The Preprocessor

Everything you need to know, and more!
What happens when you write this?

```c
#include "my_header.h"
```
```c
#include "my_header.h"

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

typedef enum {
    OR_LEFT,
    OR_RIGHT,
    OR_CENTER
} orientation;

point get_layout_left(orientation);

#include "my_header.h"

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

typedef enum {
  OR_LEFT,
  OR_RIGHT,
  OR_CENTER
} orientation;

point get_layout_left(orientation);

#include "my_header.h"

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

```c
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;
}
```
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!
About this talk
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.
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.
Remember, everything in the preprocessor happens BEFORE the compiler gets involved.
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
What’s a Directive?

A directive can be one of the following:

• **Include**
  • #include

• **Conditional**
  • #if, ifndef, ifdef, else, elif, endif, elifdef, elifndef

• **Replace**
  • define, undef, #, ##

• **Error**
  • #error

• **Compiler Extension**
  • pragma, _Pragma

• **File and Line Information**
  • #line
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*
including another file

- `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

#include file_to_include
#include

<table>
<thead>
<tr>
<th>When to use</th>
<th>C Alternative</th>
<th>C++ Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you don't have module support</td>
<td>None</td>
<td>C++20 modules</td>
</tr>
</tbody>
</table>
View preprocessor output

You can view the output after the preprocessor runs

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Complier Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC/Clang</td>
<td>-E -o output.i</td>
</tr>
<tr>
<td>MSVC</td>
<td>/P /Fioutput.i</td>
</tr>
</tbody>
</table>
Object like macros

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

int main()
{
    return (int)CIRCUMFERENCE;
}
```
Object like macros

#define <identifier> <replacement>

• define
  • Introduces an identifier to the preprocessor

#undef <identifier>

• undef
  • Removes an identifier from 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
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);
}
```
Object like macros

```c
#define PI 3.14159
#define RADIUS 2

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

```c
int main()
{
    return (int)(3.14159 * 2);
}
```
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 are 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`, `cerfl`, `csqrt`, `csqrtl`, `clog`, `clogl`, `clog2`, `clog2l`, `clgamma`, and their f and l suffixed variants, in `<cmath.h>`
  - beginning with is or te followed by a lowercase letter, in `<ctype.h>` and `<wctype.h>`
  - beginning with str followed by a lowercase letter, in `<cstdlib.h>`
  - beginning with str, mnr 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 cmdbuf_, mtx_, thrd_ or tss_ followed by a lowercase letter, in `<threads.h>`

- Typedef names:
  - beginning with int or uint and ending with _t, in `<stdlib.h>`
  - beginning with atomic_ or memory_ followed by a lowercase letter, in `<stdatomic.h>`
  - beginning with cmdbuf_, 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 `<errno.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 `<stdlib.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 `<ctime.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 cmdbuf_, mtx_, thrd_ or tss_ followed by a lowercase letter, in `<threads.h>`

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 are reserved identifiers and allow the library to use numerous behind-the-scenes implementation details.

4) All external identifiers defined by the standard library (in headers) or supplied as 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`, `exp2`, `cos`, `log10`, `log1p`, `clog2`, `clgamma`, `tgamma` and other math and trigonometric functions, in `<cmath>`.
     - Beginning with `is` or `is` followed by a lowercase letter, in `<typeinfo>`.
     - Beginning with `str` followed by a lowercase letter, in `<stdlib.h>`.
     - Beginning with `str`, `strm`, or `wcs` followed by a lowercase letter, in `<wchar.h>`.
     - Beginning with `wcs` followed by a lowercase letter, in `<wchar.h>`.
     - Beginning with `atomic` followed by a lowercase letter, in `<cstdlib.h>`.
     - Beginning with `nfd`, `_txt`, `_thrd`, or `tss` followed by a lowercase letter, in `<cstdlib.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 `nfd`, `_txt`, `_thrd`, or `tss` followed by a lowercase letter, in `<cstdlib.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 `<errno.h>`.
     - Beginning with `INT` or `UINT` and ending with `_MAX`, `_MIN`, or `_C`, in `<stdint.h>`.
     - Beginning with `PRI` or `SCN` followed by a lowercase letter or the letter `X`, in `<stdio.h>`.
     - Beginning with `LC` followed by a lowercase letter, in `<locale.h>`.
     - Beginning with `SIG` or `SIGF` 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 `<cstdlib.h>`.
   - `enumeration` constants:
     - Beginning with `memory_order` followed by a lowercase letter, in `<stdatomic.h>`.
     - Beginning with `cn`, `_txt`, `_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)
### Object like macros

<table>
<thead>
<tr>
<th>When to use</th>
<th>C Alternative</th>
<th>C++ Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>• When you need a value for use during the pre-processing phase&lt;br&gt;• When you need to support multiple compiler capabilities&lt;br&gt;• Command line flags</td>
<td>For constant expressions use enums&lt;br&gt;<code>enum {&lt;br&gt;  ARRAY_SIZE = 44&lt;br&gt;}</code>&lt;br&gt;<code>char arr[ARRAY_SIZE];</code></td>
<td>use constexpr or static const values</td>
</tr>
</tbody>
</table>
Allow command line flags

- 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

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

void debug_print([[maybe_unused]] char* str){
    if constexpr (DEBUG_BUILD) {
        std::cout << str;
    }
}
```
Directives do not have scope, the file is processed top down. So, moving directives can change generated code.
Macros have scoping issues. The MY_VAL macro is defined in the blorp function, but it is also used in the main function. When the blorp function is called, it will use the local MY_VAL defined in the blorp function, which is 7. However, when the other_fun function is called, it will use the MY_VAL defined in the main function, which is 9. This leads to different outputs for the same function call, depending on where the macro is defined.
Macro scoping issues

```
#include <stdio.h>

void blorp()
{
    #define MY_VAL 7
    printf("%s: %d\n", __FUNCTION__, MY_VAL);
}

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

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

Output:  
main: 9  
blorp: 7  
other_fun: 9  
main: 9
#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
}
Conditionals

• `<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>`

```cpp
#if <expr>
  //code
#elif <expr>
  //code
#else
  //code
#endif
```
#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
}
main.cpp

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

#ifdef ASSET_SUPPORT
return asset_get_count();
#else
return 0;
#endif

int main()
{

}
```

pub_config.h

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

pub.h

```c
#ifndef PUB_H
#define PUB_H
#include "pub_config.h"
#define ASSET_SUPPORT...
#endif
```
main.cpp

```c
#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

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

pub.h

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

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

```c
#include "pub_config.h"

int asset_get_count();
```

```c
#include "pub_config.h"

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

```c
#include "pub_config.h"

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

```c
// 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
```
```c
#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
}
```

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

```c
#ifndef PUB_CONFIG_H
#define PUB_CONFIG_H
...
#define PUB_SUPPORTED TRUE
...
#endif
```
main.cpp
#define PUB_CONFIG_H 
...
define PUB_SUPPORTED 1 
...
define PUB_H 
#elifdef PUB_CONFIG_H 
define PUB_CONFIG_H 
...
define PUB_SUPPORTED TRUE 
... 
#else 
define ASSET_SUPPORT 
...
def asset_get_count(); 
...
int main() 
{ 
    #ifdef ASSET_SUPPORT 
    return asset_get_count(); 
    #else 
    return 0; 
    #endif 
}
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
}
# define ASSET_SUPPORT
...
int asset_get_count();
...

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

main.cpp

```cpp
int asset_get_count();

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

main.cpp

```cpp
... int asset_get_count(); ...
...
int main()
{
    return asset_get_count();
}
```
## Conditionals

<table>
<thead>
<tr>
<th>When to use</th>
<th>C Alternative</th>
<th>C++ Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Include guards</td>
<td>Attempt to restructure code to limit inline code exclusion</td>
<td>• use constexpr if when all code is available</td>
</tr>
<tr>
<td>• C linkage specification</td>
<td></td>
<td>• module imports</td>
</tr>
<tr>
<td>• Very sparingly, to exclude code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Check for header or feature availability flags</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What about if constexpr?
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);
    }
}
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
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
#define BARK_USED 1

```c
#if BARK_USED
void proc_bark(void* buf){ ... }
#else
t 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
Check for files or attributes

```c
#if __has_include("myHeader.hpp")
#include "myHeader.hpp"
#else
static void flarb(int){}
#endif
```
Check for files or attributes

```c
#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);
}
```
Check for files or attributes

```c
#include "myHeader.hpp"
#endif
#include "myHeader.hpp"
#endif
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);
}
```
Code exclusion **ANTIPATERNs**

Excluding enum values

```c
enum struct PageMode {
    Visible,
    Enabled,
#if (DARK_MODE_ENABLED)
    DarkMode,
#endif
    #endif
    Highlighted
};
```
Excluding enum values

```c
enum struct PageMode {
    Visible,
    Enabled,
    #if (DARK_MODE_ENABLED)
    DarkMode,
    #endif
    Highlighted
};
```

Changing flow control

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

**ANTIPATERNs**

Excluding enum values

```c
enum struct PageMode {
    Visible,
    Enabled,
#if (DARK_MODE_ENABLED)
    DarkMode,
#endif
    Highlighted
};
```

Changing flow control

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

Excluding members

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

```c
#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);
```
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 ;
Function like macros

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

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

Output:
3.2
Function like macros - use ()

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

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

Output:
  5
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
# Function like macros

<table>
<thead>
<tr>
<th>When to use</th>
<th>C Alternative</th>
<th>C++ Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Perform more complicated pre-processor operations</td>
<td>Create multiple functions for each type with _Generic</td>
<td>• constexpr functions</td>
</tr>
<tr>
<td>• Generate blocks of code</td>
<td></td>
<td>• templates</td>
</tr>
</tbody>
</table>
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^=
```cpp
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);
```

```cpp
#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; };
```
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
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
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__`

```c
#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*

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

- Either of these options make our previous example change to

```c
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
## Variadic function like macros

<table>
<thead>
<tr>
<th>When to use</th>
<th>C Alternative</th>
<th>C++ Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Perform more complicated pre-processor operations</td>
<td>variadic functions with &lt;stdargs.h&gt;</td>
<td>variadic templates</td>
</tr>
<tr>
<td>• Absorb all parameters and turn into noop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some things you may see...
“Inheritance” in C

```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;
}
```
“Inheritance” in C

```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;
}
```
“Overloading” in C11*

```c
#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++
“Overloading” in C11*

```c
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++
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++
“Overloading” in C11*

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

* This will not compile in standard C++
What's with the do...while loop?

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

if(result == FAILED)
  printf("[%lu](%s) ", get_timer(), __func__); \
  printf(__VA_ARGS__); \
  printf("\n\n")
else
  set_state(STATE_IDLE);
No loop...oops

```c
#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);
```

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

if (result == FAILED) {
    printf("[%lu](%s)", get_timer(), __func__);
    printf(__VA_ARGS__);
    printf("\r\n");
} else {
    set_state(STATE_IDLE);
}
if(result == FAILED)
    debug_printf("FAILED");
else
    set_state(STATE_IDLE);

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

if(result == FAILED)
{
    printf("[%lu](%s ", get_timer(), "proc_state");
    printf("FAILED");
    printf("\r\n");
    ;
else
    set_state(STATE_IDELE);
So, do...while(0)

```c
#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);
```
Predefined Preprocessor Identifiers

C

Predefined macros
The following macro names are predefined in any translation unit:

C

C++

Predefined macros
The following macro names are predefined in every translation unit.

...
Predefined Preprocessor Identifiers

C

Predefined macros

The following macro names are predefined in any translation unit with

C

macro name

_value

value

The following macro names are predefined in any translation unit with

C

macro name

_value

value

C++

Predefined macros

The following macro names are predefined in any translation unit with

C

macro name

_value

value

C

macro name

_value

value

C++

https://en.cppreference.com/w/c/preprocessor/replace
https://en.cppreference.com/w/cpp/preprocessor/replace
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
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
```c
#include <stdio.h>

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

int main()
{
    #line 4444 "not_my_file.c"
    printf("%s:%d\n", __FILE__, __LINE__);
    return 0;
}

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
    not_my_file.c:4444
## File and Line Info

<table>
<thead>
<tr>
<th>When to use</th>
<th>C Alternative</th>
<th>C++ Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If you don't have C++20</td>
<td>None</td>
<td>C++20 adds std::source_location</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>LINE</em> == source_location::line</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>FILE</em> == source_location::file_name</td>
</tr>
</tbody>
</table>
#define STRINGIFY(x) #x

int main()
{
    puts(STRINGIFY(44));
    return 0;
}
• `<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\""`)
Stringification

```c
#define STRINGIFY(x) #x

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

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

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

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

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

```c
#define STRINGIFY_IMPL(x) #x

int main(){
    puts(STRINGIFY_IMPL(44));
    return 0;
}
```
#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;
}
#define CONCAT(a, b) a##b

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

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

int main(){
    int CONCAT(my_, var) = 7;
    return my_var;
}
```
#define CONCAT_IMPL(a, b) a##b
#define CONCAT(a, b) CONCAT_IMPL(a, b)
#define SUB GFX
#define FUNC _

```c
#include <stdio.h>

int get_screen_width() {
    return 1280;
}

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

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

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

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

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

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

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

```c
int main(){
    int width = GFX_get_screen_width();
    return width;
}
```
When to use | C Alternative | C++ Alternative
---|---|---
If you need to generate function or variable names | None | • Consider templates
• Future: Static Reflection/Static Generation/Metaclasses*

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

#ifdef MY_HEADER_HPP
#define MY_HEADER_HPP

#endif //MY_HEADER_HPP
Include Guard

```c
#ifdef MY_HEADER_HPP
#define MY_HEADER_HPP

//header stuff

#endif //MY_HEADER_HPP
```

```c
#pragma once
```
#ifdef MY_HEADER_HPP
#define MY_HEADER_HPP

//header stuff
#endif //MY_HEADER_HPP

#pragma once

//header stuff
The preprocessor stack

```c
#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;
}
```
The preprocessor stack

```c
#define X 1
#pragma push_macro("X")
#undef X
#define X -1
#if (X < 0)
#define NEG_X true
#endif
#pragma pop_macro("X")
#define 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;
}
```
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)
Warning/Error Manipulation

MSVC:

```c
#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>**

- **<level>**
  - Optionally change the current warning level in addition to saving all current warning settings
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);
    }
    ...
}
// 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);
    }
    ...
}
void process_packet(packet* pkt){
    // set by project config macro
    // suppress constant condition
    //define NUM_DATA_FIELDS 4
    #pragma warning(suppress : 4127)
    if(NUM_DATA_FIELDS > 2) {
        process_extended_packet(pkt);
    } else {
        process_legacy_packet(pkt);
    }
    ...
}
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
#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
• 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

#pragma pack(<N>)
#pragma pack(push, <N>)
#pragma pack(pop)
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
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.
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
A Preprocessor Directive

```plaintext
#<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
```
_Pragma

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

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

PACK_LEVEL struct S {
    char c;
    int i;
};
```
#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;
};
```
#pragma and _Pragma

<table>
<thead>
<tr>
<th>When to use</th>
<th>C Alternative</th>
<th>C++ Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit use as much as possible</td>
<td>See your compiler: <strong>attribute</strong> __declspec <strong>packed</strong></td>
<td>Same as C</td>
</tr>
</tbody>
</table>
"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++
Preprocessor usage should be eliminated

- It's not just
  - bugs from poorly designed macros
  - Bugs from poorly used macros
  - Massive overhead from nested includes

- The preprocessor cripples C++ tool building
  - Yes, D&E said so
  - We need an elegant, efficient and common non-textual representation of C++
Questions?