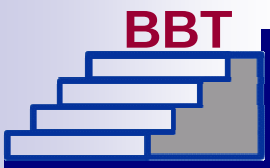

CPP

Fundamentals

Basic data type

char	8	–128 to 127
wchar_t	16	0 to 65,535
int	32	–2^31 to 2^31-1
float	32	3.4E–38 to 3.4E+38
double	64	1.7E–308 to 1.7E+308
bool	N/A	true or false
void		

Some Type Modifier

C++ allows the char, int, and double data types to have modifiers preceding them.

- signed
- unsigned
- long
- short

Some Type Modifier

- char, unsigned char , signed char
- int ,short int, long int

- Float (32), double(64)
- long double (80) 3.4E–4932 to 1.1E+4932

literal

- wchar_t L'A'
- int 1, 123, –234
- long int 35000L, –34L
- unsigned int 10000U, 987U, 40000U
- unsigned long 12323UL, 900000UL
- float 123.23F, 4.34e–3F
- double 23.23, 123123.33, –0.9876324
- long double 1001.2L

Character Escape Sequences

- \b backspace
- \f form feed
- \n newline
- \r carriage return
- \t horizontal tab
- \" double quote
- \' single quote character

Character Escape Sequences

- \\ backslash
- \v vertical tab
- \a alert
- \? ?
- \N octal constant (where N is an octal constant)
- \xN hexadecimal constant (where N is a hexadecimal constant)

Operators

- The precedence of the arithmetic operators is shown here:
- Highest ++, —
- – (unary minus)
- *, /, %
- lowest +, –

Rules to check expression

- Unary operator > binary operator
- Look at right side of a variable. See further to right if it is a unary operator until there is a binary operator.
- The right side unary operator associativity is L to R
- Check far more left side of the variable until it is a binary operator or none .
- The left side unary operator associativity is R to L
- Check binary operator from L to R

Note

```
float x, result;
 x = 1;
 result = x / ++x;
```

- The value of result is not guaranteed to be consistent across different compilers, because it is not clear whether the computer should change the variable x before using it.

- **you should not use the same variable multiple times in a single expression when using operators with side effects.**

Arrays

- type var_name[size]

Array Initialization

type-specifier array_name[size] = {value-list};

- example
- int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
- int i[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
- char str[6] = "hello";
- int sqrs[3][2] = {1, 1, 2, 4, 3, 9}
- int sqrs[3][2] = {{1, 1},{ 2, 4},{ 3, 9}}
- int sqrs[][2] = {{1, 1},{ 2, 4},{ 3, 9}}

Array Initialization -cont.

- int i[10] = {1, 2, 3, 4, 5, 6}; ?

pointer

- type *name
- type *name[size]

- Operator *: example int *p; *p=5;
- Operator &: example: int i; int *p=&i;

Functions

- return_type fun_name(arg1,…)
  - Passing Pointers and Arrays
  - Passing Strings
- Pointer to Function
  - return_type (*pf) (arg1,…)

Define a complex variable

- A declaration can have exactly one basic type, and it's always on the far left of the expression.
- The "basic types" are augmented with "derived types", and C has three of them:
  - * pointer to...
  - [] array of...
  - () function returning...

Rules

- The "array of" [] and "function returning" () type operators have higher precedence than "pointer to" *, and this leads to some fairly straightforward rules for decoding.
- Always start with the variable name
- always end with the basic type
- The "filling in the middle" part is summarize with this rule: "go right when you can, go left when you must"

Example:

long **foo[7];

- *foo is ... long*
- foo is *array of 7* ... long
- foo is array of 7 *pointer to* ... Long
- foo is array of 7 pointer to *pointer to* long

Example

int (*(*foo)())()

- foo is a pointer to function returning pointer to function returning int

Exaple:

- char *(*(**foo [][8])())[];

Answer

- foo is array of array of 8 pointer to pointer to function returning pointer to array of pointer to char

Reference Parameters

- type &name = initial_value;

- Normally used in function or class definition
- Return reference from function can used as lvalue (put at the left side of equation).

Function Overloading

- See example overload.cpp

Default Function Arguments

- See example: defaultparm.cpp

Qualifiers

- const
- volatile

Storage Class Specifiers

- register
- auto
- extern
- static
- mutable

enumeration

- Enumerations are defined using the keyword enum, and this general format:
- enum type-name { enumeration list } variable-list;

Structures

- The general form of a structure declaration is shown here:

```
struct struct-type-name {
  type element_name1;
  type element_name2;
  …
  type element_nameN;
} structure-variables;
```

Example:

```
struct inv_type {
  char item[40]; // name of item
  double cost; // cost
  double retail; // retail price
  int on_hand; // amount on hand
}
```

Bit-Fields

```
struct struct-type-name {
  type name1 : length;
  type name2 : length;
  …
  type nameN : length;
};
```

Example:

```
struct emp {
    struct addr address;
    float pay;
    unsigned lay_off: 1; // lay off or active
    unsigned hourly: 1: // hourly pay or wage
    unsigned deductions: 3: // tax deductions
};
```

Unions

Example:

```
union utype {
    short int i;
    char ch;
} ;
```