Tajemnice libstdc++

Disclaimer

Artykuł oczywiście dotyczy stricte C++. Moja wersja g++ w chwili pisania kartki to 5.3.0. Przed wyciąganiem wniosków warto sprawdzić własną wersję kompilatora lub ustalenia techniczne sprawdzarki. W każdym razie, do swojej wersji będę się tutaj odnosił. Będę też zakładał że korzystamy ze standardu C++11. Dodatkowo będę wycinał komentarze z wklejanego kodu źródłowego.
Ogólnie cała ta kartka jest pełna technikaliów. Casuals beware.

libstdc++?

Każda wersja `g++` ma w pakiecie implementację biblioteki C++, czyli STL. Jednak to nie ten sam zespół jest odpowiedzialny za tę implementację, co za kompilator - owa jest oddzielnym projektem o nazwie `libstdc++`. Interesują nas pliki źródłowe, których trzeba poszukać w swojej instalacji. W moim MinGW (które pewnie będzie trochę zbliżone do innych instalacji `g++`) są w `/lib/gcc/mingw32/5.3.0/include/c++`. Względem takiej ścieżki, w której znajdują się wszystkie pliki znacznikowe (`algorithm` i tym podobne), będę się tutaj odnosił.

Wprowadzenie

Przede wszystkim, po chwili poszukiwań, znajdziemy tu implementację każdej rzeczy z STLa, taką jaką obiecuje dokumentacja. Należy wziąć pod uwagę, że kod, który ujrzymy, będzie bardzo mocno opatrzony szablonami, makrami związanymi z implementacją kompilatora, makrami od debugu, etc., przez co może być trochę nieczytelny. Tak na przykład wygląda implementacja `std::reverse` (`algorithm` odeśle nas do `bits/stl_algo.h` oraz `bits/stl_algobase.h`, przy czym szukana implementacja jest w tym pierwszym):

▸ `std::reverse`

Widać tutaj to, o czym mówiłem. Szablon – ✓. Makra – ✓. Debug: – ✓. No i do tego odwołanie do `__reverse`, aby ukryć implementację (podwójny `_` często oznacza, że coś jest używane przez wewnętrzną implementację i lepiej tego nie psuć, podobna konwencja dotyczy pojedynczego). Patrzymy na tę funkcję i widzimy:

▸ `__reverse`

Cóż, to prawie to czego szukamy. Pozostaje rozszyfrować czym jest `std::iter_swap`. Ja nie wiedziałem. Cóż, C++ nie przestaje zaskakiwać. Tym razem znajdziemy go w `bits/stl_algobase.h`

▸ `std::iter_swap`

Pozwoliłem sobie oszczędzić pokazywania 50 linii boilerplate'u, który faktycznie nazywa się `std::iter_swap` (A wy napisaliście kiedyś odwracanie `vector`a w 150 linii?). Chodzi o prosty wniosek: znalezienie czegoś tutaj może być... skomplikowane. Można jednak znaleźć kwiatki.

Rozszerzenia GNU

Owe rozszerzenia znajdują się głównie w `ext`, i głównie o będę opowiadał. Większość ze znajdujących się tam rzeczy jest w przestrzeni nazw `__gnu_cxx`. Czeka na nas tutaj trochę ciekawych rzeczy, o ile jesteśmy dostatecznie uparci.

Szybkie potęgowanie

W pliku `ext/numeric` znajdziemy `__gnu_cxx::power` - implementację szybkiego potęgowania działającą w $O(\log n)$. Wywołanie jest proste: `power(a, b)` zwróci `a` podniesione do potęgi `b`. Dodatkowo można przekazać trzeci parametr będący obiektem który można wywołać (tj. ze zdefiniowanym `operator()`). Ten obiekt (oznaczmy go `F`) musi też mieć zdefiniowany wynik funkcji `identity_element(F)`, która posłuży za wynik dla `power(a, 0)`. Wtedy dokładniejsza definicja `power` to:

▾ Definicja `power`

```
def power(a, b, F):
    if b == 0:
        return identity_element(F)
    else:
        return F(a, power(a, b-1, F))
```

Oczywiście `power` ma sens dla całkowitych i nieujemnych `b`.

Często będziemy chcieli liczyć potęgi modulo pewna liczba, i wtedy będziemy musieli napisać tzw. *functor*, czyli prosty typ, który definiuje `operator()`. Dużo przykładów takich prostych functorów znajdziemy w `<functional>`, domyślnym w tym wypadku jest `std::multiplies<T>`. Inne przykłady: `std::plus<T>, std::modulus<T>`.

Taki functor (wraz z konstruktorem do dowolnego modulo, warto jednak modulować przez stałą, patrz: *O modulo*) oraz przykład wykorzystania w `__gnu_cxx::power`, może wyglądać tak:

▾ Przykład użycia `__gnu_cxx::power`

```cpp
#include <iostream>
#include <ext/numeric>

using namespace __gnu_cxx;
using namespace std;

template<typename T>
struct mod_multiplies
{
    T mod;
    mod_multiplies(T _mod) : mod(_mod) {}
    T operator() (T a, T b) { return (a*b) % mod; }
};
template<typename T>
T identity_element(mod_multiplies<T>) { return 1; }

int main()
{
    int64_t a, m; uint64_t b;
    cin >> a >> b >> m;
    cout << power(a, b, mod_multiplies<int64_t>(m));
}
```

Jeżeli ktoś bardzo chce skrócić kod, można: 1) nie robić szablonu, 2) odziedziczyć od `multiplies`, aby wartość `identity_element` była zdefiniowana, 3) usunąć konstruktor i modulować przez stałą.

Ciekawostka: kompilator umie zoptymalizować proste, stałe wywołania jak `power(3, 5)` do stałej (`243`), albo `power(x, 3)` do dosłownego `x*x*x`. Dobrze mieć to na uwadze, zamiast pisać to samo wiele razy.

Według mnie opłaca się znać – mniej miejsca na błąd w implementacji, a functor napisać bardzo łatwo jak się wyuczy. Jeżeli zdefniujemy dla naszego typu `operator*` nie trzeba będzie pisać dodatkowego functora.

Można w ten sposób potęgować macierze (trzeba tylko napisać odpowiedni `identity_element` – macierz tożsamościową) i inne operacje, które sensownie wpasowują się w definicję.

Policy-Based Data Structures

▸ *Notatka o notacji O*

Jeden z najbardziej rozbudowanych i najmniej udokumentowanych bytów jakie spotkałem, policy-based data structures są bardzo ciekawą rzeczą która może nas uratować od utopienia się w implementacji jakiejś struktury, takiej jak BST (*Binary Search Tree [binarne drzewo poszukiwań]* - nawiasem mówiąc temat ciekawy, który zasługuje na własną kartkę) czy hashmapa. Bez wątpienia - warto znać. Czemu policy-based? Ponieważ ich działanie możemy modyfikować wybierając `tag` (czyli jakiej dokładnie struktury danych użyć aby zaimplementować dany szablon) oraz inne fragmenty decydujące o działaniu struktury. Wszystkie są w przestrzeni nazw `__gnu_pbds`.

Dzielą się na:

- Associative containers – struktura, która przyporządkowuje *klucze* pewnego typu do *wartości* innego typu - przykładem jest `std::map`, a `std::set` jest wariacją która nie trzyma żadnych wartości. (Analogicznie odmiany `unordered` - czyli hashmapy). Znajdują się w pliku `ext/pb_ds/assoc_container.hpp`. Wśrod nich wyróżniamy:
  - Tree-based – struktury oparte na drzewach (binarnych). Pozwalają na pisanie specjalnych klas *`node_update`* pozwalających na czytanie i zapisywanie dodatkowych informacji w strukturze BST (więcej o tym później). Implementują metody `split` i `join`, działające na podstawie struktury drzewa (Uwaga: `split` działałby w czasie polilogarytmicznym, gdyby nie wredne wywołanie `std::distance` do liczenia wielkości powstałych drzew. Aby to zoptymalizować należy napisać analogiczny update do order statistics (aby zliczać wielkości drzew) oraz overloadować `std::distance` dla iteratorów w używanym typie drzewa) (Uwaga 2: `join` działa tylko jeżeli drzewa obejmują rozłączne zakresy elementów, tzn. maksymalny element jednego z drzew jest mniejszy od minimalnego elementu drugiego drzewa). Do wyboru tagi:
    - **`rb_tree_tag`** – Red-black tree [Drzewo czerwono-czarne], rodzaj samobalansującego BST i prawdopodobnie najlepszy dobór do naszych celów. Zapewnia czas $\Theta(\log n)$ na operacje.
    - `splay_tree_tag` – Splay tree [Drzewo splay], drzewo opierające się na przenoszeniu poszukiwanych elementów w pobliże korzenia (operacja `splay`), tym samym gwarantuje jedynie że zamortyzowana złożoność czasowa operacji to $O(\log n)$ (jeżeli drzewo nie zbalansuje się poprawnie może być to czas gorszy).
    - `ov_tree_tag` - Ordered vector tree. Posortowany wektor, który udaje BST. Wszelkie modyfikacje są $O(n)$, ale same wyszukiwania powinny być szybsze, ponieważ są tak na zwykłej ciągłej tablicy. Wyszukiwania te będą szybsze niż `std::lower_bound` i `std::upper_bound` z STLa, co wynika z implementacji (zamiast wyszukiwania przez skoki o potęgi dwójki, czyli wykładniczego, zwykłe patrzenie na środek zakresu).

    ▸ `tree`
  - Trie-based – na drzewach trie, do przechowywania słownika ciągów znaków. Jedyny tag to `pat_trie_tag` (Patricia Trie [skompresowane drzewo trie]). Także mają `node_update` - wbudowane `trie_order_statistics_node_update` (operacje na indeksach w porządku leksykograficznym: `find_by_order`, `order_of_key`, `order_of_prefix`) oraz `trie_prefix_search_node_update` (wyszukiwanie kluczy o danym prefiksie – `prefix_range`). Wszystkie operacje są w rozsądnej złożoności czasowej, która opiera się na długościach stringów którymi manipulujemy, czy ilości elementów, które

mają prefiks wspólny z pewnym ciągiem.
▸ `trie`

- Hash-based – Oczywiście hashmapy. Dostępne różnią się sposobem rozwiązywania kolizji hashy. W przypadku losowym są kilka razy szybsze od `std::unordered_map`, przy sekwencyjnych operacjach jedynie `gp_hash_table` jest do 2 razy szybsze. Tym razem nie używa się tagów, a są to oddzielne typy:
  - `cc_hash_table` – *Collision chaining*. Przy kolizji budujemy listę jednokierunkową zbierającą wszystkie klucze o tym samym hashu. Do wyboru rodzaj funkcji hashującej i sposób doboru wielkości tabeli.
    ▸ Deklaracja `cc_hash_table`
  - **`gp_hash_table`** – *General probing*. Tutaj w razie kolizji (w uproszczeniu) odwiedzamy kolejne komórki dopóki nie napotkamy pustej, którą właśnie zajmujemy. Oczywiście sprawia się gorzej na gęściej zaludnionych hashmapach, ale w ogólnym przypadku są efektywniejsze od `cc_hash_table`. Tutaj wybieramy rodzaj funkcji hashującej, sposób doboru wielkości oraz probingu. Uwaga: możliwe, że domyślnym hashem będzie modulo przez potęgę 2. Jeżeli jest możliwość aby test składał się z liczb które łatwo psują taki hash, to należy napisać własny hash lub dobrać resize_policy.
    ▸ Deklaracja `gp_hash_table`

  Normalnie te hashmapy zachowują się w większości jak `std::unordered_map`. Aby nie przechowywały wartości, należy je zadeklarować jako `-_hash_table<T, null_type>` (`null_mapped_type` nie istnieje, mimo tego, że na niego kieruje dokumentacja), czyli z typem wartości jako `null_type`. Wtedy będą się zachowywać jak `std::unordered_set`. Miałem pewne problemy z poprawnym działaniem `find` i iteratorów, więc lepiej uważać. Nie ma też obsługi `reserve`.

- List-based – Bardzo prosta struktura oparta na liście kierunkowej. Dokumentacja opisuje ich zamysł jako wykorzystanie w tworzeniu multimap (w odróżnieniu od STLa PBDS ich domyślnie nie oferuje, mowa tu o `std::multimap/ std::multiset`). Są zaimplementowane jako *bardzo proste* listy, więc np. `size()` jest liniowe. Wybieramy tutaj update policy, czyli sposób dopasowywania najczęstszych zapytań do budowy listy (częściej odwiedzane idą na przód - domyślnie każdy odwiedzony idzie od razu na przód). Nie są one wyjątkowo użyteczne, więc zachęcam do przejrzenia dokumentacji w celu znalezienia szczegółów.
  ▸ `list_update`

- Priority queues – najzwyklejsze kolejki priorytetowe, ale plusem jest tutaj liczność tagów. Dodatkowo zaimplementowane są dodatkowe operacje - `modify`, `join`, `split`. Znajdziemy je w `ext/pb_ds/ priority_queue.hpp`. Dostępne tagi to:
  - `binary_heap_tag` – kopiec binarny, czyli najprostszy i 'najlepszy'

(wg. dokumentacji) do prymitywnych typów, takich jak `int`. Z drugiej strony ma słaby najgorszy przypadek złożoności, a `modify` i `erase` są $O(n)$.
- `pairing_heap_tag` – 'najlepszy' dla typów nieprymitywnych, np. `std::string`. Bardzo dobry najgorszy przypadek `push`, `join` $O(1)$, ale słaby dla `pop`.
- `binomial_heap_tag` – bardziej skomplikowany, więc `push` i `pop` mają gorszą wydajność, ale dzięki temu skomplikowaniu najgorszy przypadek jest lepszy.
- `rc_binomial_heap_tag` (binomial heap z 'redundant counter') – to samo co zwykły binomial, ale bardziej.
- `thin_heap_tag` – dobry najgorszy przypadek, zamortyzowana złożoność taka jak w kopcu Fibonacciego. Dokumentacja sugeruje, że ta odmiana jest odpowiednia do pewnych algorytmów grafowych.

Więcej o konkretnych kopcach można przeczytać w dokumentacji, gdzie są odnośniki do prac z opisami. Testy wydajnościowe i tabela złożoności: klik.

Pisanie zmodyfikowanych BST za pomocą **`tree`**

Czasami implementując nasze rozwiązanie możemy dojść do wniosku, że zapewniona przez STL implementacja BST (`std::map` albo `std::set`) może być niewystarczająca (Ot, taki przykład, zadanie "Rotacje na drzewie" z XVIII OI). Wtedy trzeba by samemu napisać takie BST. Dwa konkretne przykłady bardziej skomplikowanych drzew, które mogą nam być potrzebne:

- *Order statistics tree* – drzewo, które pozwala na zliczanie, ile elementów jest mniejszych od danej wartości, oraz na znajdowanie i-tego elementu w posortowanej kolejności.
- *Interval tree* – drzewo, które pozwala na operacje na zbiorze przedziałów. Na przykład można pytać o ilość przedziałów przecinających dany punkt czy inny przedział.

Oczywiście czasami takie bardziej skomplikowane BST można symulować zwykłym drzewem przedziałowym czy Fenwickiem. Jednakże należy wziąć pod uwagę, że takie rozwiązanie może być nieoptymalne albo skomplikowane w implementacji. (w przytoczonym zadaniu należy samemu napisać merge na dynamicznych drzewach przedziałowych)

Ale! Możemy tak zmodyfikować `tree`, aby zapewniało nam potrzebną funkcjonalność. Przedstawię, jak to zrobić na przykładzie order statistics tree.

Aby implementować nasze rozszerzenia wykorzystamy dostarczony przez `tree` interfejs `NodeUpdate`.

Order statistics tree

Przede wszystkim, *order statistics tree* jest już zaimplementowane za pomocą (jedynego) wbudowanego `NodeUpdate`, czyli `tree_order_statistics_node_update`. Przykład użycia:

```
1.  #include <bits/stdc++.h>
```

```
 2.  #include <ext/pb_ds/assoc_container.hpp>
 3.
 4.  using namespace std;
 5.  using namespace __gnu_pbds;
 6.
 7.  template<typename T>
 8.  using indexed_set = tree<
 9.      T, null_type, less<T>, rb_tree_tag, tree_order_stati
10.  >;
11.
12.  int main()
13.  {
14.      indexed_set<int> S;
15.      S.insert(1); S.insert(4); S.insert(7);
16.      cout << *S.find_by_order(1) << endl; // 4
17.      cout << S.order_of_key(0) << " " << S.order_of_key(4
18.  }
```

- `7:10.` – zwykła deklaracja `using` z szablonem. Gdybyśmy chcieli zamiast zbioru mieć mapę, można wpisać coś innego niż `null_type`. Jako implementację BST używamy drzewa czerwono-czarnego (`rb_tree_tag`), bo zwykle jest najlepszym wyborem. Ostatni parametr to `NodeUpdate`.
- `15.` – większość interfejsu `tree` jest taka sama, jak `std::set` czy `std::map`.
- `16.` – `iterator find_by_order(size_t index)` zwraca iterator do i-tego elementu z kolei.
- `17.` – `size_t order_of_key(T elem)` zwraca ilość elementów mniejszych od danego.

Jednak jak wziąć się za implementację własnej aktualizacji wierzchołków? Należy przeanalizować tę wbudowaną. Idea `NodeUpdate` opiera się na przechowywaniu metadanych w wierzchołkach, tak, że metadane można wywnioskować tylko na podstawie wierzchołka oraz jego dzieci (niestety z powodu tego ograniczenia nie można pisać np. własnej opóźnionej propagacji, co można robić za pomocą napisanego własnoręcznie BST). Aby udostępniać jakąś funkcjonalność, musimy też napisać odpowiednie funkcje. Żeby móc tego dokonać, dostajemy dostęp do wewnętrznego interfejsu drzewa.

Najpierw zastanówmy się jak będzie wyglądać implementacja. Metadanymi będzie ilość wierzchołków w poddrzewie. Wtedy ilość wierzchołków w lewym poddrzewie to ilość elementów mniejszych. Na podstawie tego można wymyślić wszystkie potrzebne nam operacje.

A oto przykład implementacji order statistics tree (używa się jej tak jak wbudowanej):

▸ Implementacja order statistics tree

- `6.` – będzie nam potrzebne `branch_policy`, bo piszemy zachowanie dla struktury drzewiastej.
- `8:10.` – ponieważ `NodeUpdate` to szablon, musimy odpowiednio nazwać parametry. Co ważne, występuje tu rozróżnienie między iteratorem do wierzchołka (`node_iterator`) i iteratorem do elementu

(`iterator`). To pierwsze jest niższego poziomu i pozwala na czytanie i edycję metadanych oraz dostęp do lewego i prawego dziecka wierzchołka. W szablonie dostajemy const-iterator i iterator do wierzchołka, typ funkcji porównującej, oraz typ alokatora.
- `12:16.` – kilka typedefów. Najważniejszy jest `metadata_type`, czyli typ metadanych. Sami go sobie wybieramy. Pozostałe napisałem, bo później przydają się w kodzie. Są to typedefy rzeczy z zewnętrznego interfejsu `tree`.
- `18:22.` – Funkcje wirtualne, ustawiane nam przy konstrukcji przez naszego właściciela (czyli pewnego `tree`). Są to funkcje zwracające iteratory na wierzchołkach i instancję funkcji porównującej.
- `23.` – Jeżeli jest to nam potrzebne, możemy napisać własny destruktor.
- `26:32.` – Aktualizację metadanych piszemy za pomocą `operator()`. Jako parametry dostajemy iterator do aktualizowanego wierzchołka oraz wierzchołkowy iterator końcowy. Mamy gwarancję, że gdy aktualizowane są metadane wierzchołka, to metadane jego dzieci zostały już zaktualizowane. Kilka metod interfejsu `node_iterator`:
  - `const metadata_type& node_iterator::get_metadata()` – zwraca metadane.
  - `node_iterator node_iterator::get_l_child()` oraz `get_r_child()` – zwraca wskaźniki do lewego oraz prawego dziecka wierzchołka. Jeżeli wierzchołek nie ma dziecka, to zwracany jest wskaźnik końcowy (w przykładzie `end_it`).

  Robię tutaj kilka `const_cast`-ów, bo z taką praktyką spotkałem się w oryginalnej implementacji. Wygląda na to, że domyślnie `get_metadata` zwraca const-referencję, więc trzeba pozbyć się const, aby zmienić metadane.
- `33:51.` – mamy tutaj przykład wykorzystania `node_begin` i `node_end`. Warto zauważyć, że napisane w ten sposób funkcje będą w dokładnie ten sam sposób dostępne z poziomu `tree`.
- `52:74.` – tutaj jest przykład ekstrakcji funkcji porównującej i wyciągania klucza wierzchołka z iteratora. `extract_key` istnieje, ponieważ normalnie dereferencja iteratora może zwrócić parę, ale gdy wartość jest typu `null_type`, to nie zwróci nam pary.

Rozszerzenia kompilatora

Notatka: większość funkcji postaci `__builtin_[...]` jest dostępna globalnie i nie wymaga żadnego pliku nagłówkowego.

Mimo tego, że rozszerzenia kompilatora nie mają zbyt dużo wspólnego z samym libstdc++, uznałem, że tematycznie tutaj pasują.

Zaawansowane operacje bitowe

Czasami potrzebujemy czegoś bardziej skomplikowanego niż zwykłe operacje bitowe. W GCC do dyspozycji dostajemy bliźniacze funkcje: zliczanie zer wiodących oraz kończących w reprezentacji bitowej liczby. Funkcje te (kryją się pod nazwami `int __builtin_clz(int x)` (count leading zeros) oraz `int __builtin_ctz(int x)` (count trailing zeros)). Należy wziąć pod uwagę, że zachowanie funkcji dla `0` jest niezdefiniowane.

Wynika to z tego, że zwykle te funkcje kompilator zapisze jako wywołanie odpowiedniej instrukcji procesora (na systemach gdzie ta instrukcja nie jest dostępna – kompilator ma swoją zastępczą implementację), a różne procesory mogą różnie obsługiwać ten przypadek.

Ponieważ domyślnie działają na `int`, funkcje występują w odmianach pracujących na `long` i `long long`, i aby je wydobyć należy (w stylu C) do nazwy funkcji dodać sufiks `l` lub `ll`.

Do czego mogą nam się przydać te operacje? Przede wszystkim `clz` okaże się bardzo użyteczne. Okazuje się, że dla danej dodatniej liczby n indeks najbardziej znączącej jedynki (indeks, czyli wykładnik potęgi, którą reprezentuje) jest jednocześnie równy $\lfloor \log_2 n \rfloor$. A skąd wziąć ten indeks? Jest to oczywiście ilość wszystkich bitów (minus jeden) odjąć ilość zer wiodących. Na podstawie tego możemy sobie napisać taką oto funkcję:

```
uint32_t log2floor(uint32_t x)
{
    return 31 - __builtin_clz(x);
}
```

A jak otrzymać sufit z logarytmu dwójkowego, który też czasami jest potrzebny? (Na przykład przy drzewie przedziałowym - w ten sposób otrzymamy wysokość tudzież ilość liści). Tym razem okazuje się, że dla liczb całkowitych spełniona jest tożsamość $\lceil \log_2 n \rceil = \lfloor \log_2 (2n-1) \rfloor$ (Można to sobie uzasadnić techniką 'dodaj jak najwięcej, ale nie tak, żeby zmienić wynik', której używamy przy dzieleniu z sufitem zamiast podłogą - $\lceil \frac{a}{b} \rceil = \lfloor \frac{a+b-1}{b} \rfloor$).

Słowem komentarza: funkcja `log2floor` jest dostępna jako **`std::__lg`** w `<algorithm>`. Jest ona szablonem *poprawnie* obsługującym wszystkie typy całkowite co najmniej wielkości `int` – warto pamiętać.

Poza zliczaniem zer można też zliczać wszystkie jedynki. Funkcja ta, zwana za morzem `popcount` jest nam dana jako **`int __builtin_popcount(int x)`**. Także działa na `int` i ma odmiany do innych typów z odpowiednimi sufiksami. Analogicznie też jest zastąpywana instrukcją, o ile jest to możliwe. Przydaje się np. do liczenia wielkości pewnego podzbioru przy pracy na bitmaskach.

Wielkie nadzieje

Poza w miarę logicznymi operacjami bitowymi, mamy też specjalne funkcje dające kompilatorowi więcej informacji, które mają mu ułatwić optymalizację kodu. Z tych łatwiejszych do użycia mamy **`long`** `__builtin_expect(`**`long`** `value, `**`long`** `expected)` (wraz z **`long`** `__builtin_expect_with_probability(`**`long`** `value, `**`long`** `expected, `**`double`** `probability)`) oraz **`void`** `__builtin_unreachable();`.

Unreachable wywołujemy, gdy chcemy kompilator zapewnić, że program *nigdy* nie dojdzie do pewnego miejsca w kodzie. Na przykład, jeżeli chcemy znaleźć w pewnym ciągu liczbę 0, a jesteśmy pewni że owa w nim występuje, i przechodzimy pętlą, returnując indeks jeżeli znaleźliśmy owe 0, to po pętli można dać `__builtin_unreachable();` - bo zawsze

returnujemy przed zakończeniem pętli.

Natomiast expect wykorzystujemy, aby dać kompilatorowi wskazówkę, jak zoptymalizować pewien wyjątek prowadzący do rozgałęzienia (tak zwany branching). Na przykład, jeżeli jesteśmy pewni, że warunek zachodzi właściwie zawsze (a jednak czasami nie), to używamy właśnie `__builtin_expect`. Zwróci nam niezerową wartość, jeżeli warunek zachodzi.
Tematyczny przykład: w funkcji `log2floor`, aby zdefiniować wartość wywołania dla `0` można użyć `__builtin_expect(x != 0, true) ? 31 - __builtin_clz(x) : -1u`, zakładając, że funkcja prawie nigdy nie będzie wywoływana z `0`. W porównaniu z `x != 0 ? 31 - __builtin_clz(x) : -1u`: kompilator wygeneruje kod, w którym procesor, zamiast domyślnie najpierw ładować na stos `-1u`, będzie preferował liczenie clz, oraz zamiast jumpować do miejsca, gdzie liczone jest clz, będzie jumpował do miejsca gdzie zwracane jest `-1u`. Teoretycznie rzecz biorąc, kod został lepiej zoptymalizowany - jumpy są kosztowne, lepiej żeby były wykorzystywane w przypadku rzadszym. Inny przykład to przy modulo przez liczby Mersenne'a ('O modulo'): można w ten sposób wyifować przypadek, kiedy x jest wielokrotnością $2^k - 1$.
Domyślne wywołanie zakłada, że warunek zachodzi w 90% przypadków. Jeżeli chcemy założyć inne prawdopodobieństwo, należy użyć `__builtin_expect_with_probability` (w analogiczny sposób jak wersję zwykłą), i podać prawdopodobieństwo jako liczbę od 0 do 1. Podane prawdopodobieństwo pozwala kompilatorowi na ocenę, jaką optymalizację zastosować.

Bibliografia

- https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/ – oficjalna 'dokumentacja' Policy-Based Data Structures
- https://medium.com/omarelgabrys-blog/the-big-scary-o-notation-ce9352d827ce – notacja wielkiego O
- https://codeforces.com/blog/entry/60737 – o hashmapach z PBDS
- https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html – oficjalna dokumentacja GCC o wbudowanych funkcjach związanych z działaniem kompilatora (`__builtin`)