# □□□□□□□ □□ □□□

□□□□□ □□□□□□□□

□□□□□□ □□□□□□□□ □□□□□□□□□□ □□□□□□□□

□□□□□□□□□ □□□□□□ □□□

# Unicode in C++

James McNellis (@JamesMcNellis)

Senior Software Development Engineer

Microsoft Visual C++

# Before there was Unicode…

# Single-Byte Encodings

# ASCII

!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~

# ASCII

| H | e | l | l | o | ! | \0 |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 48 | 65 | 6C | 6C | 6F | 21 | 00 |

# ASCII

- A 7-bit character encoding
- 32 control characters
- 95 printable characters
- Near-ubiquitous, but often with substitutions or extensions
- Great for English; not so great for most other languages

# Extended ASCII

- IBM code pages, e.g. CP437 (“Latin US”), CP737 (“Greek”)
- Mac OS Roman
- DEC Multinational Character Set
- ISO/IEC 8859
- …and many more

# Code page 437

# ISO/IEC 8859

- Latin-1 Western European
- Latin-2 Central European
- Latin-3 South European
- Latin-4 North European
- Latin/Cyrillic
- Latin/Arabic
- Latin/Greek
- Latin/Hebrew
- Latin-5 Turkish
- Latin-6 Nordic
- Latin/Thai
- Latin-7 Baltic Rim
- Latin-8 Celtic
- Latin-9 (revision of Latin-1)
- Latin-10 South-Eastern European

# ISO/IEC 8859-1 (“Latin 1”)

¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿

ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß

àáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ

# ISO/IEC 8859-5 (“Latin/Cyrillic”)

ЁЂЃЄЅІЇЈЉЊЋЎЏАБВГДЕЖЗИЙКЛМНОП

РСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп

рстуфхцчшщъыьэюя№ёђѓєѕіїјљњћќ§ўџ

# ISO/IEC 8859-5 (“Latin/Cyrillic”)

| А | л | л | о | \0 |
|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ |
| B0 | DB | DB | DE | 00 |

# ISO/IEC 8859-5 (“Latin/Cyrillic”)

| B0 | DB | DB | DE | 00 |
|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ |
| А | л | л | о | \0 |

# But You Have to be Careful…

| B0 | DB | DB | DE | 00 |
|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ |
| ° | Ÿ | Ÿ | ή | \0 |

# Single-Byte Encodings

- Many nice properties…
  - Each character is the same size
  - The encodings are compact
  - String operations are generally straightforward
- …but there aren't enough code points to represent all characters
  - Different encodings can be used for different sets of characters…
  - …but this doesn't work for all languages,
  - …and it makes text interchange difficult

# Variable-Length Encodings

# Shift-JIS

- Some characters are representable using a single byte; others require two bytes
- Two-byte characters consist of a lead byte and a trail byte
  - The lead byte will always have the high bit set; the trail byte may have any value
- Starts from 7-bit ASCII, with a couple of substitutions (replaces \ with ¥ and ~ with ‾)
- The encoding form is very complex due to overlap

# Shift-JIS

http://en.wikipedia.org/wiki/Shift_JIS

First byte

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | SP | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | ¥ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ‾ | DEL |
| 8 | | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | | |
| A | | 。 | 「 | 」 | 、 | ・ | ヲ | ァ | ィ | ゥ | ェ | ォ | ャ | ュ | ョ | ッ |
| B | ー | ア | イ | ウ | エ | オ | カ | キ | ク | ケ | コ | サ | シ | ス | セ | ソ |
| C | タ | チ | ツ | テ | ト | ナ | ニ | ヌ | ネ | ノ | ハ | ヒ | フ | ヘ | ホ | マ |
| D | ミ | ム | メ | モ | ヤ | ユ | ヨ | ラ | リ | ル | レ | ロ | ワ | ン | ゛ | ゜ |
| E | | | | | | | | | | | | | | | | |
| F | | | | | | | | | | | | | | | | |

Second byte

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | | | | | | | | | |
| 1 | | | | | | | | | | | | | | | | |
| 2 | | | | | | | | | | | | | | | | |
| 3 | | | | | | | | | | | | | | | | |
| 4 | | | | | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | | |
| 9 | | | | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | | | | |
| B | | | | | | | | | | | | | | | | |
| C | | | | | | | | | | | | | | | | |
| D | | | | | | | | | | | | | | | | |
| E | | | | | | | | | | | | | | | | |
| F | | | | | | | | | | | | | | | | |

# Shift-JIS

```
44       → D (Latin Capital D)
84 44    → Д (Cyrillic Capital De)
84 84    → т (Cyrillic Capital Te)
```

# Shift-JIS

| 44 | 44 | 84 | 84 | 84 | 84 | 84 | 44 |
|---|---|---|---|---|---|---|---|
| ↓ | ↓ | | ↓ | | ↓ | | ↓ |
| D | D | | т | | т | | Д |

# Shift-JIS

… 84 84 84 84 84 84 84 …

p

# Shift-JIS

- Note that this is just one technique; there are others
- E.g., using escape sequences to switch between "working sets" of characters
  - See, for example, ISO/IEC 2022

# Multi-Byte Encodings

- Substantially expanded code space
  - With two bytes, up to 32,896 characters theoretically representable…
  - …though real-world encodings tend to have lower limits
- Some byte-oriented string operations still work, e.g. strcpy
- Many disadvantages:
  - Complex to parse (though it does depend on the particular encoding)
  - Some common string operations require a linear scan over the string
  - Many simple string operations become difficult or require special APIs
  - Still have the problem that there are many different encoding standards

# Unicode 1.0

# Unicode Design Goals

- One Character Encoding to Rule Them All…

- **Universal:** Must be able to represent all characters likely to be used in text interchange
- **Efficient:** Plain text should be simple to parse
- **Unambiguous:** Any given Unicode code point always represents the same character

- A set of characters, not glyphs (so, not concerned with visual representation)

# Unicode 1.0

- Each character is mapped to a 16-bit code point
  - Up to 65,536 characters can be represented (in theory; fewer in practice)
- All of the characters in ISO/IEC 8859-1 (“Latin- 1”) map to the same code points
- The UCS-2 encoding is used:  each code point is mapped to a single 16-bit code unit

# Unicode 1.0

| H | e | l | l | o | ! | \0 |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| U+0048 | U+0065 | U+006C | U+006C | U+006F | U+0021 | U+0000 |

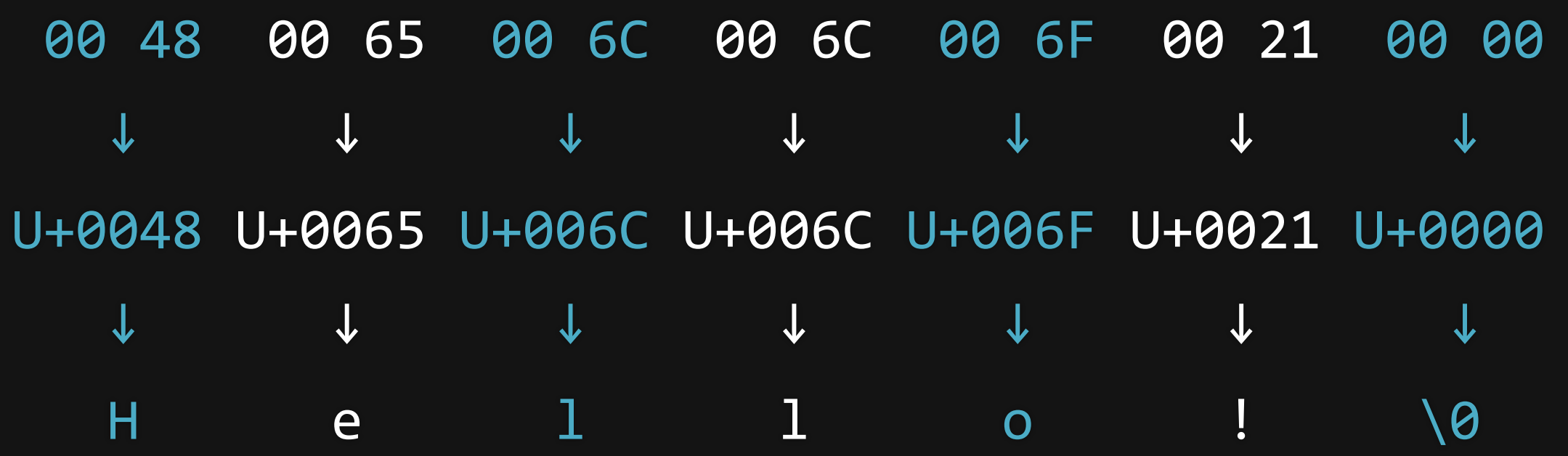
# Unicode 1.0

| H | e | l | l | o | ! | \0 |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| U+0048 | U+0065 | U+006C | U+006C | U+006F | U+0021 | U+0000 |

# Unicode 1.0

| X | Δ | Ж | ヸ | ヶ | ☃ | ڳ |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| U+0058 | U+0394 | U+0436 | U+30F8 | U+183C | U+2603 | U+06B3 |

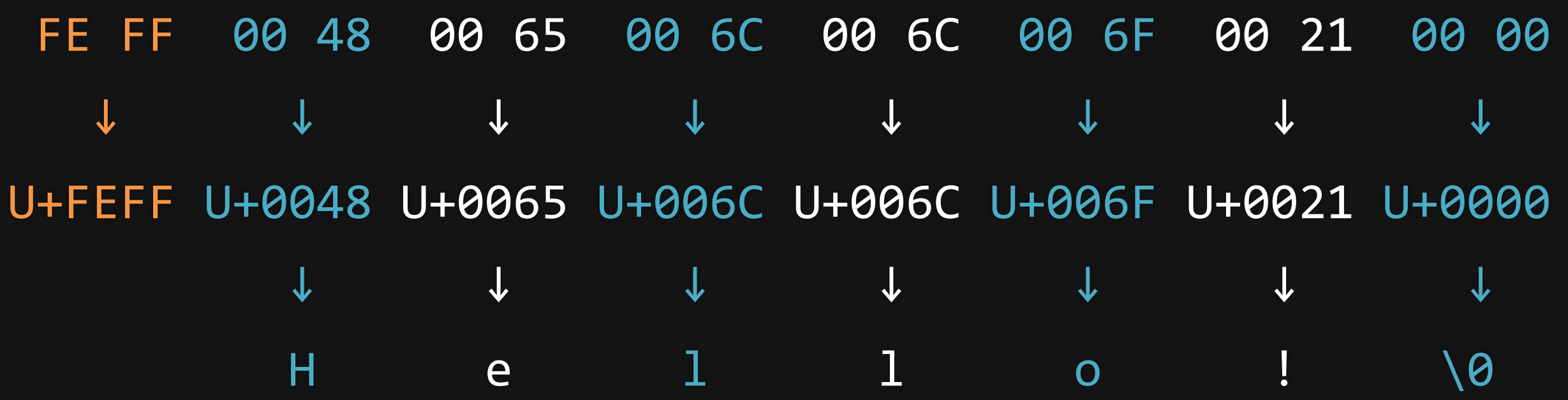
# UCS-2


00 48 00 65 00 6C 00 6C 00 6F 00 21 00 00
U+0048 U+0065 U+006C U+006C U+006F U+0021 U+0000
H e l l o ! \0

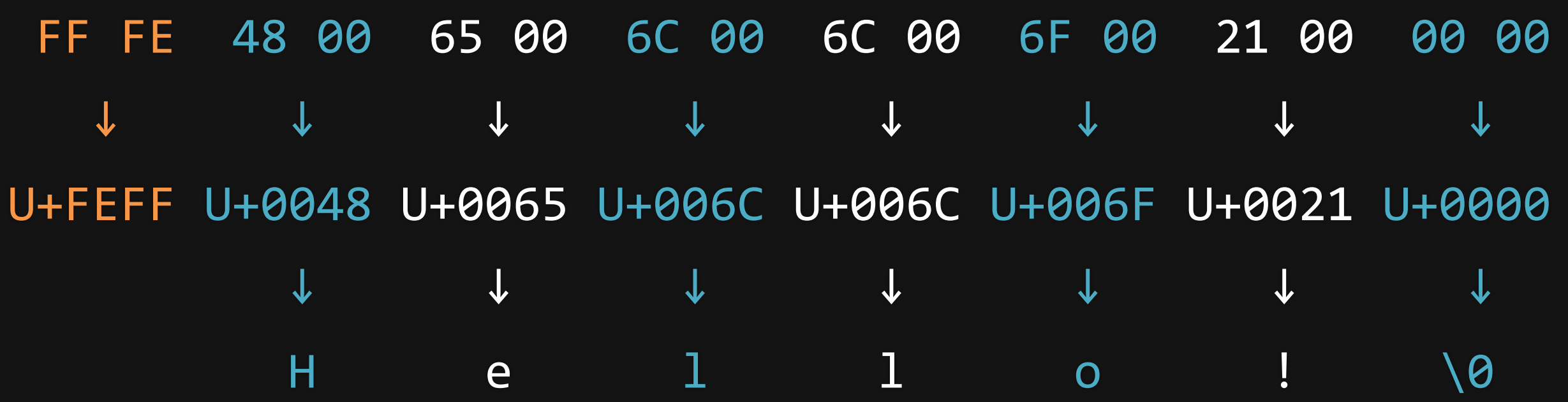
# UCS-2

| 00 48 | 00 65 | 00 6C | 00 6C | 00 6F | 00 21 | 00 00 |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| U+4800 | U+6500 | U+6C00 | U+6C00 | U+6F00 | U+2100 | U+0000 |
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 蹄 | 攀 | 氈 | 氈 | 褽 | ℅ | \0 |

# UCS-2


FE FF 00 48 00 65 00 6C 00 6C 00 6F 00 21 00 00
U+FEFF U+0048 U+0065 U+006C U+006C U+006F U+0021 U+0000
H e l l o ! \0

# UCS-2


FF FE 48 00 65 00 6C 00 6C 00 6F 00 21 00 00 00
U+FEFF U+0048 U+0065 U+006C U+006C U+006F U+0021 U+0000
H e l l o ! \0

# UCS-2

- Advantages:
  - A huge number of characters are representable
  - Characters from different scripts are easily combinable in a single string
  - Each code point is representable using a single code unit, so the encoding is simple
- Disadvantages:
  - Multiple possible byte orderings, so byte order mark (BOM) is required for interchange
  - Every character requires two bytes, so text for many languages require twice as much storage
  - None of the byte-oriented string functions (like strcpy) work with UCS-2 strings

# Code Space Usage

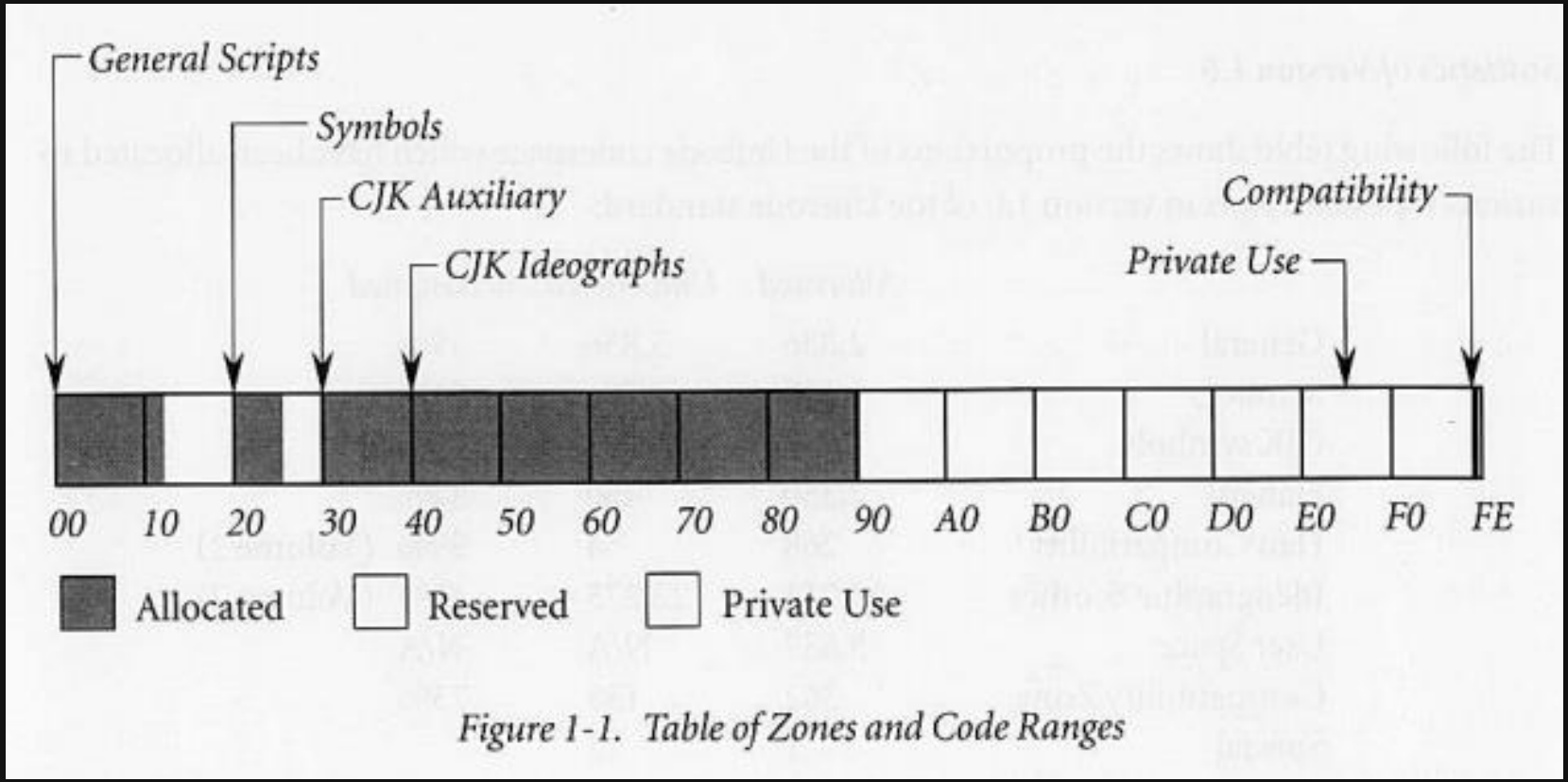


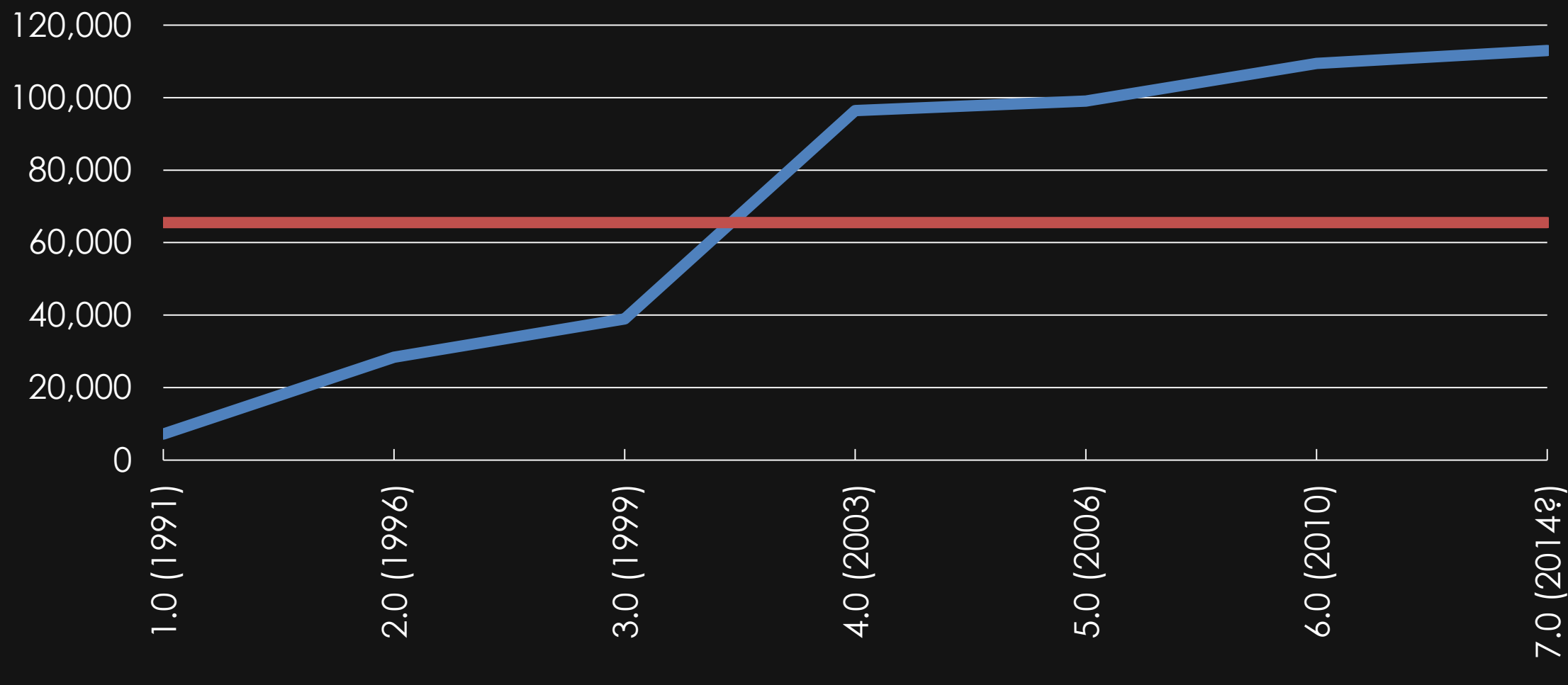

*Figure 1-1. Table of Zones and Code Ranges*

# Code Space Usage

"With over 30,000 unallocated character positions, the Unicode character encoding provides sufficient space for foreseeable future expansion."

– Unicode 1.0 Introduction

# Number of Characters



# Unicode Encodings Today

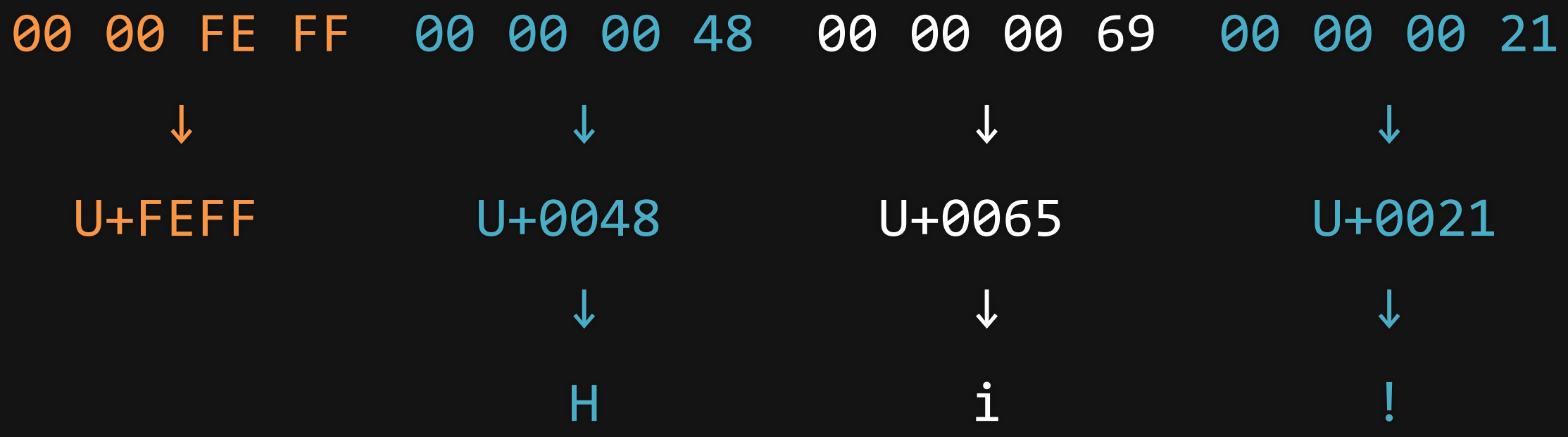
# Expanding the Code Space

- It turns out we need to be able to encode more than 65,536 characters
- In Unicode 2.0, the code space was expanded
  - Total code points:  1,112,064
  - Maximum number of characters:  1,111,998
- We can't store that many distinct values using 16 bits, so the encoding has to change too
- If 16 bits isn't enough space, how about 32 bits?

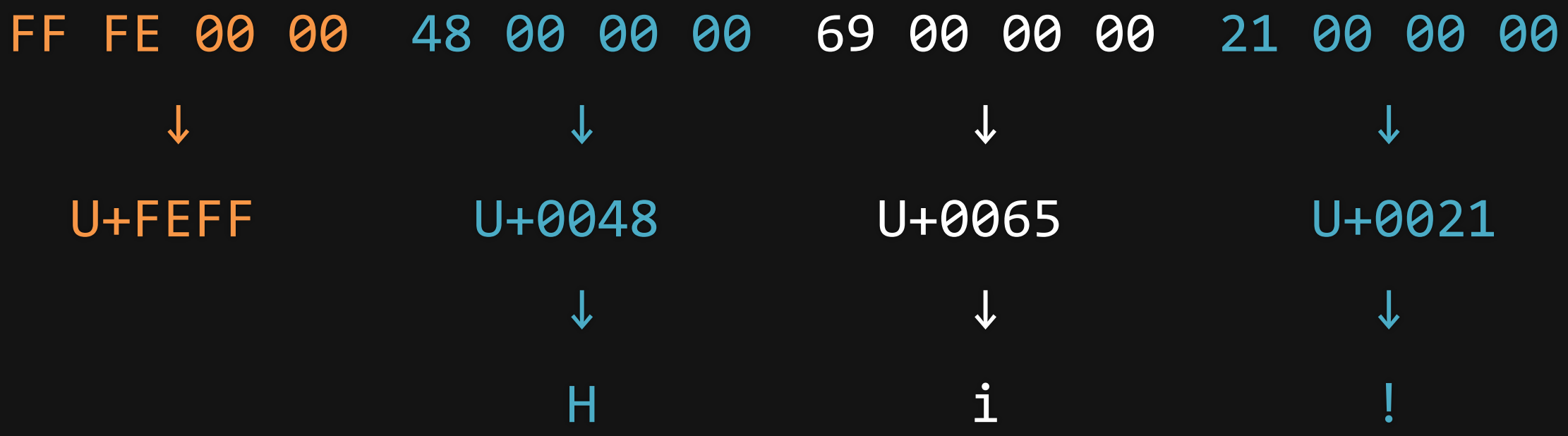
# UTF-32

| 00 00 00 48 | 00 00 00 69 | 00 00 00 21 |
| --- | --- | --- |
| ↓ | ↓ | ↓ |
| U+0048 | U+0065 | U+0021 |
| ↓ | ↓ | ↓ |
| H | i | ! |

# UTF-32


00 00 FE FF
00 00 00 48
00 00 00 69
00 00 00 21
U+FEFF
U+0048
U+0065
U+0021
H
i
!

# UTF-32


FF FE 00 00
48 00 00 00
69 00 00 00
21 00 00 00
U+FEFF
U+0048
U+0065
U+0021
H
i
!

# UTF-32

| FF FE 00 00 | 48 00 00 00 | 69 00 00 00 | 30 F4 01 00 |
| --- | --- | --- | --- |
| ↓ | ↓ | ↓ | ↓ |
| U+FEFF | U+0048 | U+0065 | U+1F430 |
| | ↓ | ↓ | ↓ |
| | H | i |  |

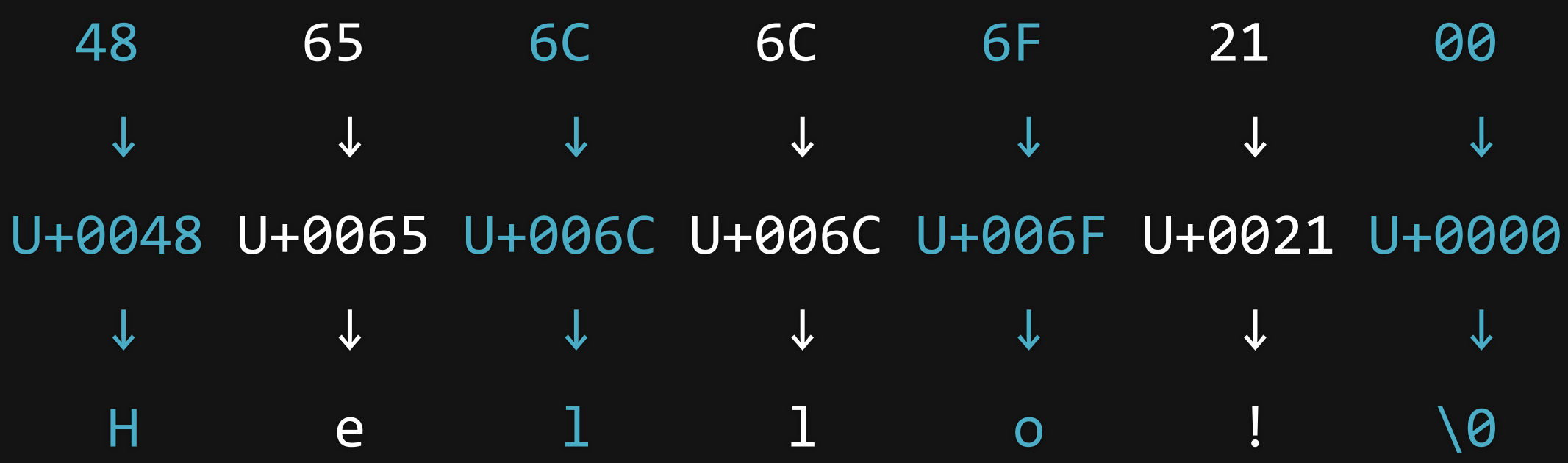
# UTF-32

- Advantages:
  - A huge number of characters are representable
  - Each code point is representable using a single code unit, so the encoding is simple
  - …this isn't all that useful in most practical string usage; we'll see why later…
- Disadvantages:
  - Two possible byte orderings, so byte order mark (BOM) is required
  - Every character requires four bytes; wasting at least 11 bits per code point
  - None of the byte-oriented string functions (like strcpy) work with UTF-32 strings
  - Nothing that was written to work with UCS-2 works with UTF-32

# UTF-8

| Bits of Code Point | First Code Point | Last Code Point | Bytes in Sequence | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
|---|---|---|---|---|---|---|---|
| 7 | U+0000 | U+007F | 1 | 0xxxxxxx | | | |
| 11 | U+0080 | U+07FF | 2 | 110xxxxx | 10xxxxxx | | |
| 16 | U+08000 | U+FFFF | 3 | 1110xxxx | 10xxxxxx | 10xxxxxx | |
| 21 | U+10000 | U+10FFFF | 4 | 11110xxx | 10xxxxxx | 10xxxxxx | 10xxxxxx |

# UTF-8


48 65 6C 6C 6F 21 00
U+0048 U+0065 U+006C U+006C U+006F U+0021 U+0000
H e l l o ! \0

# UTF-8

| 58 | CE 94 | D0 B6 | E3 83 B8 | E1 A0 BC | F0 9F 90 B0 | 00 |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| U+0058 | U+0394 | U+0436 | U+30F8 | U+183C | U+1F430 | U+0000 |
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| X | Δ | Ж | ヸ | ᠼ |  | \0 |

# UTF-8

| Bits of Code Point | First Code Point | Last Code Point | Bytes in Sequence | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
|---|---|---|---|---|---|---|---|
| 7 | U+0000 | U+007F | 1 | 0xxxxxxx | | | |
| 11 | U+0080 | U+07FF | 2 | 110xxxxx | 10xxxxxx | | |
| 16 | U+08000 | U+FFFF | 3 | 1110xxxx | 10xxxxxx | 10xxxxxx | |
| 21 | U+10000 | U+10FFFF | 4 | 11110xxx | 10xxxxxx | 10xxxxxx | 10xxxxxx |

# UTF-8

- Advantages:
  - ASCII text has the same representation in UTF-8
  - No byte order mark (BOM) is required, though there is an optional BOM (EF BB BF)
  - Many byte-oriented string functions (strcpy, strcat, strlen, etc.) work with UTF-8 strings
  - UTF-8 encoded text requires less storage than 16-bit and 32-bit encodings for most languages
- Disadvantages:
  - It's a variable-width encoding
  - Text for a few languages requires more storage than 16-bit encodings

# UTF-16

- UCS-2 is obsolete since it can't encode all code points
  - But many large systems and libraries were built using the 16-bit UCS-2 encoding
  - UTF-16 supersedes UCS-2 and provides a 16-bit Unicode encoding
- For code points in [U+0000, U+FFFF], UTF-16 is equivalent to UCS-2
  - Valid UCS-2 text is also valid UTF-16 text
  - These are the most commonly used code points, called the Basic Multilingual Plane (BMP)
- 2,048 code points in [U+D800, U+DFFF] are reserved for UTF-16
  - These are used to encode code points outside of the Basic Multilingual Plane
  - Code points in [U+10000, U+10FFFF] encoded using surrogate pairs

# UTF-16

How do we encode code points in [U+10000, U+10FFFF]?

1. Subtract 0x010000 from the code point; this yields a 20 bit number in [0x0, 0xFFFFF]
2. Split the number in half, with the upper ten bits in one half; the lower ten bits in the second
3. The upper ten bits are added to 0xD800 to form the lead surrogate
4. The lower ten bits are added to 0xDC00 to form the trail surrogate

# UTF-16

```
   0x1F378
-  0x10000
-----------
   0x0F378  → 0b00001111001101111000
```

```
0b00001111001101111000
      ↓           ↓
   0x003C      0x0378
+  0xD800      0xDC00
-------------------------
   0xD83C      0xDF78
```

# UTF-16

- Advantages:
  - Some level of compatibility with UCS-2 and systems designed to use a 16-bit encoding
  - Text for a few languages is smaller in UTF-16 than in UTF-8

- Disadvantages:
  - Multiple possible byte orderings, so byte order mark (BOM) is required for interchange
  - Every character requires at least two bytes
  - None of the byte-oriented string functions (like strcpy) work with UTF-16 strings
  - It's a variable-width encoding, but it's often misused as a fixed-width encoding
  - Binary string comparison for UTF-16 produces different results than for UTF-8 and UTF-32

# UTF-8, UTF-16, UTF-32

- UTF-8
  - More compact than UTF-32; usually more compact than UTF-16
  - It's a variable width encoding, so it's more complex
  - By far the most commonly used for storage and data transmission
  - Dominant character encoding on the Internet
- UTF-32
  - Simple, fixed width encoding
  - Lots of wasted space because of the large 32-bit code units

# UTF-8 vs. UTF-16 Text Size

```
pnb     1443     1204 119.85 % // Panjabi, Western
kor    67522    55332 122.03 % // Korean
mal    76590    60468 126.66 % // Malayalam
tel     2642     2080 127.02 % // Telugu
hin   455546   349438 130.37 % // Hindi
mar  1332509  1008604 132.11 % // Marathi
npi      460      348 132.18 % // Nepali
ben    27733    20802 133.32 % // Bengali
san     1465     1094 133.91 % // Sanskrit
kat    29970    22344 134.13 % // Georgian
ain      546      384 142.19 % // Ainu (Japan)
lao     1576     1084 145.39 % // Lao
khm    22676    15348 147.75 % // Central Khmer
cmn  1563663  1055924 148.08 % // Mandarin Chinese
yue   115488    77874 148.30 % // Yue Chinese
wuu   150945   101588 148.59 % // Wu Chinese
tha    18745    12610 148.65 % // Thai
lzh    73285    48982 149.62 % // Literary Chinese
bod     3925     2622 149.69 % // Tibetan
jpn  9778643  6524720 149.87 % // Japanese
```

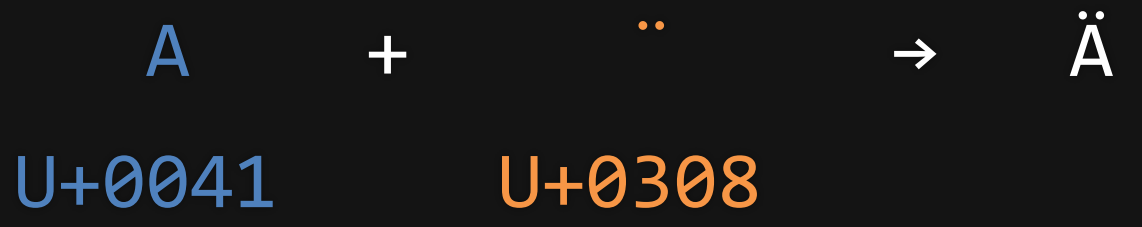
# Dynamic Composition

# Dynamic Composition

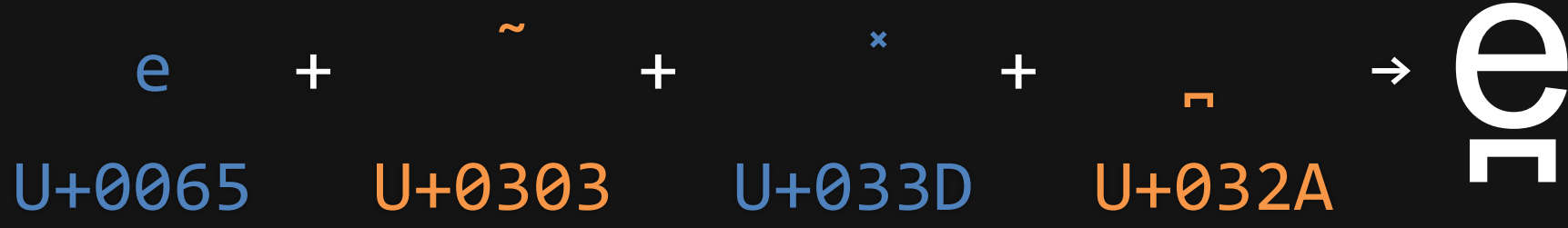
A

ÀÁÂÃÄÅ

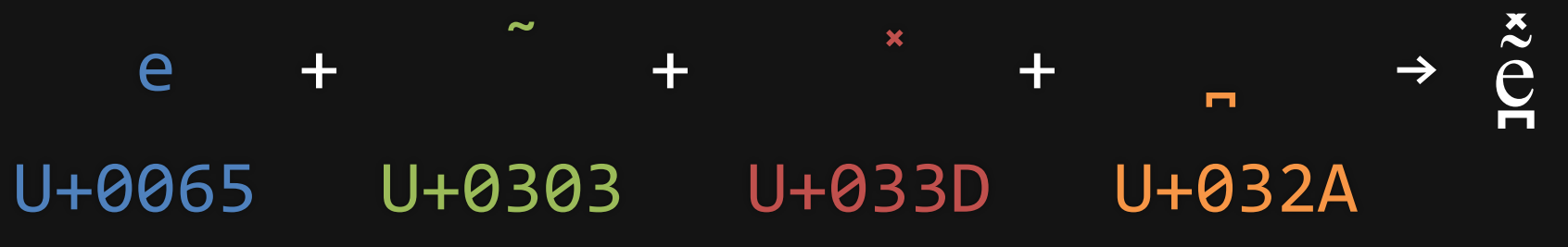
# Dynamic Composition

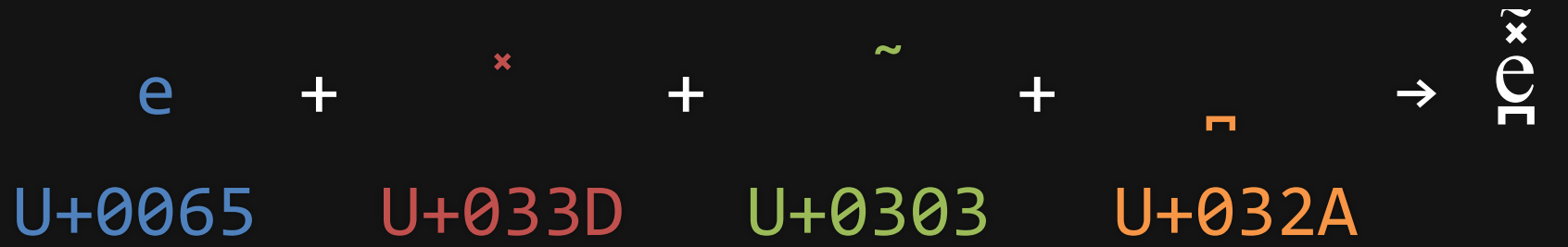


# Dynamic Composition



# Dynamic Composition



2 x U+25D4 (CIRCLE WITH UPPER RIGHT QUADRANT BLACK); ROLLING EYES

# Unicode Types in C++

# Unicode Code Unit Types

| | Code Unit Type |
|---|---|
| UTF-8 | `char` |
| UTF-16 | `char16_t` |
| UTF-32 | `char32_t` |

# Unicode String Literals

```
char const a[]{u8"Hello, \u2603!"};
char const b[]{u8"Hello, ☃!"};
// sizeof(a) == sizeof(b) == 12

char16_t const c[]{u"Hello, \u2603!"};
char16_t const d[]{u"Hello, ☃!"};
// sizeof(c) == sizeof(d) == 20

char32_t const e[]{U"Hello, \u2603!"};
char32_t const f[]{U"Hello, ☃!"};
// sizeof(e) == sizeof(f) == 40
```

# Unicode Character Literals (Sort Of)

```
char a{'x'};
char b{'☃'}; // Wrong
char c{'🍸'}; // Wrong

char16_t d{u'x'};
char16_t e{u'☃'};
char16_t f{u'🍸'}; // Wrong

char32_t g{U'x'};
char32_t h{U'☃'};
char32_t i{U'🍸'};
```

```
char a{'\u0078'};
char b{'\u2603'};      // Wrong
char c{'\U0001F378'}; // Wrong

char16_t d{u'\u0078'};
char16_t e{u'\u2603'};
char16_t f{u'\U0001F378'}; // Wrong

char32_t g{U'\U00000078'};
char32_t h{U'\U00002603'};
char32_t i{U'\U0001F378'};
```

# The Amazing C++ Unicode String Type

- [This slide intentionally left blank.]

# std::basic_string specialization typedefs

| | Code Unit Type | String Type |
|---|---|---|
| UTF-8 | `char` | `std::string` |
| UTF-16 | `char16_t` | `std::u16string` |
| UTF-32 | `char32_t` | `std::u32string` |

# std::basic_string

```
std::string a{u8"Hello, \u2603!"};
std::string b{u8"Hello, ☃!"};

std::u16string c[]{u"Hello, \u2603!"};
std::u16string d[]{u"Hello, ☃!"};

std::u32string e[]{U"Hello, \u2603!"};
std::u32string f[]{U"Hello, ☃!"};
```

# std::basic_string

- You must remember that it's just a dumb sequence of code units
  - NOT code points, characters, text elements, grapheme clusters, etc.
- Iterators (using begin() and end()) iterate over code units
- size() does not return the number of code points; it returns the number of code units
- Functions like front, back, operator[], push_back, find(CharT) overloads operate on code units

# std::basic_string

```
std::string a{"Hello, "};

// Wrong:
a.push_back('☃'); // U+2603

// Right:
char const snowman[]{u8"☃"}; // U+2603
a.insert(a.end(), begin(snowman), end(snowman));
```

# std::basic_string

```
std::u16string a{u"A glass: "};

// Wrong:
a.push_back(u'🍸'); // U+1F378

// Right:
char16_t const glass[]{u"🍸"}; // U+1F378
a.insert(a.end(), begin(glass), end(glass));
```

# std::basic_string

```
std::u32string a{U"A glass: "};

// Right:
a.push_back(U'🍸'); // U+1F378
```

# “Length”

U+1F4CF (STRAIGHT RULER)

# String Length

"Hello"

↑↑↑↑↑

12345

# String Length

"1 Ä 🍸"

| | 1 | A | ¨ | 🍸 |
|---|---|---|---|---|
| | U+0031 | U+0041 | U+0308 | U+1F378 |
| UTF-8 | 31 | 41 | CC 88 | F0 9F 8D B8 |
| UTF-16 | 0031 | 0041 | 0308 | D83C DF78 |
| UTF-32 | 00000031 | 00000041 | 00000308 | 0001F378 |

# String Length: Number of Bytes

"1 Ä 🍸"

| 1 | A | ¨ | 🍸 |
|---|---|---|---|
| U+0031 | U+0041 | U+0308 | U+1F378 |

| | | | | | Length |
|---|---|---|---|---|---|
| UTF-8 | 31 | 41 | CC 88 | F0 9F 8D B8 | 8 |
| UTF-16 | 0031 | 0041 | 0308 | D83C DF78 | 10 |
| UTF-32 | 00000031 | 00000041 | 00000308 | 0001F378 | 16 |

# String Length: Number of Code Units

"1 Ä 🍸"

| | 1 | A | ¨ | 🍸 |
|---|---|---|---|---|
| | U+0031 | U+0041 | U+0308 | U+1F378 |

| | | | | | Length |
|---|---|---|---|---|---|
| UTF-8 | 31 | 41 | CC 88 | F0 9F 8D B8 | 8 |
| UTF-16 | 0031 | 0041 | 0308 | D83C DF78 | 5 |
| UTF-32 | 00000031 | 00000041 | 00000308 | 0001F378 | 4 |

# String Length: Number of Code Points

"1 Ä 🍸"

| 1 | A | ¨ | 🍸 |
|---|---|---|---|
| U+0031 | U+0041 | U+0308 | U+1F378 |

| | | | | | Length |
|---|---|---|---|---|---|
| UTF-8 | 31 | 41 | CC 88 | F0 9F 8D B8 | 4 |
| UTF-16 | 0031 | 0041 | 0308 | D83C DF78 | 4 |
| UTF-32 | 00000031 | 00000041 | 00000308 | 0001F378 | 4 |

# String Length: Number of Text Elements

"1 Ä 🍸"

| 1 | A | ¨ | 🍸 |
|---|---|---|---|
| U+0031 | U+0041 | U+0308 | U+1F378 |

| | | | | | Length |
|---|---|---|---|---|---|
| UTF-8 | 31 | 41 | CC 88 | F0 9F 8D B8 | 3 |
| UTF-16 | 0031 | 0041 | 0308 | D83C DF78 | 3 |
| UTF-32 | 00000031 | 00000041 | 00000308 | 0001F378 | 3 |

# String Length

- At least four possible meanings of “string length”
  - Number of bytes
  - Number of code units
  - Number of code points
  - Number of text elements
- The # of code points and the # of text elements is the same regardless of encoding
- For UTF-8, the # of bytes is the same as the # of code units
- For UTF-32, the # of code units is the same as the # of code points

# String Length

```
std::string s{u8"1ÄŸ"};

std::size_t number_of_code_units{s.size()};

for (auto&& c : s)
{
    // Iterating over code units, not characters
}
```

# “Equality”

U+2248 (ALMOST EQUAL TO)

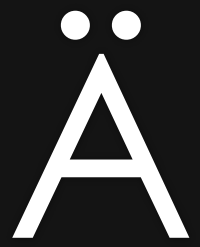
# Representational Equality

```
std::string a{"Hello, Aspen!"};
std::string b{"Hello, Aspen!"};

if (a == b)
{
    std::cout << "The strings are equal.\n";
}
```

# Representational Equality

```
std::string a{u8"1ÄẎ"};
std::string b{u8"1ÄẎ"};

if (a == b)
{
    std::cout << "The strings are equal.\n";
}
```

# Multiple Representations



Latin Capital Letter A; Combining Diaeresis

U+0041 U+0308

A + ¨

Ä

U+00C4

Latin Capital Letter A With Diaeresis

# Multiple Representations

A

Latin Capital Letter A

U+0041

A

A

U+FF21

Fullwidth Latin Capital Letter A

# Multiple Representations

fi

Latin Small Letter F; Latin Small Letter I

U+0066 U+0069

f + i

fi

U+FB01

Latin Small Ligature Fi

# Multiple Representations

Latin Capital Letter X; Latin Capital Letter I (x2)

U+0058 U+0049 U+0049

X + I + I

Ⅻ

U+216B

Roman Numeral Twelve

# Multiple Representations

Latin Capital Letter A

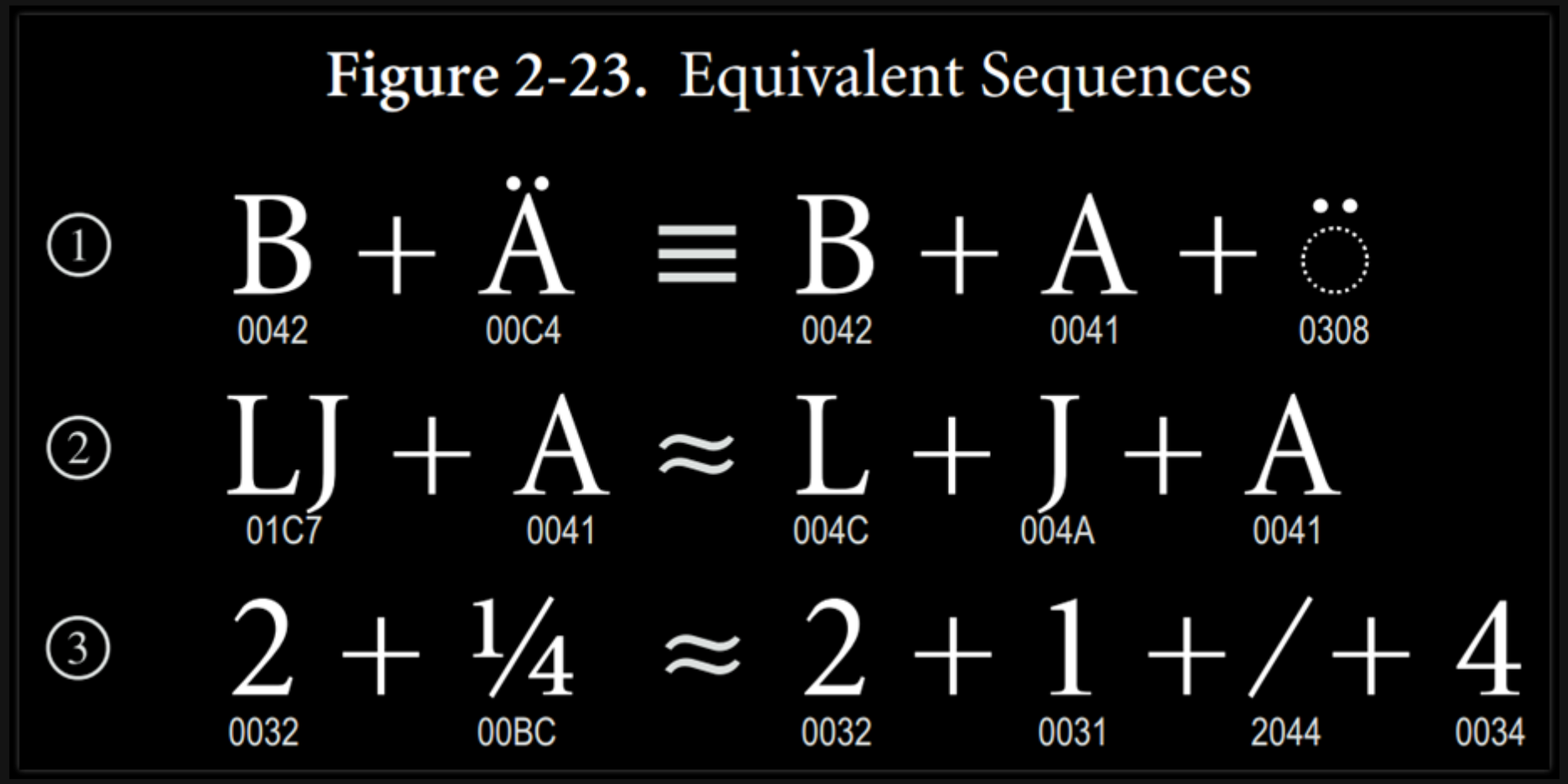
Combining Equals Sign Below

Combining Circumflex Accent

U+0041 U+0347 U+0302

U+0041 U+0302 U+0347

# Equivalent Sequences




Unicode 6.3 §2.12

# Normalization

- Four Normalization Forms
  - **NFC:** Canonical Composed
  - **NFD:** Canonical Decomposed
  - **NFKC:** Compatibility Composed
  - **NFKD:** Compatibility Decomposed
- Normalization includes a well-defined ordering for combining marks

# Normalization Using the Standard Library

- [This slide intentionally left blank.]

# Ordering

U+003C (LESS-THAN SIGN)

# Ordering

```
std::string a{"a"};
std::string b{"b"};

if (a < b)
{
    std::cout << "a is ordered before b.\n";
}
```

# Ordering

```
std::string bear {u8"🐻"};
std::string whale{u8"🐋"};

if (bear < whale)
{
    std::cout << "the bear is ordered before the whale.\n";
}
```

# Ordering

```
std::vector<std::string> v{"c", "X", "b", "Y", "a", "Z"};

// Alphabetize the strings...
std::sort(v.begin(), v.end());

for (auto&& s : v) { std::cout << s << ' '; }
```

# Ordering

```
std::vector<std::string> v{"c", "X", "b", "Y", "a", "Z"};

// Alphabetize the strings...
std::sort(v.begin(), v.end());

for (auto&& s : v) { std::cout << s << ' '; }

// Output: X Y Z a b c
```

# String Collation

```
std::vector<std::string> v{"c", "X", "b", "Y", "a", "Z"};

std::locale en_us{"en-US"};
std::sort(v.begin(), v.end(), en_us);

for (auto&& s : v) { std::cout << s << ' '; }

// Output: a b c X Y Z
```

# String Collation

```
std::vector<std::string> v{"z", "ä", "b", "a"};

std::locale en_us{"de-DE"};
std::sort(v.begin(), v.end(), en_us);

for (auto&& s : v) { std::cout << s << ' '; }

// Output: a ä b z
```

# String Collation

```
std::vector<std::string> v{"z", "ä", "b", "a"};

std::locale en_us{"sv-SE"};
std::sort(v.begin(), v.end(), en_us);

for (auto&& s : v) { std::cout << s << ' '; }

// Output: a b z ä
```

# Unicode Collation using the Standard Library

- [This slide intentionally left blank.]

# Other Text Operations

# Text Manipulation

```
std::locale loc{"en-US"};

char lowercase_a{'a'};
char uppercase_a{std::toupper(lowercase_a, loc)}; // A
```

# Text Manipulation

- Most Standard Library manipulation functions (like toupper) are code unit based
- But Unicode text manipulation does not work well with this model
- Consider ß in German:  the uppercase form is SS, which is two characters

# Text Manipulation

Even the classification functions are unusable…

```
template <typename CharT> bool isspace(CharT c, locale const& loc);
template <typename CharT> bool isalpha(CharT c, locale const& loc);
template <typename CharT> bool isalnum(CharT c, locale const& loc);
```

…except possibly for UTF-32 (char32_t).

# Encoding Conversions using <codecvt>

```
std::string utf8{u8"1ÄŸ"};

std::wstring_convert<
    std::codecvt_utf8<char32_t>, char32_t
> utf32_converter;

std::u32string utf32{utf32_converter.from_bytes(utf8)};
```

# I/O Conversions using <codecvt>

```
// Write UTF-8 data with BOM:
std::ofstream("text.txt")
    <<"\xef\xbb\xbf\x31\x41\xcc\x88\xf0\x9f\x8d\xb8";

// Read the UTF8 file, skipping BOM:
std::basic_ifstream<char32_t> f("text.txt");
f.imbue(std::locale(f.getloc(),
    new std::codecvt_utf8<char32_t, 0x10ffff, std::consume_header>));

for (char32_t c{}; f.get(c); ) { std::cout << std::hex << c << ' '; }
// Output:  31 41 308 1f378
```

# International Components for Unicode (ICU)

# International Components for Unicode

- Huge set of features—probably everything you'd ever need
- Very widely-used, so it's well-tested in a lot of real-world software
- Runs on many platforms; has permissive license

- It has its own string type, UnicodeString, which uses UTF-16
- It doesn't work well with other string types (like std::string) in most contexts
- It's not "modern C++" by any definition of that term
- Originally written for Java; ported to C++

# UChar32 and UChar

```
// UTF-32 code units
UChar32 rook_utf32{static_cast<UChar32>(U'♖')}; // U+2656
UChar32 cake_utf32{static_cast<UChar32>(U'🍰')}; // U+1F370

// UTF-16 code units
UChar rook_utf16{u'♖'};                  // U+2656
UChar cake_utf16[2]{0xD83C, 0xDF70}; // U+1F370

// UTF-8 code units
char rook_utf8[4]{u8"♖"}; // U+2656
char cake_utf8[5]{u8"🍰"}; // U+1F370
```

# UnicodeString

```
UChar32 cake_utf32c{static_cast<UChar32>(U'🍰')};
UChar32 cake_utf32s[2]{static_cast<UChar32>(U'🍰'), 0};
UChar   cake_utf16 [3]{0xD83C, 0xDF70, 0};
char    cake_utf8  [5]{u8"🍰"};

UnicodeString cake_1{cake_utf32c};
UnicodeString cake_2{UnicodeString::fromUTF32(cake_utf32s, 1)};
UnicodeString cake_2{cake_utf16};
UnicodeString cake_3{UnicodeString::fromUTF8(cake_utf8)};
```

# UnicodeString

- compare, compareCodePointOrder, caseCompare
- length, countChar32
- startsWith, endsWith, indexOf, lastIndexOf
- append, insert, replace, findAndReplace, reverse
- padLeading, pad, padTrailing
- toUpper, toLower, toTitle
- toUTF8String, toUTF32

# Normalization

```
UErrorCode status{U_ZERO_ERROR};
Normalizer2 const& normalizer{*icu::Normalizer2::getInstance(
    nullptr, "nfc", UNORM2_COMPOSE, status)};

// U+0041 U+0308 (Latin Capital Letter A, Combining Diaeresis)
UnicodeString source{UnicodeString::fromUTF8(u8"\u0041\u0308")};
bool is_normalized{normalizer.isNormalized(source, status)};

UnicodeString result{};
normalizer.normalize(source, result, status);
// U+00C4 (Latin Capital Letter A with Diaeresis)
```

# Collation

```
UErrorCode status{U_ZERO_ERROR};
std::unique_ptr<Collator> collator{
    Collator::createInstance(Locale("en", "US"), status)};

UnicodeString a{UnicodeString::fromUTF8(u8"a")};
UnicodeString z{UnicodeString::fromUTF8(u8"Z")};

UCollationResult result{collator->compare(a, z, status)};
```

# Regular Expressions

```
UErrorCode status{U_ZERO_ERROR};
std::unique_ptr<RegexMatcher> matcher{new RegexMatcher{
    UnicodeString::fromUTF8(u8R"(\p{Number})"), 0, status}};

matcher->reset(UnicodeString::fromUTF8(u8"Ⅻ⅝"));
while (matcher->find())
{
    // Handle match
}
```

# Boost.Locale and Boost.Regex

# Boost.Locale

- Introduced in Boost 1.48
- A modern C++ API that can use various backends (with varying levels of feature support)
  - ICU
  - std (C++ Standard Library, plus workarounds for common bugs)
  - posix
  - winapi
- Not nearly as feature-filled as ICU, but supports most commonly used features

# Boost.Locale

```
boost::locale::generator gen{};

std::locale loc{gen.generate("en-US.UTF-8")};
```

# Normalization

```
// Latin Capital Letter A With Diaeresis
std::string a{u8"\u00C4"};
std::string b{u8"A\u0308"};

std::string a_decomp{boost::locale::normalize(a, norm_nfd, loc)};
std::string b_comp  {boost::locale::normalize(b, norm_nfc, loc)};

assert(a_decomp == b);
assert(b_comp   == a);
```

# Collation

```
std::string a{u8"A"};
std::string accented_a{u8"A\u0308"};

int result{std::use_facet<boost::locale::collator<char>>(loc)
    .compare(boost::locale::collator_base::primary, a, accented_a)};

typedef std::set<std::string, boost::locale::comparator<char>> set_t;
set_t s{boost::locale::comparator<char>{
    loc, boost::locale::collator_base::primary}};
```

# Collation Comparison Levels

Supports all five Unicode comparison levels:

- L1 (Primary): Ignores case and accents; compares base characters only
- L2 (Secondary): Ignores case but considers accents
- L3 (Tertiary): Considers both case and accents
- L4 (Quaternary): Considers case, accents, and punctuation
- Ln (Identical): Considers case, accents, and punctuation; requires identical code points

# Conversions

```
std::string    a{u8"1ÄŸ"};
std::u16string b{boost::locale::conv::utf_to_utf<char16_t>(a)};
std::u32string c{boost::locale::conv::utf_to_utf<char32_t>(a)};
std::u32string d{boost::locale::conv::utf_to_utf<char32_t>(b)};
std::string    e{boost::locale::conv::utf_to_utf<char    >(b)};
```

# Conversions

```
//                  А   л   л   о
std::string s{"\xB0\xDB\xDB\xDE"};
std::string utf8_s{boost::locale::conv::to_utf<char>(s, "ISO-8859-5")};
```

# Text Manipulation

- Provides string-based case manipulation (to_upper, to_lower, to_title, fold_case)

# Boundary Analysis

```
namespace ba = boost::locale::boundary;

std::string subject{u8"1ÄŸ"};

ba::segment_index<std::string::const_iterator> map(
    ba::character, subject.begin(), subject.end(), loc);

size_t const byte_length{subject.size()};                    // 8
size_t const text_length{std::distance(map.begin(), map.end())}; // 3
```

# Boost.Regex

```
boost::u32regex r{boost::make_u32regex(u8R"(\p{Number})")};

std::string subject{u8"XII⅝"};

typedef boost::u32regex_token_iterator<...> iterator_type;
iterator_type first{boost::make_u32regex_token_iterator(subject, r)};
iterator_type last{};
for (auto it(first); it != last; ++it)
{
    // Process matches
}
```

# Proposals for Standardization

# N3336: Adapting Standard Library Strings and I/O to a Unicode World

- www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3336.html

- Retrofit some new features onto existing Standard Library stuff
- New overloads of basic_string constructors, operator=, etc., to do encoding conversions
- New overloads of formatted I/O operators for I/O streams to handle different string types
- New encoding conversion iterators

# N3572: Unicode Support in the Standard Library

- www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3572.html

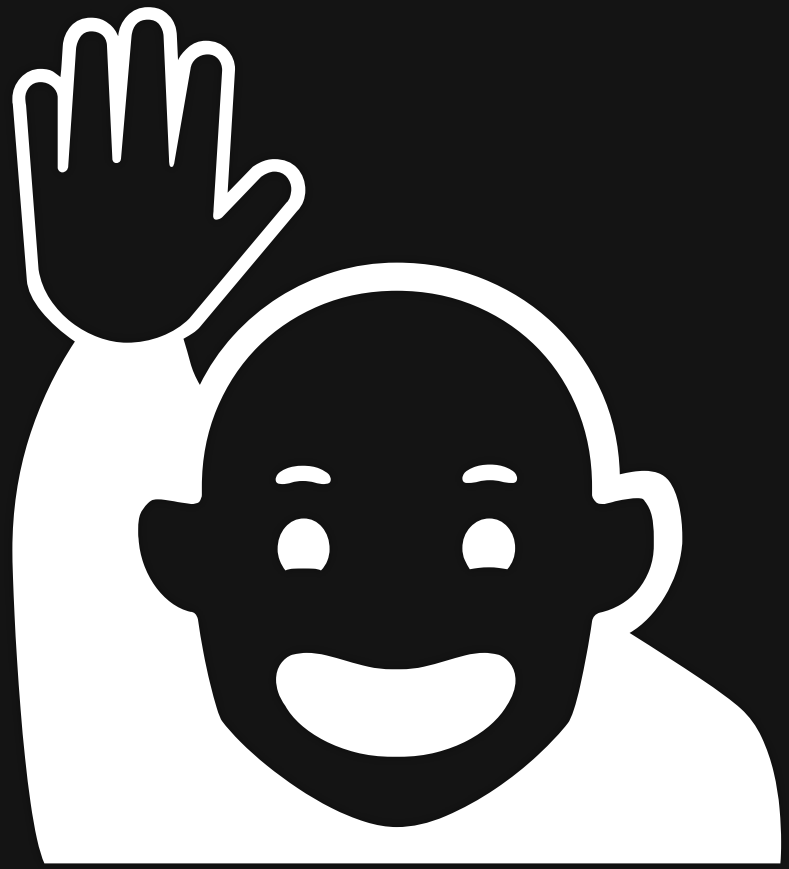
- Proposes new Unicode-aware functionality for the Standard Library
- A new encoded_string type that supports various Unicode and other encodings
  - Provides a slimmed-down std::string-like interface
  - All operations are on code points, not code units
  - Including iteration, size(), push_back(), etc.
  - Iterators are bidirectional only
  - Operators to support mixing/matching different encoded_string specializations
  - Hashing and comparison produce correct results for canonically equivalent strings

# Resources

U+1F4D6 (OPEN BOOK)

- **Unicode Explained**, by Jukka K. Korpela: http://shop.oreilly.com/product/9780596101213.do
- **The Unicode Standard:** http://www.unicode.org/standard/standard.html
- **International Components for Unicode (ICU):** http://site.icu-project.org/
- **Boost.Locale**: http://www.boost.org/doc/libs/1_55_0/libs/locale/doc/html/index.html
- **Boost.Nowide:** http://cppcms.com/files/nowide/html/ (not part of Boost…yet?)
- **http://www.utf8everywhere.org/**
- **"Should UTF-16 be considered harmful?"** http://programmers.stackexchange.com/q/102205/206

# Questions?



U+1F64B (HAPPY PERSON RAISING ONE HAND)

# Questions?

\0

U+0000 (HAPPY PERSON RAISING ONE HAND?)

# Unicode in C

# Unicode Support in C90

- [This slide intentionally left blank.]

# Unicode Support in C90

- wchar_t
- mbtowc, wctomb
- mbstowcs, wcstombs

# Unicode Support in C99

- wchar_t
- mbtowc, wctomb
- mbstowcs, wcstombs
- wchar_t-equivalents for many I/O and string handling function

# Unicode Support in C11

- New code unit types:  char16_t and char32_t
  - Actual encoding is implementation-defined
  - __STDC_UTF_16__ and __STDC_UTF_32__
- New string literal prefixes:  u"Hello!" and U"Hello!"
- <uchar.h>:  mbrtoc16, c16rtomb, mbrtoc32, c32rtomb

# mbrtoc32

```
setlocale(LC_CTYPE, "en_US.UTF-8");

char const utf8_c[5] = "\U0001f378";

char32_t  utf32_c = 0;
mbstate_t state   = { 0 };
mbrtoc32(&utf32_c, utf8_c, 4, &state);

printf("0x%8x\n", utf32_c);
```

# mbrtoc32

```
setlocale(LC_CTYPE, "en_US.UTF-8");

char const utf8_c[5] = "\U0001f378";

char16_t   utf16_c[2] = { 0 };
mbstate_t  state      = { 0 };
mbrtoc16(&utf16_c[0], utf8_c, 4, &state);
mbrtoc16(&utf16_c[1], utf8_c, 4, &state);

printf("0x%4x 0x%4x\n", utf16_c[0], utf16_c[1]);
```