# Part 42: Type Casting and NULL

In this part of our compiler writing journey, I've implemented type casting. I thought this would allow me to do:

```
#define NULL (void *)0
```

but I hadn't done enough to get `void *` to work properly. So I've added type casting and also got `void *` to work.

## What is Type Casting?

Type casting is where you forcibly change the type of an expression to be something else. Common reasons are to narrow an integer value down to a smaller range type, or to assign a pointer from one type into a pointer storage of another type, e.g.

```
int   x= 65535;
char  y= (char)x;     // y is now 255, the lower 8 bits
int  *a= &x;
char *b= (char *)a;   // b point at the address of x
long *z= (void *)0;   // z is a NULL pointer, not pointing at anything
```

Notice above that I've used the casts in assignment statements. For expressions within functions, we will need to add an A_CAST node to our AST tree to say "cast the original expression type to this new type".

For global variable assignments, we will need to modify the assignment parser to allow a cast to come before the literal value.

## A New Function, `parse_cast()`

I've added this new function in `decl.c` :

```
// Parse a type which appears inside a cast
int parse_cast(void) {
  int type, class;
  struct symtable *ctype;

  // Get the type inside the parentheses
  type= parse_stars(parse_type(&ctype, &class));

  // Do some error checking. I'm sure more can be done
  if (type == P_STRUCT || type == P_UNION || type == P_VOID)
    fatal("Cannot cast to a struct, union or void type");
  return(type);
}
```

The parsing of the surrounding '(' ... ')' is done elsewhere. We get the type identifier and the following '*' tokens to get the type of the cast. Then we prevent casts to structs, unions and to `void` .

We need a function to do this as we have to do it in expressions and also in global variable assignments. I didn't want any DRY code.

## Cast Parsing in Expressions

We already parse parentheses in our expression code, so we will need to modify this. In `primary()` in `expr.c` , we now do this:

```
static struct ASTnode *primary(void) {
  int type=0;
  ...
  switch (Token.token) {
  ...
    case T_LPAREN:
      // Beginning of a parenthesised expression, skip the '('.
      scan(&Token);


      // If the token after is a type identifier, this is a cast expression
      switch (Token.token) {
        case T_IDENT:
          // We have to see if the identifier matches a typedef.
          // If not, treat it as an expression.
          if (findtypedef(Text) == NULL) {
            n = binexpr(0); break;
          }
        case T_VOID:
        case T_CHAR:
        case T_INT:
        case T_LONG:
        case T_STRUCT:
        case T_UNION:
        case T_ENUM:
          // Get the type inside the parentheses
          type= parse_cast();

          // Skip the closing ')' and then parse the following expression
          rparen();

        default: n = binexpr(0); // Scan in the expression
      }

      // We now have at least an expression in n, and possibly a non-zero type i
      // if there was a cast. Skip the closing ')' if there was no cast.
      if (type == 0)
        rparen();
      else
        // Otherwise, make a unary AST node for the cast
        n= mkastunary(A_CAST, type, n, NULL, 0);
      return (n);
    }
  }
```

That's a lot to digest, so let's go through it in stages. All of the cases ensure that we have a type identifier after the '(' token. We call `parse_cast()` to get the cast type and parse the ')' token.

We don't have an AST tree to return yet because we don't know which expression we are casting. So we fall through to the default case where the next expression is parsed.

At this point either `type` is still zero (no cast) or non-zero (there was a cast). If no cast, the right parenthesis has to be skipped and we can simply return the expression in parentheses.

If there was a cast, we build an A_CAST node with the new `type` and with the following expression as the child.

## Generating the Assembly Code for a Cast

Well, we are lucky because the expression's value will be stored in a register. So if we do:

```
int   x= 65535;
char  y= (char)x;     // y is now 255, the lower 8 bits
```

then we can simply put the 65535 into a register. But when we save it to y, then the lvalue's type will be invoked to generate the correct code to save the right size:

```
        movq    $65535, %r10            # Store 65535 in x
        movl    %r10d, -4(%rbp)
        movslq  -4(%rbp), %r10          # Get x into %r10
        movb    %r10b, -8(%rbp)         # Store one byte into y
```

So, in `genAST()` in `gen.c`, we have this code to deal with casting:

```
...
leftreg = genAST(n->left, NOLABEL, NOLABEL, NOLABEL, n->op);
...
switch (n->op) {
  ...
  case A_CAST:
    return (leftreg);         // Not much to do
  ...
}
```

## Casts in Global Assignments

The above is fine when the variables are local variables, as the compiler does the above assignments as expressions. For global variables, we have to hand-parse the cast and apply it to a literal value that follows it.

So, for example, in `scalar_declaration` in `decl.c` we need this code:

```
    // Globals must be assigned a literal value
    if (class == C_GLOBAL) {
      // If there is a cast
      if (Token.token == T_LPAREN) {
        // Get the type in the cast
        scan(&Token);
        casttype= parse_cast();
        rparen();

        // Check that the two types are compatible. Change
        // the new type so that the literal parse below works.
        // A 'void *' casttype can be assigned to any pointer type.
        if (casttype == type || (casttype== pointer_to(P_VOID) && ptrtype(type
          type= P_NONE;
        else
          fatal("Type mismatch");
      }

      // Create one initial value for the variable and
      // parse this value
      sym->initlist= (int *)malloc(sizeof(int));
      sym->initlist[0]= parse_literal(type);
      scan(&Token);
    }
```

First of all, note that we set `type= P_NONE` when there is a cast, and we call `parse_literal()` with P_NONE when there is a cast. Why? Because this function used to required that the literal being parsed was exactly the type which was the argument, i.e. a string literal had to be of type `char *`, a `char` had to be matched by a literal in the range 0 ... 255 etc.

Now that we have a cast, we should be able to accept:

```
char a= (char)65536;
```

So the code in `parse_literal()` in `decl.c` now does this:

```
int parse_literal(int type) {

  // We have a string literal. Store in memory and return the label
  if (Token.token == T_STRLIT) {
    if (type == pointer_to(P_CHAR) || type == P_NONE)
    return(genglobstr(Text));
  }
```

```
    // We have an integer literal. Do some range checking.
    if (Token.token == T_INTLIT) {
      switch(type) {
        case P_CHAR: if (Token.intvalue < 0 || Token.intvalue > 255)
                       fatal("Integer literal value too big for char type");
        case P_NONE:
        case P_INT:
        case P_LONG: break;
        default: fatal("Type mismatch: integer literal vs. variable");
      }
    } else
      fatal("Expecting an integer literal value");
    return(Token.intvalue);
  }
```

and the P_NONE is used to relax the type restrictions.

## Dealing with void *

A `void *` pointer is one that can be used in place of any other pointer type. So we have to implement this.

We already did this for global variable assignments above:

```
     if (casttype == type || (casttype== pointer_to(P_VOID) && ptrtype(type)))
```

i.e. if the types are equal, or if a `void *` pointer is being assigned to a pointer. This allows the following global assignment:

```
    char *str= (void *)0;
```

even though `str` is of type `char *` and not `void *`.

Now we need to deal with `void *` (and other pointer/pointer operations) in expressions. To do this, I had to change `modify_type()` in `types.c`. As a refresher, here is what this function does:

```
  // Given an AST tree and a type which we want it to become,
  // possibly modify the tree by widening or scaling so that
  // it is compatible with this type. Return the original tree
  // if no changes occurred, a modified tree, or NULL if the
```

```
// tree is not compatible with the given type.
// If this will be part of a binary operation, the AST op is not zero.
struct ASTnode *modify_type(struct ASTnode *tree, int rtype, int op);
```

This is the code that widens values, e.g. `int x= 'Q';` to make `x` into a 32-bit value. We also use it for scaling: when we do:

```
int x[4];
int y= x[2];
```

The "2" index is scaled by the size of `int` to be eight bytes offset from the base of the `x[]` array.

So, inside a function, when we write:

```
char *str= (void *)0;
```

we get the AST tree:

```
         A_ASSIGN
          /    \
      A_CAST  A_IDENT
        /       str
    A_INTLIT
        0
```

the type of the left-hand `tree` will be `void *` and the `rtype` will be `char *`. We had better ensure that the operation can be performed.

I've changed `modify_type()` to do this for pointers:

```
// For pointers
if (ptrtype(ltype) && ptrtype(rtype)) {
  // We can compare them
  if (op >= A_EQ && op <= A_GE)
    return(tree);

  // A comparison of the same type for a non-binary operation is OK,
  // or when the left tree is of  `void *` type.
  if (op == 0 && (ltype == rtype || ltype == pointer_to(P_VOID)))
    return (tree);
}
```

Now, pointer comparison is OK but other binary operations (e.g. addition) is bad. A "non-binary operation" means something like an assignment. We can definitely assign between two things of the same type. Now, we can also assign from a `void *` pointer to any pointer.

## Adding NULL

Now that we can deal with `void *` pointers, we can add NULL to our include files. I've added this to both `stdio.h` and `stddef.h` :

```
#ifndef NULL
# define NULL (void *)0
#endif
```

But there was one final wrinkle. When I tried this global declaration:

```
#include <stdio.h>
char *str= NULL;
```

I got this:

```
str:
        .quad   L0
```

because every initialisation value for a `char *` pointer is treated as a label number. So the "0" in the NULL was being turned into an "L0" label. We need to fix this. Now, in `cgglobsym()` in `cg.c` :

```
      case 8:
        // Generate the pointer to a string literal. Treat a zero value
        // as actually zero, not the label L0
        if (node->initlist != NULL && type== pointer_to(P_CHAR) && initvalue !
          fprintf(Outfile, "\t.quad\tL%d\n", initvalue);
        else
          fprintf(Outfile, "\t.quad\t%d\n", initvalue);
```

Yes it's ugly but it works!

## Testing the Changes

I won't go through all the tests themselves, but files `tests/input101.c` to `tests/input108.c` test the above functionality and also the error checking of the compiler.

## Conclusion and What's Next

I thought casting was going to be easy, and it was. What I didn't reckon with was the issues surrounding `void *`. I feel that I've covered most bases here but not all of them, so expect to see some `void *` edge cases that I haven't spotted yet.

In the next part of our compiler writing journey, we'll add some missing operators. Next step