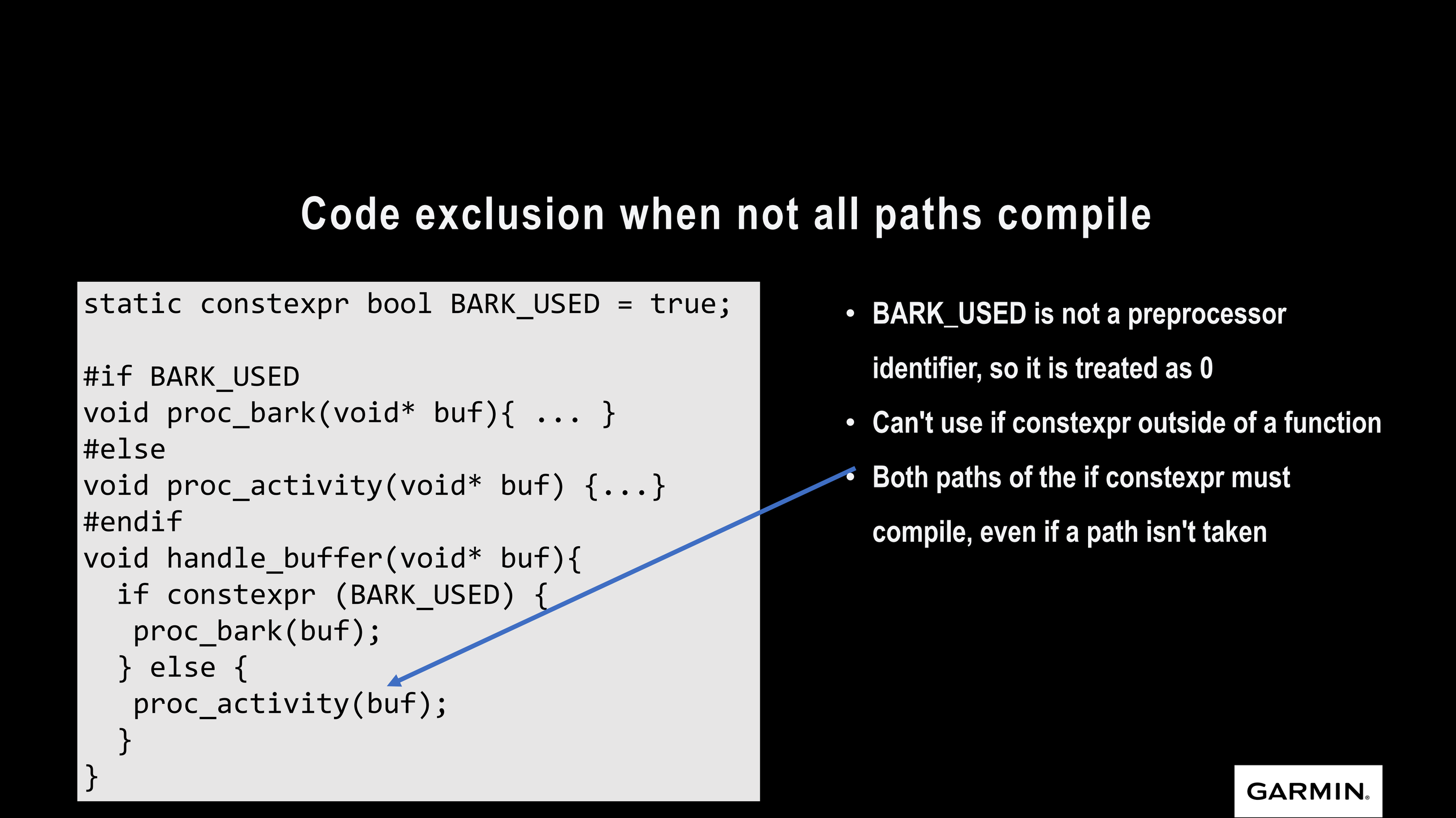
```
static constexpr bool BARK_USED = true;

#if BARK_USED
void proc_bark(void* buf){ ... }
#else
void proc_activity(void* buf) {...}
#endif
void handle_buffer(void* buf){
  if constexpr (BARK_USED) {
   proc_bark(buf);
  } else {
   proc_activity(buf);
  }
}
```

- BARK_USED is not a preprocessor identifier, so it is treated as 0
- Can't use if constexpr outside of a function

GARMIN.

# Code exclusion when not all paths compile

```
static constexpr bool BARK_USED = true;

#if BARK_USED
void proc_bark(void* buf){ ... }
#else
void proc_activity(void* buf) {...}
#endif
void handle_buffer(void* buf){
  if constexpr (BARK_USED) {
   proc_bark(buf);
  } else {
   proc_activity(buf);
  }
}
```



- BARK_USED is not a preprocessor identifier, so it is treated as 0
- Can't use if constexpr outside of a function
- Both paths of the if constexpr must compile, even if a path isn't taken

GARMIN.

# Code exclusion when not all paths compile

```
#define BARK_USED 1

#if BARK_USED
void proc_bark(void* buf){ ... }
#else
void proc_activity(void* buf) {...}
#endif
void handle_buffer(void* buf){
  #if BARK_USED
   proc_bark(buf);
  #else
   proc_activity(buf);
  #endif
}
```

- BARK_USED is not a preprocessor identifier, so it is treated as 0
- Can't use if constexpr outside of a function
- Both paths of the if constexpr must compile, even if a path isn't taken
- if constexpr is not a substitute for #if

GARMIN.

# Check for files or attributes

```
#if __has_include("myHeader.hpp")
#include "myHeader.hpp"
#else
static void flarb(int){}
#endif
```

GARMIN®

# Check for files or attributes

```
#if __has_include("myHeader.hpp")
#include "myHeader.hpp"
#else
static void flarb(int){}
#endif

#if __has_attribute(__cpp_constexpr)
#define CONSTEXPR constexpr
#else
#define CONSTEXPR
#endif
```

GARMIN.

# Check for files or attributes

```
#if __has_include("myHeader.hpp")
#include "myHeader.hpp"
#else
static void flarb(int){}
#endif

#if __has_attribute(__cpp_constexpr)
#define CONSTEXPR constexpr
#else
#define CONSTEXPR
#endif

void someFunc() {
 static CONSTEXPR int max_items = 4;
 flarb(max_items);
}
```

GARMIN.

# Code exclusion *ANTIPATERNS*

**Excluding enum values**

```
enum struct PageMode {

   Visible,
   Enabled,
 #if (DARK_MODE_ENABLED)
   DarkMode,
 #endif
   Highlighted
};
```

GARMIN.

# Code exclusion *ANTIPATERNS*

**Excluding enum values**

```
enum struct PageMode {

   Visible,
   Enabled,
 #if (DARK_MODE_ENABLED)
   DarkMode,
 #endif
   Highlighted
};
```

**Changing flow control**

```
#if (CONTACT_SUPPORT)
  if( item->contact_count > 0)
{
   ...
  } else
#endif
  if(item->is_active) {
   ...
  }
```

GARMIN.

# Code exclusion *ANTIPATERNS*

**Excluding enum values**

```
enum struct PageMode {

   Visible,
   Enabled,
 #if (DARK_MODE_ENABLED)
   DarkMode,
 #endif
   Highlighted
};
```

**Changing flow control**

```
#if (CONTACT_SUPPORT)
  if( item->contact_count > 0)
{
   ...
  } else
#endif
  if(item->is_active) {
   ...
  }
```

**Excluding members**

```
struct Item {
 bool is_active = false;
 #if (CONTACT_SUPPORT)
 int contact_count = 0;
 #endif
 Color track_color =
Color::Red;
};
```

GARMIN.

# Function like macros

```
#define maxval(x, y) ((x) > (y) ? (x) : (y))
```

```
float f1 = 4.1f;
float f2 = 5.2f;
auto res = maxval(f1,--f2);
printf("%f\n", res);
```

GARMIN®

# Function like macros

```
#define <identifier>(<arg1>, <arg2>) \
<replacement_code >
```

- **<identifier>**
  - must be a valid name according to function naming rules
  - when used elsewhere, must include parenthesis, otherwise is a noop
  - there can be no space between <identifier> and (
- **<arg#>**
  - comma separated list of arguments, # of allowed args is implementation defined
  - whatever is passed in as <arg#> is pasted into the code
  - always surround with parenthesis ( <arg#> ) when referenced in replacement code
  - if referenced more than once, args with side effects will be evaluated multiple times
- ****
  - required if macro spans multiple lines
  - \ must be immediately followed by newline on every line that is part of this macro
- **<replacement_code>**
  - cannot introduce new preprocessor directive
  - pretty much anything else goes here, no guarantee it will compile though
  - last line generally does not end in ;

GARMIN.

# Function like macros

```
#define maxval(x, y) ((x) > (y) ? (x) : (y))
```

```
float f1 = 4.1f;
float f2 = 5.2f;
auto res = maxval(f1,--f2);
printf("%f\n", res);
```

```
float f1 = 4.1f;
float f2 = 5.2f;
auto res = ((f1) > (--f2) ? (f1) : (--f2));
printf("%f\n", res);
```

Output:

3.2

GARMIN.

# Function like macros - use ()

```
#define mult(x, y) x*y
```

```
int i1 = 2;
int i2 = 3;
auto res = mult(i1 + 1, i2); // 3, 3
printf("%d\n", res);
```

GARMIN.

# Function like macros - use ()

```
#define mult(x, y) x*y
```

```
int i1 = 2;
int i2 = 3;
auto res = mult(i1 + 1, i2);
printf("%d\n", res);
```

```
int i1 = 2;
int i2 = 3;
auto res = i1 + 1*i2; //2 + 1*3
printf("%d\n", res);
```

Output:

5

GARMIN.

# Function like macros - use ()

```
#define mult(x, y) (x)*(y)
```

```
int i1 = 2;
int i2 = 3;
auto res = mult(i1 + 1, i2);
printf("%d\n", res);
```

```
int i1 = 2;
int i2 = 3;
auto res = (i1 + 1)*(i2); // 3 * 3
printf("%d\n", res);
```

Output:

9

GARMIN.

# Function like macros

| When to use | C Alternative | C++ Alternative |
|---|---|---|
| • Perform more complicated pre-processor operations<br>• Generate blocks of code | Create multiple functions for each type with _Generic | • constexpr functions<br>• templates |

GARMIN.

# Code Generation

```
enum struct FontAttr { None, Bold = 0b001, Underline = 0b010, Italic = 0b100};

constexpr bool EnableBitMaskOperators(FontAttr) { return true; };
  using UTL::operator|;
  using UTL::operator&;
  using UTL::operator^;
  using UTL::operator~;
  using UTL::operator|=;
  using UTL::operator&=;
  using UTL::operator^=
```

GARMIN®

# Code Generation

```
enum struct FontAttr { None, Bold = 0b001, Underline = 0b010, Italic = 0b100};
MAKE_CPP_BITMASK_ENUM(FontAttr);

enum struct WidgetAttr { Default, Enabled = 0b001, Highlighted = 0b010 };
MAKE_CPP_BITMASK_ENUM(WidgetAttr);
```

```
#define MAKE_CPP_BITMASK_ENUM(Enum)                                    \
 constexpr bool EnableBitMaskOperators(Enum) { return true; };         \
  using UTL::operator|;                                                \
  using UTL::operator&;                                                \
  using UTL::operator^;                                                \
  using UTL::operator~;                                                \
  using UTL::operator|=;                                               \
  using UTL::operator&=;                                               \
  using UTL::operator^=

template <typename Enum> constexpr bool EnableBitMaskOperators(Enum)
 {  return false; };
```

GARMIN®

# Variadic function like macros

#define <identifier>(<arg1>, ...) \

<replacement_code >

- **<arg#>**
  - optional, same rules as previously, but must precede ... if used
- **...**
  - represents any number of unnamed arguments*
  - values accessed through __VA_ARGS__ preprocessor identifier which maps to a va_list containing the unnamed arguments
  - can only forward values to other macros or functions if no <arg#> is specified
  - if an <arg#> is specified, __VA_ARGS__ can be manipulated through functions in <stdarg.h>
    - va_arg, va_start, va_copy, va_end

GARMIN.

# What if I pass 0 variadic args?

- You can have a named item in a function like macro an an ellipsis, but the ellipsis must come at the end

```
#define DEBUG(fmt, ...) printf(fmt, __VA_ARGS__)
```

- However, if the variable arguments are empty

```
DEBUG(“just text, thank you\r\n);
```

- After the preprocessor this becomes…

```
printf(“just text, thank you\r\n”, );
```

Trailing comma, won’t compile

GARMIN.

# What if I pass 0 variadic args?

- GCC has a non-portable extension that will omit the trailing comma if the variable argument is empty if you prepend ## to __VA_ARGS__

```
#define DEBUG(fmt, ...) printf(fmt, ##__VA_ARGS__)
```

- And C++20 introduces __VA_OPT__ to optionally omit portions of a macro if there are no variadic arguments*

```
#define DEBUG(fmt, ...) printf(fmt __VA_OPT__(,) __VA_ARGS__)
```

- Either of these options make our previous example change to

```
printf(“just text, thank you\r\n”);
```

* As of now, this is not part of the C2X working draft, but seems to be supported as an extension

GARMIN.

# Variadic function like macros

| When to use | C Alternative | C++ Alternative |
|---|---|---|
| • Perform more complicated pre-processor operations<br>• Absorb all parameters and turn into noop | variadic functions with <stdargs.h> | variadic templates |

GARMIN.

# Some things you may see...

GARMIN®

# “Inheritance” in C

```
#define coord_common int x;\
  int y

typedef struct {
 coord_common;
} point;

typedef struct {
 coord_common;
 int z;
} three_d_point;

typedef struct {
 coord_common;
 color_ref color;
} color_point;

void draw_at_point(point* p);

int main() {
 three_d_point p3d;
 draw_at_point((point*)&p3d);
 color_point cp;
 draw_at_point((point*)&cp);
 return 0;
}
```

GARMIN®

# “Inheritance” in C

```
#define coord_common int x;\
  int y

typedef struct {
 coord_common;
} point;

typedef struct {
 coord_common;
 int z;
} three_d_point;

typedef struct {
 coord_common;
 color_ref color;
} color_point;

void draw_at_point(point* p);

int main() {
 three_d_point p3d;
 draw_at_point((point*)&p3d);
 color_point cp;
 draw_at_point((point*)&cp);
 return 0;
}
```

```
typedef struct {
 int x;
 int y;
} point;

typedef struct {
 int x;
 int y;
 int z;
} three_d_point;

typedef struct {
 int x;
 int y;
 color_ref color;
} color_point;

void draw_at_point(point* p);

int main() {
 three_d_point p3d;
 draw_at_point((point*)&p3d);
 color_point cp;
 draw_at_point((point*)&cp);
 return 0;
}
```

GARMIN®

# “Overloading” in C11*

```
#define abs(X) _Generic((X),       \
               long double: fabsl,  \
                    double: fabs,   \
                     float: fabsf,  \
                       int: abs,    \
                      long: labs,   \
                 long long: llabs,  \
                   default: fabs    \
                )(X)


int x = 7;
double y = -4.445;
double absval = abs(x * y);
```

* This will not compile in standard C++

GARMIN®

# “Overloading” in C11*

```
int x = 7;
double y = -4.445;
double absval = _Generic((x * y), long double: fabsl, double: fabs, float: fabsf, int: abs, long: labs, long long: llabs, default: fabs )(x * y);
```

* This will not compile in standard C++

GARMIN®

# “Overloading” in C11*

```
int x = 7;
double y = -4.445;
double absval = _Generic((x * y), long double: fabsl, double: fabs, float: fabsf, int: abs, long: labs, long long: llabs, default: fabs )(x * y);
```

* This will not compile in standard C++

GARMIN®

# “Overloading” in C11*

```
int x = 7;
double y = -4.445;
double absval = fabs(x * y);
```

* This will not compile in standard C++

GARMIN®

# What’s with the do...while loop?

```
#define debug_printf(...)                                  \
  do {                                                     \
    printf("[%lu](%s) ", get_timer(), __func__);           \
    printf(__VA_ARGS__);                                   \
    printf("\r\n");                                        \
  } while (0)
```

GARMIN®

# No loop...

```
#define debug_printf(...)                                \
    printf("[%lu](%s) ", get_timer(), __func__);    \
    printf(__VA_ARGS__);                                 \
    printf("\r\n")
```

```
if(result == FAILED)
 debug_printf(“FAILED”);
else
 set_state(STATE_IDLE);
```

GARMIN®

# No loop...oops

```
#define debug_printf(...)                                \
    printf("[%lu](%s) ", get_timer(), __func__);    \
    printf(__VA_ARGS__);                                 \
    printf("\r\n")
```

```
if(result == FAILED)
 debug_printf(“FAILED”);
else
 set_state(STATE_IDLE);
```

```
if(result == FAILED)
 printf("[%lu](%s) ", get_timer(), "proc_state");
 printf(“FAILED”);
 printf("\r\n");
else
 set_state(STATE_IDLE);
```

GARMIN®

# No loop...oops

```
#define debug_printf(...)                                   \
    printf("[%lu](%s) ", get_timer(), __func__);    \
    printf(__VA_ARGS__);                                    \
    printf("\r\n")
```

```
if(result == FAILED)
 debug_printf(“FAILED”);
else
 set_state(STATE_IDLE);
```

```
if(result == FAILED)
 printf("[%lu](%s) ", get_timer(), "proc_state");
 printf(“FAILED”);
 printf("\r\n");
else
 set_state(STATE_IDLE);
```

GARMIN®

# {} leaves dangling ;

```
#define debug_printf(...)                                  \
  {                                                        \
    printf("[%lu](%s) ", get_timer(), __func__);           \
    printf(__VA_ARGS__);                                   \
    printf("\r\n");                                        \
  }
```

```
if(result == FAILED)
 debug_printf(“FAILED”);
else
 set_state(STATE_IDLE);
```

```
if(result == FAILED)
 {
 printf("[%lu](%s) ", get_timer(), "proc_state");
 printf(“FAILED”);
 printf("\r\n");
 };
else
 set_state(STATE_IDLE);
```

GARMIN®

# {} leaves dangling ;

```
#define debug_printf(...)                                   \
  {                                                         \
    printf("[%lu](%s) ", get_timer(), __func__);            \
    printf(__VA_ARGS__);                                    \
    printf("\r\n")                                          \
  }
```

```
if(result == FAILED)
 debug_printf(“FAILED”);
else
 set_state(STATE_IDLE);
```

```
if(result == FAILED)
 {
 printf("[%lu](%s) ", get_timer(), "proc_state");
 printf(“FAILED”);
 printf("\r\n");
 };
else
 set_state(STATE_IDLE);
```

GARMIN.

# So, do...while(0)

```
#define debug_printf(...)                                   \
  do {                                                      \
    printf("[%lu](%s) ", get_timer(), __func__);            \
    printf(__VA_ARGS__);                                    \
    printf("\r\n");                                         \
  } while (0)
```

```
if(result == FAILED)
 debug_printf(“FAILED”);
else
 set_state(STATE_IDLE);
```

```
if(result == FAILED)
 do {
  printf("[%lu](%s) ", get_timer(), "proc_state");
  printf(“FAILED”);
  printf("\r\n");
 } while(0);
else
 set_state(STATE_IDLE);
```

GARMIN®

# Predefined Preprocessor Identifiers

C

### Predefined macros

The following macro names are predefined in any translation unit:

| | |
|---|---|
| `__STDC__` | expands to the integer constant `1`. This macro is intended to indicate a conforming implementation (macro constant) |
| `__STDC_VERSION__` (C95) | expands to an integer constant of type `long` whose value increases with each version of the C standard:<br>• `199409L` (C95)<br>• `199901L` (C99)<br>• `201112L` (C11)<br>• `201710L` (C17)<br>(macro constant) |
| `__STDC_HOSTED__` (C99) | expands to the integer constant `1` if the implementation is hosted (runs under an OS), `0` if freestanding (runs without an OS) (macro constant) |
| `__FILE__` | expands to the name of the current file, as a character string literal, can be changed by the `#line` directive (macro constant) |
| `__LINE__` | expands to the source file line number, an integer constant, can be changed by the `#line` directive (macro constant) |
| `__DATE__` | expands to the date of translation, a character string literal of the form "Mmm dd yyyy". The name of the month is as if generated by `asctime` and the first character of "dd" is a space if the day of the month is less than 10 (macro constant) |
| `__TIME__` | expands to the time of translation, a character string literal of the form "hh:mm:ss", as in the time generated by `asctime()` (macro constant) |

The following additional macro names may be predefined by an implementation:

| | |
|---|---|
| `__STDC_ISO_10646__` (C99) | expands to an integer constant of the form yyyymmL, if `wchar_t` uses Unicode; the date indicates the latest revision of Unicode supported (macro constant) |
| `__STDC_IEC_559__` (C99) | expands to `1` if IEC 60559 is supported (deprecated) (since C23) (macro constant) |
| `__STDC_IEC_559_COMPLEX__` (C99) | expands to `1` if IEC 60559 complex arithmetic is supported (deprecated) (since C23) (macro constant) |
| `__STDC_UTF_16__` (C11) | expands to `1` if `char16_t` use UTF-16 encoding (macro constant) |
| `__STDC_UTF_32__` (C11) | expands to `1` if `char32_t` use UTF-32 encoding (macro constant) |
| `__STDC_MB_MIGHT_NEQ_WC__` (C99) | expands to `1` if `'x' == L'x'` might be false for a member of the basic character set, such as on EBCDIC-based systems that use Unicode for `wchar_t` (macro constant) |
| `__STDC_ANALYZABLE__` (C11) | expands to `1` if analyzability is supported (macro constant) |
| `__STDC_LIB_EXT1__` (C11) | expands to an integer constant `201112L` if bounds-checking interfaces are supported (macro constant) |
| `__STDC_NO_ATOMICS__` (C11) | expands to `1` if atomic types and atomic operations library are not supported (macro constant) |
| `__STDC_NO_COMPLEX__` (C11) | expands to `1` if complex types and complex math library are not supported (macro constant) |
| `__STDC_NO_THREADS__` (C11) | expands to `1` if multithreading is not supported (macro constant) |
| `__STDC_NO_VLA__` (C11) | expands to `1` if variable-length arrays are not supported (macro constant) |
| `__STDC_IEC_60559_BFP__` (C23) | expands to `202ymmL` (date to be determined) if IEC 60559 binary floating-point arithmetic is supported (macro constant) |
| `__STDC_IEC_60559_DFP__` (C23) | expands to `202ymmL` (date to be determined) if IEC 60559 decimal floating-point arithmetic is supported (macro constant) |
| `__STDC_IEC_60559_COMPLEX__` (C23) | expands to `202ymmL` (date to be determined) if IEC 60559 complex arithmetic is supported (macro constant) |

The values of these macros (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit. Attempts to redefine or undefine these macros result in undefined behavior.

The predefined variable `__func__` (see function definition for details) is not a preprocessor macro, even though it is sometimes used together with `__FILE__` and `__LINE__`, e.g. by `assert`. (since C99)

https://en.cppreference.com/w/c/preprocessor/replace

C++

### Predefined macros

The following macro names are predefined in every translation unit.

| | |
|---|---|
| `__cplusplus` | denotes the version of C++ standard that is being used, expands to value 199711L(until C++11), 201103L(C++11), 201402L(C++14), 201703L(C++17), or 202002L(C++20) (macro constant) |
| `__STDC_HOSTED__` (C++11) | expands to the integer constant `1` if the implementation is hosted (runs under an OS), `0` if freestanding (runs without an OS) (macro constant) |
| `__FILE__` | expands to the name of the current file, as a character string literal, can be changed by the `#line` directive (macro constant) |
| `__LINE__` | expands to the source file line number, an integer constant, can be changed by the `#line` directive (macro constant) |
| `__DATE__` | expands to the date of translation, a character string literal of the form "Mmm dd yyyy". The first character of "dd" is a space if the day of the month is less than 10. The name of the month is as if generated by `std::asctime()` (macro constant) |
| `__TIME__` | expands to the time of translation, a character string literal of the form "hh:mm:ss" (macro constant) |
| `__STDCPP_DEFAULT_NEW_ALIGNMENT__` (C++17) | expands to an `std::size_t` literal whose value is the alignment guaranteed by a call to alignment-unaware operator new (larger alignments will be passed to alignment-aware overload, such as `operator new(std::size_t, std::align_val_t)` (macro constant) |

The following additional macro names may be predefined by the implementations.

| | |
|---|---|
| `__STDC__` | implementation-defined value, if present, typically used to indicate C conformance (macro constant) |
| `__STDC_VERSION__` (C++11) | implementation-defined value, if present (macro constant) |
| `__STDC_ISO_10646__` (C++11) | expands to an integer constant of the form yyyymmL, if `wchar_t` uses Unicode, the date indicates the latest revision of Unicode supported (macro constant) |
| `__STDC_MB_MIGHT_NEQ_WC__` (C++11) | expands to `1` if `'x' == L'x'` might be false for a member of the basic character set, such as on EBCDIC-based systems that use Unicode for wchar_t. (macro constant) |
| `__STDCPP_THREADS__` (C++11) | expands to `1` if the program can have more than one thread of execution (macro constant) |
| `__STDCPP_STRICT_POINTER_SAFETY__` (C++11)(removed in C++23) | expands to `1` if the implementation has strict `std::pointer_safety` (macro constant) |

The values of these macros (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit. Attempts to redefine or undefine these macros result in undefined behavior.

Note: in the scope of every function body, there is a special function-local predefined variable named `__func__`, defined as a static character array holding the name of the function in implementation-defined format. It is not a preprocessor macro, but it is used together with `__FILE__` and `__LINE__`, e.g. by `assert`. (since C++11)

#### Language feature-testing macros

The standard defines a set of preprocessor macros corresponding to C++ language features introduced in C++11 or later. They are intended as a simple and portable way to detect the presence of said features. (since C++20)

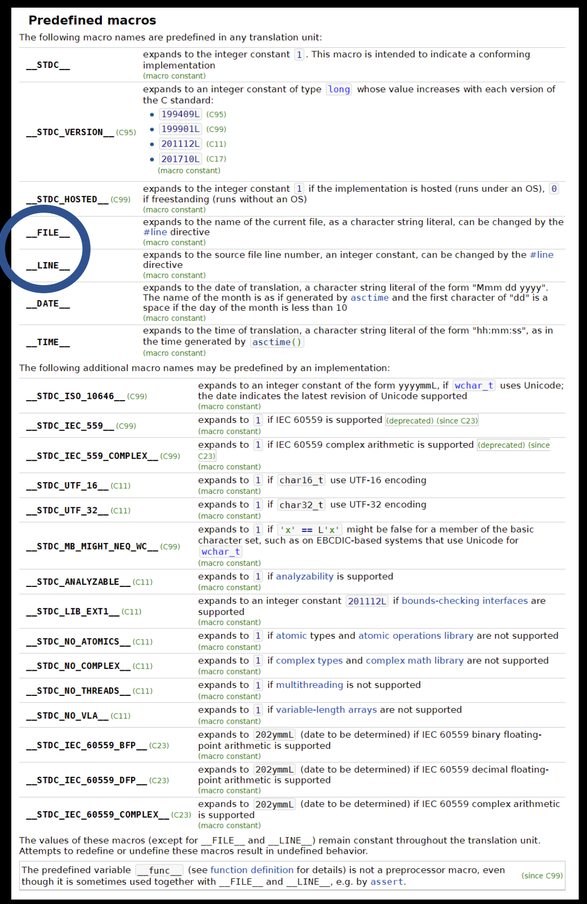
See Feature testing for details.

https://en.cppreference.com/w/cpp/preprocessor/replace

GARMIN

# Predefined Preprocessor Identifiers

C



### Predefined macros

The following macro names are predefined in any translation unit:

| | |
|---|---|
| `__STDC__` | expands to the integer constant `1`. This macro is intended to indicate a conforming implementation (macro constant) |
| `__STDC_VERSION__` (C95) | expands to an integer constant of type `long` whose value increases with each version of the C standard:<br>• `199409L` (C95)<br>• `199901L` (C99)<br>• `201112L` (C11)<br>• `201710L` (C17)<br>(macro constant) |
| `__STDC_HOSTED__` (C99) | expands to the integer constant `1` if the implementation is hosted (runs under an OS), `0` if freestanding (runs without an OS) (macro constant) |
| `__FILE__` | expands to the name of the current file, as a character string literal, can be changed by the #line directive (macro constant) |
| `__LINE__` | expands to the source file line number, an integer constant, can be changed by the #line directive (macro constant) |
| `__DATE__` | expands to the date of translation, a character string literal of the form "Mmm dd yyyy". The name of the month is as if generated by `asctime` and the first character of "dd" is a space if the day of the month is less than 10 (macro constant) |
| `__TIME__` | expands to the time of translation, a character string literal of the form "hh:mm:ss", as in the time generated by `asctime()` (macro constant) |

The following additional macro names may be predefined by an implementation:

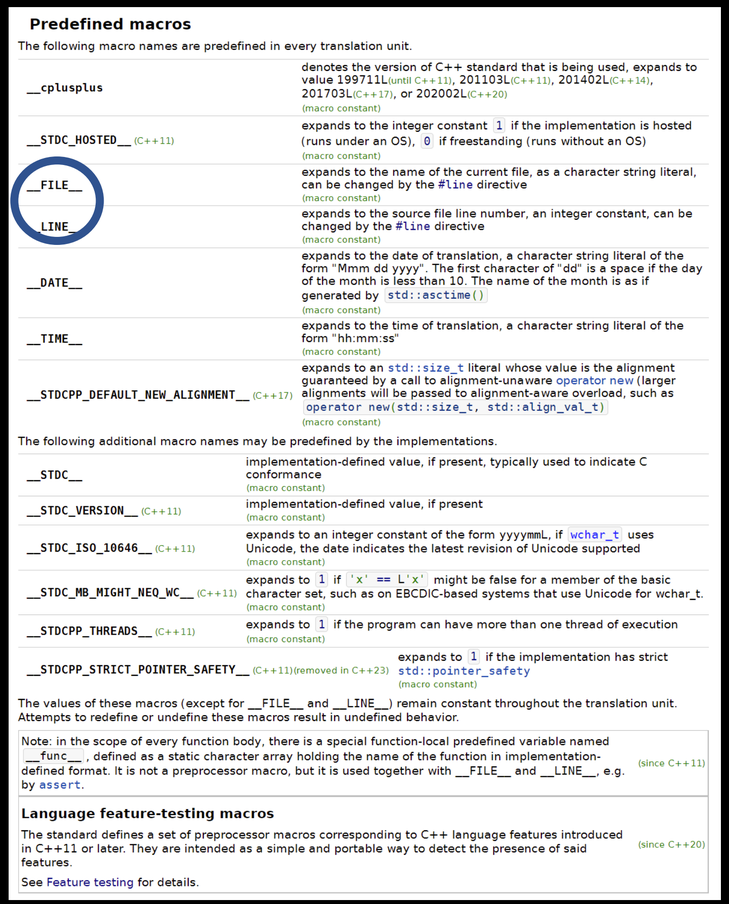
| | |
|---|---|
| `__STDC_ISO_10646__` (C99) | expands to an integer constant of the form yyyymmL, if `wchar_t` uses Unicode; the date indicates the latest revision of Unicode supported (macro constant) |
| `__STDC_IEC_559__` (C99) | expands to `1` if IEC 60559 is supported (deprecated) (since C23) (macro constant) |
| `__STDC_IEC_559_COMPLEX__` (C99) | expands to `1` if IEC 60559 complex arithmetic is supported (deprecated) (since C23) (macro constant) |
| `__STDC_UTF_16__` (C11) | expands to `1` if `char16_t` use UTF-16 encoding (macro constant) |
| `__STDC_UTF_32__` (C11) | expands to `1` if `char32_t` use UTF-32 encoding (macro constant) |
| `__STDC_MB_MIGHT_NEQ_WC__` (C99) | expands to `1` if `'x' == L'x'` might be false for a member of the basic character set, such as on EBCDIC-based systems that use Unicode for `wchar_t` (macro constant) |
| `__STDC_ANALYZABLE__` (C11) | expands to `1` if analyzability is supported (macro constant) |
| `__STDC_LIB_EXT1__` (C11) | expands to an integer constant `201112L` if bounds-checking interfaces are supported (macro constant) |
| `__STDC_NO_ATOMICS__` (C11) | expands to `1` if atomic types and atomic operations library are not supported (macro constant) |
| `__STDC_NO_COMPLEX__` (C11) | expands to `1` if complex types and complex math library are not supported (macro constant) |
| `__STDC_NO_THREADS__` (C11) | expands to `1` if multithreading is not supported (macro constant) |
| `__STDC_NO_VLA__` (C11) | expands to `1` if variable-length arrays are not supported (macro constant) |
| `__STDC_IEC_60559_BFP__` (C23) | expands to `202ymmL` (date to be determined) if IEC 60559 binary floating-point arithmetic is supported (macro constant) |
| `__STDC_IEC_60559_DFP__` (C23) | expands to `202ymmL` (date to be determined) if IEC 60559 decimal floating-point arithmetic is supported (macro constant) |
| `__STDC_IEC_60559_COMPLEX__` (C23) | expands to `202ymmL` (date to be determined) if IEC 60559 complex arithmetic is supported (macro constant) |

The values of these macros (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit. Attempts to redefine or undefine these macros result in undefined behavior.

The predefined variable `__func__` (see function definition for details) is not a preprocessor macro, even though it is sometimes used together with `__FILE__` and `__LINE__`, e.g. by `assert`. (since C99)

https://en.cppreference.com/w/c/preprocessor/replace

C++



### Predefined macros

The following macro names are predefined in every translation unit.

| | |
|---|---|
| `__cplusplus` | denotes the version of C++ standard that is being used, expands to value 199711L (until C++11), 201103L (C++11), 201402L (C++14), 201703L (C++17), or 202002L (C++20) (macro constant) |
| `__STDC_HOSTED__` (C++11) | expands to the integer constant `1` if the implementation is hosted (runs under an OS), `0` if freestanding (runs without an OS) (macro constant) |
| `__FILE__` | expands to the name of the current file, as a character string literal, can be changed by the `#line` directive (macro constant) |
| `__LINE__` | expands to the source file line number, an integer constant, can be changed by the `#line` directive (macro constant) |
| `__DATE__` | expands to the date of translation, a character string literal of the form "Mmm dd yyyy". The first character of "dd" is a space if the day of the month is less than 10. The name of the month is as if generated by `std::asctime()` (macro constant) |
| `__TIME__` | expands to the time of translation, a character string literal of the form "hh:mm:ss" (macro constant) |
| `__STDCPP_DEFAULT_NEW_ALIGNMENT__` (C++17) | expands to an `std::size_t` literal whose value is the alignment guaranteed by a call to alignment-unaware operator new (larger alignments will be passed to alignment-aware overload, such as `operator new(std::size_t, std::align_val_t)` (macro constant) |

The following additional macro names may be predefined by the implementations.

| | |
|---|---|
| `__STDC__` | implementation-defined value, if present, typically used to indicate C conformance (macro constant) |
| `__STDC_VERSION__` (C++11) | implementation-defined value, if present (macro constant) |
| `__STDC_ISO_10646__` (C++11) | expands to an integer constant of the form yyyymmL, if `wchar_t` uses Unicode, the date indicates the latest revision of Unicode supported (macro constant) |
| `__STDC_MB_MIGHT_NEQ_WC__` (C++11) | expands to `1` if `'x' == L'x'` might be false for a member of the basic character set, such as on EBCDIC-based systems that use Unicode for wchar_t. (macro constant) |
| `__STDCPP_THREADS__` (C++11) | expands to `1` if the program can have more than one thread of execution (macro constant) |
| `__STDCPP_STRICT_POINTER_SAFETY__` (C++11)(removed in C++23) | expands to `1` if the implementation has strict `std::pointer_safety` (macro constant) |

The values of these macros (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit. Attempts to redefine or undefine these macros result in undefined behavior.

Note: in the scope of every function body, there is a special function-local predefined variable named `__func__`, defined as a static character array holding the name of the function in implementation-defined format. It is not a preprocessor macro, but it is used together with `__FILE__` and `__LINE__`, e.g. by `assert`. (since C++11)

**Language feature-testing macros**

The standard defines a set of preprocessor macros corresponding to C++ language features introduced in C++11 or later. They are intended as a simple and portable way to detect the presence of said features. (since C++20)

See Feature testing for details.

https://en.cppreference.com/w/cpp/preprocessor/replace

GARMIN

# File and Line Info

- __LINE__
  - the current line number in this translation unit
  - incremented after every source line, including blank lines
  - excludes lines added to translation unit via #include directive
- __FILE__
  - the name of the current translation unit’s file

GARMIN®

# File and Line Info

- __LINE__
  - the current line number in this translation unit
  - incremented after every source line, including blank lines
  - excludes lines added to translation unit via #include directive
- __FILE__
  - the name of the current translation unit's file

__func__ is not a macro, but a const char* that is local to each function

GARMIN®

# File and Line Info

main.c

```c
#include <stdio.h>
int main()
{
 printf(“%s:%d\r\n”, __FILE__, __LINE__);
 return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  main.c:4
```

GARMIN.

# File and Line Info

#line <next_line_number> <file_name>

- **<next_line_number>**
  - **must evaluate to a sequence of digits**
  - **this number will be used for the line following this statement**
- **<file_name>**
  - **optional, if omitted, uses the current translation unit's file name**
  - **a quoted string specifying the new file name to use for this translation unit**

GARMIN®

# File and Line Info

main.c

```
1. #include <stdio.h>
2. int main()
3. {
4.  #line 4444 "not_my_file.c"
5.  printf("%s:%d\r\n", __FILE__, __LINE__);
6.  return 0;
7. }
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  not_my_file.c:4444
```

GARMIN.

# File and Line Info

| When to use | C Alternative | C++ Alternative |
|---|---|---|
| • If you don't have C++20 | None | C++20 adds std::source_location<br>`__LINE__ == source_location::line`<br>`__FILE__ == source_location::file_name` |

GARMIN.

# Stringification

```
#define STRINGIFY(x) #x

int main(){
 puts(STRINGIFY(44));
 return 0;
}
```

GARMIN®

# Stringification

#define <identifier>(<text>) #<text>

- **<identifier>**
  - introduce a function like macro
- **<text>**
  - the text to turn into a string
  - can be any text, but must not interfere with translation phases 1 – 3 (e.g contains \")
- **#<text>**
  - the surrounds the literal text with quotes, if <text> already contains quotes, they are escaped (e.g. `STRINGIFY_IMPL("stuff here")` becomes `"\"stuff here\""` )

GARMIN.

# Stringification

```
#define STRINGIFY(x) #x

int main(){
 puts(STRINGIFY(44));
 return 0;
}
```

GARMIN®

# Stringification

```
#define STRINGIFY(x) #x

int main(){
 puts(STRINGIFY(44));
 return 0;
}
```

```
int main(){
 puts("44");
 return 0;
}
```

GARMIN.

# Stringification

```
#define STRINGIFY(x) #x
#define VALUE 44
int main(){
 puts(STRINGIFY(VALUE));
 return 0;
}
```

GARMIN®

# Stringification

```
#define STRINGIFY(x) #x
#define VALUE 44
int main(){
 puts(STRINGIFY(VALUE));
 return 0;
}
```

```
int main(){
 puts("VALUE");
 return 0;
}
```

GARMIN

# Stringification

#define <identifier>(<text>) #<text>

- **<identifier>**
  - introduce a function like macro
- **<text>**
  - the text to turn into a string
  - can be any text, but must not interfere with translation phases 1 – 3 (e.g contains \")
- **#<text>**
  - the surrounds the literal text with quotes, if <text> already contains quotes, they are escaped (e.g. `STRINGIFY("stuff here")` becomes `"\"stuff here\""` )
  - if you want <text> evaluated, you need a 2nd level of function like macro invocation

GARMIN.

# Stringification

```
#define STRINGIFY_IMPL(x) #x
#define STRINGIFY(x) STRINGIFY_IMPL(x)
#define VALUE 44
int main(){
 puts(STRINGIFY(VALUE));
 return 0;
}
```

GARMIN®

# Stringification

```
#define STRINGIFY_IMPL(x) #x
#define STRINGIFY(x) STRINGIFY_IMPL(x)
#define VALUE 44
int main(){
 puts(STRINGIFY(VALUE));
 return 0;
}
```

```
#define STRINGIFY_IMPL(x) #x

int main(){
 puts(STRINGIFY_IMPL(44));
 return 0;
}
```

GARMIN.

# Stringification

```
#define STRINGIFY_IMPL(x) #x
#define STRINGIFY(x) STRINGIFY_IMPL(x)
#define VALUE 44
int main(){
 puts(STRINGIFY(VALUE));
 return 0;
}
```

```
#define STRINGIFY_IMPL(x) #x

int main(){
 puts(STRINGIFY_IMPL(44));
 return 0;
}
```

```
int main(){
 puts("44");
 return 0;
}
```

GARMIN

# Concatenation

```
#define CONCAT(a, b) a##b

int main(){
 int CONCAT(my_, var) = 7;
 return my_var;
}
```

GARMIN®

# Concatenation

#define <identifier>(<text1>, <text2>) <text1>##<text2>

- **<identifier>**
  - **introduce a function like macro**
- **<text1>, <text2>**
  - **the text to concatenate**
  - **can be any text, but must not interfere with translation phases 1 – 3 (e.g contains \") and must evaluate to a valid identifier (most of the time)**
- **<text1>##<text2>**
  - **concatenates the literal values of <text1> and <text2>**
  - **if you want <text1> and/or <text2> evaluated, you need a 2nd level of function like macro invocation**

GARMIN.

# Concatenation

```
#define CONCAT(a, b) a##b

int main(){
 int CONCAT(my_, var) = 7;
 return my_var;
}
```

```
int main(){
 int my_var = 7;
 return my_var;
}
```

GARMIN

# Concatenation

```
#define CONCAT_IMPL(a, b) a##b
#define CONCAT(a, b) CONCAT_IMPL(a, b)
#define SUB GFX
#define FUNC _get_screen_width

int main(){
 int width = CONCAT(SUB, FUNC)();
 return width;
}
```

GARMIN®

# Concatenation

```
#define CONCAT_IMPL(a, b) a##b
#define CONCAT(a, b) CONCAT_IMPL(a, b)
#define SUB GFX
#define FUNC _get_screen_width

int main(){
 int width = CONCAT(SUB, FUNC)();
 return width;
}
```

```
#define CONCAT_IMPL(a, b) a##b

int main(){
 int width = CONCAT_IMPL(GFX, _get_screen_width)();
 return width;
}
```

GARMIN

# Concatenation

```
#define CONCAT_IMPL(a, b) a##b
#define CONCAT(a, b) CONCAT_IMPL(a, b)
#define SUB GFX
#define FUNC _get_screen_width

int main(){
 int width = CONCAT(SUB, FUNC)();
 return width;
}
```

```
#define CONCAT_IMPL(a, b) a##b

int main(){
 int width = CONCAT_IMPL(GFX, _get_screen_width)();
 return width;
}
```

```
int main(){
 int width = GFX_get_screen_width();
 return width;
}
```

GARMIN®

# ## and #

| When to use | C Alternative | C++ Alternative |
|---|---|---|
| If you need to generate function or variable names | None | • Consider templates<br>• Future: Static Reflection/Static Generation/Metaclasses* |

*https://herbsutter.com/2017/07/26/metaclasses-thoughts-on-generative-c/

GARMIN.

# #pragma

There is no fixed format for #pragma directives, as they can vary across compilers. This is a way to control how the compiler will process code at any point in the translation process.

GARMIN®

# Include Guard

```
#ifdef MY_HEADER_HPP
#define MY_HEADER_HPP



#endif //MY_HEADER_HPP
```

GARMIN®

# Include Guard

```
#ifdef MY_HEADER_HPP
#define MY_HEADER_HPP



#endif //MY_HEADER_HPP
```

```
#pragma once

//header stuff
```

GARMIN®

# Include Guard

```
#ifdef MY_HEADER_HPP
#define MY_HEADER_HPP

//header stuff

#endif //MY_HEADER_HPP
```

```
#pragma once

//header stuff
```

- **Uses the file's location as an identifier**
  - **Can be included twice if the file reachable in multiple locations, this could be separate directories or symlinks**

GARMIN.

# The preprocessor stack

```
#define X 1
#pragma push_macro(“X”)
#undef X
#define X -1
#if (X < 0)
#define NEG_X true
#endif
#pragma pop_macro(“X”)
#if (X > 0)
#define POS_X true
#endif

int main()
{
  #if NEG_X
  puts(“X is negative”);
  #endif
  #if POS_X
  puts(“X is positive”);
  #endif
  return 0;
}
```

GARMIN.

# The preprocessor stack

```
#define X 1
#pragma push_macro(“X”)
#undef X
#define X -1
#if (X < 0)
#define NEG_X true
#endif
#pragma pop_macro(“X”)
#if (X > 0)
#define POS_X true
#endif

int main()
{
  #if NEG_X
  puts(“X is negative”);
  #endif
  #if POS_X
  puts(“X is positive”);
  #endif
  return 0;
}
```

```
X is negative
X is positive
```

GARMIN

# Warning/Error Manipulation

GCC/Clang:

#pragma <compiler> diagnostic <action> "<flag>"

- **<compiler>**
  - **Either GCC or Clang**
- **<action>**
  - **push - pushes the current diagnostic settings for later retrieval**
  - **pop - replaces the current diagnostic settings with the last pushed settings**
  - **ignored - ignores the compiler diagnostic specified by <flag>**
  - **warning - treats the compiler diagnostic specified by <flag> as a warning**
  - **error - treats the compiler diagnostic specified by <flag> as an error**
- **<flag>**
  - **Omitted when using push or pop**
  - **quoted string matching a single compiler command line flag (e.g. -Wuninitialized)**

GARMIN.

# Warning/Error Manipulation

MSVC:

```
#pragma warning(<action> : <warning-number> \
    [; <action2>:<warning-number2>...] )
#pragma warning( push, <level>)
#pragma warning(pop)
```

- **<action>**
  - **1, 2, 3, 4 - changes the default warning level for <warning-number>**
  - **default - sets the default behavior for <warning-number>**
  - **disable - disables the specified <warning-number>**
  - **error - treats <warning-number> as an error**
  - **once - only issue <warning-number> message once**
  - **suppress - disables <warning-number> for the next code line, then restore previous <action> for <warning-number>**
- **<warning-number>**
  - **MSVC warning number: (https://docs.microsoft.com/en-us/cpp/error-messages/compiler-warnings/compiler-warnings-by-compiler-version )**
- **<level>**
  - **Optionally change the current warning level in addition to saving all current warning settings**

GARMIN.

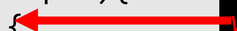
# Warning Suppression (MSVC)

```
// in some configuration somewhere
// #define NUM_DATA_FIELDS 4

void process_packet(packet* pkt){
  if(NUM_DATA_FIELDS > 2) {
    process_extended_packet(pkt);
  } else {
    process_legacy_packet(pkt);
  }
...
}
```

GARMIN®

# Warning Suppression (MSVC)

```
// in some configuration somewhere
// #define NUM_DATA_FIELDS 4

void process_packet(packet* pkt){
  if(NUM_DATA_FIELDS > 2) {
    process_extended_packet(pkt);
  } else {
    process_legacy_packet(pkt);
  }
...
}
```



warning C4127: conditional expression is constant

GARMIN®

# Warning Suppression (MSVC)

```
// in some configuration somewhere
// #define NUM_DATA_FIELDS 4

void process_packet(packet* pkt){
  // set by project config macro
  // suppress constant condition
  #pragma warning(suppress : 4127)
  if(NUM_DATA_FIELDS > 2) {
    process_extended_packet(pkt);
  } else {
    process_legacy_packet(pkt);
  }
...
}
```

GARMIN®

# MSVC and Clang(ELF) Linker control

#pragma comment (<type>, <value>)

- **<type>**
  - **compiler - adds compiler version to object file, ignored by linker**
  - **lib - adds a library to search for when linking**
  - **linker - command line option to pass to linker**
- **<value>**
  - **compiler - causes an error if <value> specified**
  - **lib - the path of the library to add**
  - **linker - the command to pass to the linker**

GARMIN.

# MSVC and Clang(ELF) Linker control

#pragma comment (<type>, <value>)

```
#ifdef __WIN64
#pragma comment(lib, "mylib.lib");
#elif __linux__ && __clang__
#pragma comment(lib, "mylib.a");
#endif
```

- **<type>**
  - **compiler - adds compiler version to object file, ignored by linker**
  - **lib - adds a library to search for when linking**
  - **linker - command line option to pass to linker**
- **<value>**
  - **compiler - causes an error if <value> specified**
  - **lib - the path of the library to add**
  - **linker - the command to pass to the linker**

GARMIN.

# Struct Packing

#pragma pack( <N> )

#pragma pack(push, <N>)

#pragma pack(pop)

- **pack(<N>)**
  - **Change alignment to N byte alignment**
- **pack(push, <N>)**
  - **pushes the current alignment to the alignment stack, then sets the alignment to N bytes.**
- **pack(pop)**
  - **restores the alignment before the most recent push**

GARMIN.

# Struct Packing (MSVC extension)

#pragma pack(push, <ID>, <N>)

#pragma pack(pop, <ID>)

- **push <ID>**
  - **Associates the current alignment with <ID> then change alignment to N byte alignment**
- **pop <ID>**
  - **pops all push calls until the stack record with <ID> is found, then restores the byte alignment associated with <ID> and pops it**
  - **if <ID> is NOT found, the call to pop is ignored**

GARMIN.

# Struct Packing *BEWARE!*

- Packing lasts until the end of the current translation unit, this can mean it is applied to structs defined in other headers through #include
- This should be used for serialization, or extreme memory constraints only, since it can mess with optimization and cause extra instructions for unaligned access
- Affects the alignment of items in a struct that has a packed struct as a member

GARMIN®

# A Preprocessor Directive

#<directive> <stuff>

- **#**
  - **introduces a <directive>**
  - **# can be followed by 0 or more spaces**
  - **# followed by a newline is a no-op**
  - **# must be the first non-whitespace character**
- **<directive>**
  - **we'll see these later**
- **<stuff>**
  - **depends on the <directive>, more on this later**
  - **cannot introduce another directive either via # or via macro expansion**

GARMIN.

# A Preprocessor Directive

#<directive> <stuff>

- #
  - introduces a <directive>
  - # can be followed by 0 or more spaces
  - # followed by a newline is a no-op
  - # must be the first non-whitespace character
- <directive>
  - we'll see these later
- <stuff>
  - depends on the <directive>, more on this later
  - cannot introduce another directive either via # or via macro expansion

GARMIN

# _Pragma

**Allows anything that could be used with #pragma to be part of a preprocessor directive's stuff**

```
#if PACK
#define PACK_LEVEL _Pragma("pack (push, 1)")
#else
#define PACK_LEVEL
#endif

PACK_LEVEL struct S {
    char c;
    int i;
};
```

GARMIN.

# _Pragma

**Allows anything that could be used with #pragma to be part of a preprocessor directive's stuff**

```
#if PACK
#define PACK_LEVEL _Pragma("pack (push, 1)")
#else
#define PACK_LEVEL
#endif

PACK_LEVEL struct S {
    char c;
    int i;
};
```

```
// -DPACK=TRUE
#pragma pack(push, 1)
struct S {
    char c;
    int i;
};
```

GARMIN

# #pragma and _Pragma

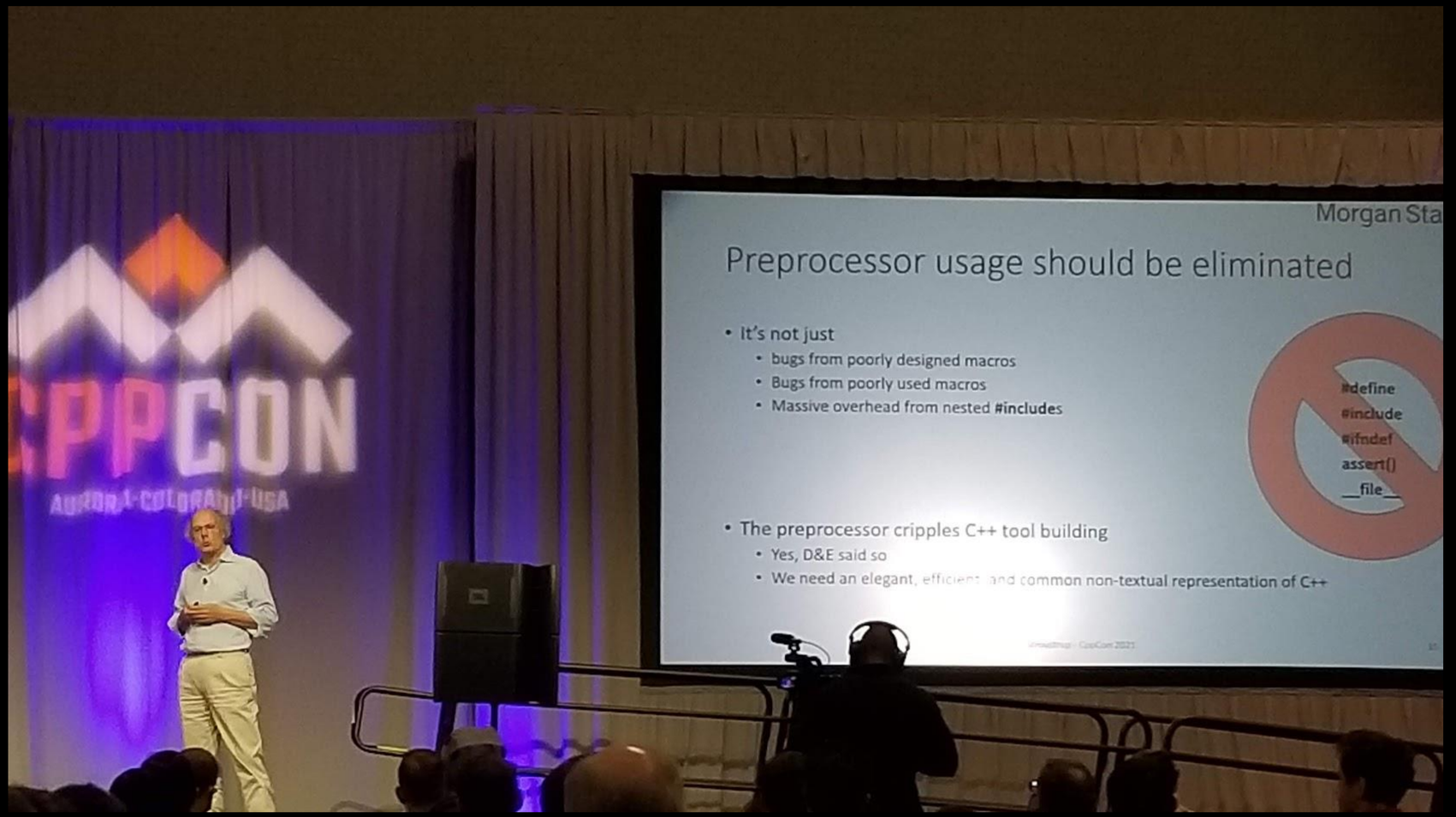
| When to use | C Alternative | C++ Alternative |
|---|---|---|
| Limit use as much as possible | See your compiler:<br>__attribute__<br>__declspec<br>__packed__ | Same as C |

*"I'd like to see the C preprocessor abolished. However, the only realistic and responsible way of doing that is first to make it redundant, then, encourage people to use the better alternatives, and then - years later - banish [it]"*

–Bjarne Stroustrup

Design and Evolution of C++

GARMIN.


CPPCON
Morgan Sta
Preprocessor usage should be eliminated
• It's not just
• bugs from poorly designed macros
• Bugs from poorly used macros
• Massive overhead from nested #includes
#define
#include
#ifndef
assert()
__file__
• The preprocessor cripples C++ tool building
• Yes, D&E said so
• We need an elegant, efficient, and common non-textual representation of C++

GARMIN.

# Questions?

GARMIN®