Subject: get_i
Posted by mirek on Sun, 14 Jun 2020 17:10:00 GMT
View Forum Message <> Reply to Message

A complemement to findarg and decode:

get_i(-1, "zero", "one", "two") = zero
get_i(0, "zero", "one", "two") = zero
get_i(2, "zero", "one", "two") = two
get_i(3, "zero", "one", "two") = two

Subject: Re: get_i
Posted by Novo on Tue, 16 Jun 2020 15:45:38 GMT
View Forum Message <> Reply to Message

Thank you.
I played a little bit with get_i and godbolt.org and got results below.
Test: const char* c = get_i(-1, "zero", "one", "two");
Assembly for the original code (-O2):

```
.LC0:
        .string "zero"
_GLOBAL__sub_I_c:
        mov     QWORD PTR c[rip], OFFSET FLAT:.LC0
        ret
c:
        .zero   8
```

I changes U++ code a little bit:

```
template <class T> constexpr const T& min(const T& a, const T& b) { return a < b ? a : b; }
template <class T> constexpr const T& max(const T& a, const T& b) { return a > b ? a : b; }

template <class T> // deprecated name, use clamp
constexpr T minmax(T x, T _min, T _max) { return min(max(x, _min), _max); }

template <class T>
constexpr T clamp(T x, T _min, T _max) { return minmax(x, _min, _max); }

inline constexpr const char *get_i(int i, const char *p0, __List##I(E__NFValue)) \
```

Resulting assenbly:

```
.LC0:
        .string "zero"
c:
        .quad   .LC0
```

Conclusion: "constexpr" is quite useful ...

Subject: Re: get_i
Posted by mirek on Tue, 16 Jun 2020 15:52:49 GMT
View Forum Message <> Reply to Message

I am not 100% if above is an experiment or suggestion to change things.

If later, then the obvious counterargument is that what I wrote is just demonstration, in real life the expression will not be constant...

Mirek

Subject: Re: get_i
Posted by Novo on Tue, 16 Jun 2020 16:02:22 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 16 June 2020 11:52I am not 100% if above is an experiment or suggestion to change things.

If later, then the obvious counterargument is that what I wrote is just demonstration, in real life the expression will not be constant...

Mirek
constexpr behaves as normal function when expression is not a constant ...
So, it will work.
It is a suggestion to change things ...
It is a suggestion to use constexpr ...

Subject: Re: get_i
Posted by Novo on Tue, 16 Jun 2020 16:21:00 GMT
View Forum Message <> Reply to Message

Another experiment/suggestion.
I rewrote get_i using variadic template:

```
template <typename A, typename... T>
constexpr A get_i(int i, const A& p0, const T& ...args)
{
 A list[] = {p0, args...};
 int n = sizeof...(args);
 return list[clamp(i, 0, n)];
}
```

```
const char* cr = get_i(1, "zero", "one", "two");
RDUMP(cr);
int ir = get_i(1, 0, 1, 2);
RDUMP(ir);
```

IMHO, my implementation is much shorter and it will compile faster.
IMHO, macroses __List and __Expand are not needed anymore ...

Subject: Re: get_i
Posted by mirek on Tue, 16 Jun 2020 19:20:19 GMT
View Forum Message <> Reply to Message

Novo wrote on Tue, 16 June 2020 18:21Another experiment/suggestion.
I rewrote get_i using variadic template:

```
template <typename A, typename... T>
constexpr A get_i(int i, const A& p0, const T& ...args)
{
 A list[] = {p0, args...};
 int n = sizeof...(args);
 return list[clamp(i, 0, n)];
}

const char* cr = get_i(1, "zero", "one", "two");
RDUMP(cr);
int ir = get_i(1, 0, 1, 2);
RDUMP(ir);
```

IMHO, my implementation is much shorter and it will compile faster.
IMHO, macroses __List and __Expand are not needed anymore ...

Yes, you are right about this, I have used old tricks mostly out of habit. I guess I will have to rewrite it all now

However, constexpr I still do not agree. Following your logic, we should add constexpr to every single function everywhere - these are as likely to have constant parameters as get_i (which has like 0.00000001% chance that first parameter will be const in real code).

Subject: Re: get_i
Posted by Novo on Tue, 16 Jun 2020 19:22:34 GMT
View Forum Message <> Reply to Message

Another implementation using initializer_list:

```
template <typename T>
constexpr T get_i(int i, std::initializer_list<T> list)
```

```
{
 return list[clamp(i, 0, list.size())];
}
```

This one is trictly-typed, although I couldn't check assembly with godbolt because it complains about something ...

Subject: Re: get_i
Posted by mirek on Tue, 16 Jun 2020 19:25:08 GMT
View Forum Message <> Reply to Message

Novo wrote on Tue, 16 June 2020 21:22Another implementation using initializer_list:

```
template <typename T>
constexpr T get_i(int i, std::initializer_list<T> list)
{
 return list[clamp(i, 0, list.size())];
}
```

This one is trictly-typed, although I couldn't check assembly with godbolt because it complains about something ...

Nah, we do not want strict typing here.

Subject: Re: get_i
Posted by Novo on Tue, 16 Jun 2020 19:42:23 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 16 June 2020 15:25Novo wrote on Tue, 16 June 2020 21:22Another implementation using initializer_list:

```
template <typename T>
constexpr T get_i(int i, std::initializer_list<T> list)
{
 return list[clamp(i, 0, list.size())];
}
```

This one is trictly-typed, although I couldn't check assembly with godbolt because it complains about something ...

Nah, we do not want strict typing here.
Sorry, last one won't compile.
The one using variadic template is fine, although it still needs specialization for const char* ... :-/

Subject: Re: get_i

Posted by Novo on Tue, 16 Jun 2020 20:15:37 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 16 June 2020 15:20
However, constexpr I still do not agree. Following your logic, we should add constexpr to every single function everywhere - these are as likely to have constant parameters as get_i (which has like 0.00000001% chance that first parameter will be const in real code).
You cannot add constexpr to every single function everywhere. There are restrictions ...
But, IMHO, function, which can be compiled with constexpr, should have it ...
At this time you are not using functions in compile-time context because you just cannot do that.

I personally often write code like this:
```
enum e {
  e01 = 1,
  e02 = 100,
  e03 = e01 + e02
};
```
In case of constexpr functions I'll be able to write this:
```
enum e {
  e01 = min(something, something_else),
  e02 = max(something, something_else)
};
```
Another observation: template functions/methods are inline by default.

Subject: Re: get_i
Posted by mirek on Tue, 16 Jun 2020 22:01:23 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 16 June 2020 21:20Novo wrote on Tue, 16 June 2020 18:21Another experiment/suggestion.
I rewrote get_i using variadic template:

```
template <typename A, typename... T>
constexpr A get_i(int i, const A& p0, const T& ...args)
{
 A list[] = {p0, args...};
 int n = sizeof...(args);
 return list[clamp(i, 0, n)];
}

const char* cr = get_i(1, "zero", "one", "two");
RDUMP(cr);
int ir = get_i(1, 0, 1, 2);
RDUMP(ir);
```

IMHO, my implementation is much shorter and it will compile faster.
IMHO, macroses __List and __Expand are not needed anymore ...

Yes, you are right about this, I have used old tricks mostly out of habit. I guess I will have to rewrite it all now

Do you see any problems?

```
template <class T, class V>
constexpr V decode(const T& sel, const V& def)
{
 return def;
}

template <class T>
constexpr const char *decode(const T& sel, const char *def)
{
 return def;
}

template <class T, class K, class V, typename... L>
constexpr V decode(const T& sel, const K& k, const V& v, const L& ...args)
{
 return sel == k ? v : (V)decode(sel, args...);
}

template <class T, class K, typename... L>
constexpr const char *decode(const T& sel, const K& k, const char *v, const L& ...args)
{
 return sel == k ? v : decode(sel, args...);
}


template <class T, class K>
constexpr int findarg(const T& x, const K& k)
{
 return x == k ? 0 : -1;
}

template <class T, class K, typename... L>
constexpr int findarg(const T& sel, const K& k, const L& ...args)
{
 if(sel == k)
  return 0;
 int q = findarg(sel, args...);
 return q >= 0 ? q + 1 : -1;
}
```

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 04:39:56 GMT
View Forum Message <> Reply to Message

Novo wrote on Tue, 16 June 2020 15:42mirek wrote on Tue, 16 June 2020 15:25Novo wrote on Tue, 16 June 2020 21:22Another implementation using initializer_list:

```
template <typename T>
constexpr T get_i(int i, std::initializer_list<T> list)
{
 return list[clamp(i, 0, list.size())];
}
```

This one is trictly-typed, although I couldn't check assembly with godbolt because it complains about something ...

Nah, we do not want strict typing here.
Sorry, last one won't compile.
The one using variadic template is fine, although it still needs specialization for const char* ... :-/
Fixed version. No performance degradation.

```
template <typename T>
constexpr T get_i2(int i, const std::initializer_list<T>& list)
{
 const int n = list.size();
 return *(list.begin() + clamp(i, 0, n));
}

const char* c = get_i2(1, {"zero", "one", "two"});
```

Assembler:

```
.L.str:
        .asciz  "one"

c:
        .quad   .L.str
```

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 05:01:04 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 16 June 2020 18:01
Do you see any problems?

```
 int ind = findarg(1, "0", 1.5, 2, 3);
 RDUMP(ind);
```

ind = 1 in my case ...

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 05:09:57 GMT
View Forum Message <> Reply to Message

Novo wrote on Wed, 17 June 2020 01:01mirek wrote on Tue, 16 June 2020 18:01
Do you see any problems?

```
 int ind = findarg(1, "0", 1.5, 2, 3);
 RDUMP(ind);
```

ind = 1 in my case ...
Sorry, I was testing against current implementation in U++ again.
New implementation is fine.

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 05:23:13 GMT
View Forum Message <> Reply to Message

Well, it is "fine" because "findarg(1, "0", 1.5, 2, 3)" won't compile ...
But if you need a heterogeneous set of arguments, then you need to implement it differently ...

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 05:50:16 GMT
View Forum Message <> Reply to Message

Code below works for all data types, including const char*.

```
template <class T, class V>
constexpr auto decode(const T& sel, const V& def)
{
 return def;
}

template <class T, class K, class V, typename... L>
constexpr auto decode(const T& sel, const K& k, const V& v, const L& ...args)
{
 return sel == k ? v : decode(sel, args...);
}
```

Subject: Re: get_i
Posted by mirek on Wed, 17 Jun 2020 07:35:05 GMT
View Forum Message <> Reply to Message

Novo wrote on Wed, 17 June 2020 07:23Well, it is "fine" because "findarg(1, "0", 1.5, 2, 3)" won't

compile ...
But if you need a heterogeneous set of arguments, then you need to implement it differently ...

Well, I really think above one should not compile... Heterogenous yes, but arguments must be comparable...

Mirek

Subject: Re: get_i
Posted by mirek on Wed, 17 Jun 2020 11:03:35 GMT
View Forum Message <> Reply to Message

So I started looking into eliminating all instances of Expand macro usage and identified that following helpers could be quite useful:

```
template <class I, class V>
void iter_set(I t, V&& v)
{
 *t++ = v;
}

template <class I, class V, typename... Args>
void iter_set(I t, V&& v, Args&& ...args)
{
 *t++ = v;
 iter_set(t, args...);
}

template <class C, typename... Args>
C gather(Args&& ...args)
{
 C x;
 x.SetCount(sizeof...(args));
 iter_set(x.begin(), args...);
 return x;
}

template <class I, class V>
void iter_get(I s, V& v)
{
 v = *s++;
}

template <class I, class V, typename... Args>
void iter_get(I s, V& v, Args& ...args)
{
```

```
 v = *s++;
 iter_get(s, args...);
}

template <class C, typename... Args>
int scatter(int n, const C& c, Args& ...args)
{
 if(n < sizeof...(args))
  return 0;
 iter_get(c.begin(), args...);
 return sizeof...(args);
}

template <class C, typename... Args>
int scatter(const C& c, Args& ...args)
{
 return scatter(c.GetCount(), c, args...);
}
```

Usage example:

```
template <typename... Args>
String Format(const char *fmt, const Args& ...args)
{
 return Format(fmt, gather<Vector<Value>>(args...));
}
```

But I guess this would work even better if containers interface was amended to be more "std" (Vector::Vector(int count), size() synonyme for GetCount), so I guess that needs a bit more work...

Mirek

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 17:00:03 GMT
View Forum Message <> Reply to Message

A fix:

```
 template <class I, class V>
 void iter_set(I t, V&& v)
 {
  *t++ = std::forward<V>(v);
 }

 template <class I, class V, typename... Args>
```

```
 void iter_set(I t, V&& v, Args&& ...args)
 {
  *t++ = std::forward<V>(v);
  iter_set(t, std::forward<Args>(args)...);
 }

 template <class C, typename... Args>
 C gather(Args&& ...args)
 {
  C x;
  x.SetCount(sizeof...(args));
  iter_set(x.begin(), std::forward<Args>(args)...);
  return x;
 }


template <typename... Args>
String Format(const char *fmt, Args&& ...args)
{
 return Format(fmt, gather<Vector<Value>>(std::forward<Args>(args)...));
}
```

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 17:14:14 GMT
View Forum Message <> Reply to Message

Another fix to avoid extra-copying ...

```
 template <class I, class V>
 void iter_set(I& t, V&& v)
 {
  *t = std::forward<V>(v);
 }

 template <class I, class V, typename... Args>
 void iter_set(I& t, V&& v, Args&& ...args)
 {
  *t++ = std::forward<V>(v);
  iter_set(t, std::forward<Args>(args)...);
 }

 template <class C, typename... Args>
 C gather(Args&& ...args)
 {
  C x;
  x.SetCount(sizeof...(args));
  auto iter = x.Begin();
  iter_set(iter, std::forward<Args>(args)...);
```

```
 return x;
}
```

Subject: Re: get_i
Posted by mirek on Wed, 17 Jun 2020 19:03:19 GMT
View Forum Message <> Reply to Message

Novo wrote on Wed, 17 June 2020 19:14Another fix to avoid extra-copying ...

```
template <class I, class V>
void iter_set(I& t, V&& v)
{
 *t = std::forward<V>(v);
}

template <class I, class V, typename... Args>
void iter_set(I& t, V&& v, Args&& ...args)
{
 *t++ = std::forward<V>(v);
 iter_set(t, std::forward<Args>(args)...);
}

template <class C, typename... Args>
C gather(Args&& ...args)
{
 C x;
 x.SetCount(sizeof...(args));
 auto iter = x.Begin();
 iter_set(iter, std::forward<Args>(args)...);
 return x;
}
```

Thanks. Trunk version are a bit different now, can you review?

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 21:21:29 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 17 June 2020 15:03
Thanks. Trunk version are a bit different now, can you review?
In your last commit you've added std::forward not to all places.
I've attached my version.
I also changed a functor signature in "void iter_get(I s, Args& ...args)".
Just a reference is fine there ...

IMHO, functors and iterators should be passed by reference to avoid extra-copying ...

File Attachments
1) Fn.h, downloaded 295 times

Subject: Re: get_i
Posted by Novo on Wed, 17 Jun 2020 21:59:57 GMT
View Forum Message <> Reply to Message

A more generic version of
```
template <typename... T>
constexpr const char *get_i(int i, const char* p0, const T& ...args)
{
 const char *list[] = { p0, args... };
 return list[clamp(i, 0, (int)sizeof...(args))];
}

 template <typename A, typename... T>
 constexpr A* get_i(int i, A* p0, const T& ...args)
 {
  A* list[] = { p0, args... };
  return list[clamp(i, 0, (int)sizeof...(args))];
 }
```

Example:
```
 const char* cr = get_i(1, "0", "11", "222");
 RLOG(cr);
 cr = get_i(1, "0", String("11"), "222");
 RLOG(cr);
 const wchar _0[] = {0};
 const wchar _3[] = {2, 2, 2};
 const wchar* wcr = get_i(1, _0, WString("11"), _3);
 RLOG(wcr);
```

Subject: Re: get_i
Posted by mirek on Thu, 18 Jun 2020 06:39:32 GMT
View Forum Message <> Reply to Message

Thanks, applied...

Mirek

Subject: Re: get_i
Posted by Novo on Thu, 18 Jun 2020 21:56:23 GMT
View Forum Message <> Reply to Message

mirek wrote on Thu, 18 June 2020 02:39Thanks, applied...

Mirek
No problem.
There is still a bug with
```
template <typename P, typename... T>
constexpr const P *get_i(int i, const P* p0, const T& ...args)
{
 const char *list[] = { p0, args... };
 return list[clamp(i, 0, (int)sizeof...(args))];
}
```

It has to look like below.
```
template <typename P, typename... T>
constexpr const P *get_i(int i, const P* p0, const T& ...args)
{
 const P *list[] = { p0, args... };
 return list[clamp(i, 0, (int)sizeof...(args))];
}
```

Subject: Re: get_i
Posted by mirek on Mon, 29 Jun 2020 17:23:31 GMT
View Forum Message <> Reply to Message

I have just found that this fails with Visual C++ compiler:

```
 String n = " 2";
 ASSERT(decode(4, 1, "one", 2, "two", 3, "three", "unknown" + n) == String("unknown 2"));
```

The problem is that temporaty object gets destroyed too early...

I _believe_ this is a compiler error. In any case, it is pretty bad.

EDIT: Not a compiler bug. The problem is in your decode. Fixed it with

```
template <class T, class V>
constexpr const V& decode(const T& sel, const V& def)
{
 return def;
```

```
}

template <class T, class K, class V, typename... L>
constexpr const V& decode(const T& sel, const K& k, const V& v, const L& ...args)
{
 return sel == k ? v : decode(sel, args...);
}

template <class T>
constexpr const char *decode(const T& sel, const char *def)
{
 return def;
}

template <class T, class K, typename... L>
constexpr const char *decode(const T& sel, const K& k, const char *v, const L& ...args)
{
 return sel == k ? v : (const char *)decode(sel, args...);
}
```

Subject: Re: get_i
Posted by Novo on Thu, 02 Jul 2020 20:08:36 GMT
View Forum Message <> Reply to Message

Sorry for the late response.
My code is correct. Temporary String lives only during function call. This is how C++ works.
Return type is a value, not a reference. So, no temporaries ...

"The type of the ternary ?: expression is the common type of its second and third argument. If both types are the same, you get a reference back. If they are convertable to each other, one gets chosen and the other gets converted (promoted in this case). Since you can't return an lvalue reference to a temporary (the converted / promoted variable), its type is a value type."

Basically, the ternary ?: is needed to convert "const char[N]" and "const char[M]" to "const char*".

On the other side, templates is a complicated thing.
If you have a non-template version of "decode" declared before template instantiation point, compiler will choose it ...
Also MSVC is very well known for broken "two-phase name lookup". Even till these days, I believe ...
IMHO, a safer version would look like this:

```
namespace details {
 template <class T, class V>
 constexpr auto decode(const T& sel, const V& def)
 {
  return def;
```

```
 }

 template <class T, class K, class V, typename... L>
 constexpr auto decode(const T& sel, const K& k, const V& v, const L& ...args)
 {
  return sel == k ? v : details::decode(sel, args...);
 }
}

template <class T, class K, class V, typename... L>
constexpr auto decode(const T& sel, const K& k, const V& v, const L& ...args)
{
 return details::decode(sel, k, v, args...);
}
```

Subject: Re: get_i
Posted by mirek on Fri, 03 Jul 2020 07:38:07 GMT
View Forum Message <> Reply to Message

Novo wrote on Thu, 02 July 2020 22:08Sorry for the late response.
My code is correct. Temporary String lives only during function call. This is how C++ works.

I would not be fixing if it was correct. Whole thing was actual error in actual application.

Indeed, temp string lives only during function call. What happened here is that in some circustances when mixing String and const char * parameters, const char * gets converted to String temporary, then back to const char *, then temporary is detroyed and dangling const char * returned.

Quote:
Return type is a value, not a reference. So, no temporaries ...

And that is exactly the problem. That return value is temporary one level up and gets converted to const char *....

It is very tricky indeed. Actually the version posted here was not final, it needed more fixes for other situations (namely enums). Hopefully trunk version is now ok. Full test is in "autotest/decode". Also the error only appears with MSC, but I have checked, the problem is not in the compiler.

Subject: Re: get_i
Posted by Novo on Fri, 03 Jul 2020 16:53:00 GMT

View Forum Message <> Reply to Message

I guess that the problem is that in case of MSVC common type of const char* and String is const char*, and in case of Clang it is String.

Subject: Re: get_i
Posted by mirek on Fri, 03 Jul 2020 17:03:22 GMT
View Forum Message <> Reply to Message

Novo wrote on Fri, 03 July 2020 18:53I guess that the problem is that in case of MSVC common type of const char* and String is const char*, and in case of Clang it is String.

Yes, however that does not break C++ specification... BTW, I had to do another fix today... Converting decode to vararg templates is really pandora's box...